CSCI 2132
Software Development

---

# Lecture 12:

# C compared to Java: Expressions and Statements

Instructor: Vlado Keselj

Faculty of Computer Science

Dalhousie University

# Previous Lecture

- Processes and programs
  - threads
  - process Control Block (PCB)
- Process creation
  - fork and exec system calls
- Job control and process control
  - foreground and background processes
  - commands for managing jobs and processes

# C Operators, Expressions, and Statements

- We assume that you know Java well
- Focus on differences between C and java
- **Arithmetic Operators**
- Similar; e.g,, `+`, `-`, `*`, `/`, `%`, `++`, `--`, `=`, `+=` ...
- `%` cannot be applied on floating-point numbers
- Integer operator `/` has an **implementation-defined behaviour** for negative numbers in earlier standards
- C99 defines that division is rounded toward 0
- Concept of **implementation-defined behaviour**

# Expression Evaluation

- In Java: from left to right
- In C: order of expression evaluation is **unspecified behaviour**
- Example:

```
a = 5;
c = (b = a + 2) - (a = 1);
```

# Logical Expressions

- Similar to Java; e.g.,
  - comparison operators: `<`, `>`, `<=`, `>=`, `==`, `!=`
  - logic operators: `!`, `&&`, `||`
- Difference:
  - Java has `boolean` primitive type,
  - C uses '`int`' as a boolean type for true and false
  - a type `bool` similar to Java introduced in C99 standard
  - `int` is still in use, since `bool` is not mandatory
- Use of `int`: 0 is false and 1 is true
- More generally: 0 is false, anything else is true (in `if`, `while`, ...)

# Boolean Interpretation of `int`

- Provides convenient short notation, as in:

```
int f=1, i=n;
while (--i) f *= i+1;
```

- But also traps such as:

```
if (a < i < b) { ... }
```

- and

```
if (x = a + b) { ... }
```

# Short-Circuit Evaluation

- done for `&&` and `||`, as in Java
- Consider example:

```
if (a != 0 && b/a > 2) { ... }
```

# Control Structures

- Similar to Java: `if`, `switch`, `while`, `do-while`, `for`
- Breaking a loop or switch: `break`, but no label
- Continuing a loop: `continue`
- Returning from a function: `return`
- In C but not allowed in Java: `goto` *label*
  - *label*`:` used with a statement
  - local jump, within the same function
- To exit a program: `exit` defined in `stdlib.h`
  - A `return` from the function `main` exists the program as well

# Variable Declaration in 'for' Loop

- Allowed in Java; e.g., `for(int i; i<10; i++) ...`
- Not allowed in C prior to C99
- Allowed in C99 and later

**The Comma Operator**

- Used implicitly in 'for' loops; e.g.,
  `for(i=0, j=0; i<10; i++)...`
- However, it has explicit meaning
- *( expr1, expr2, . . . )* — evaluate *expr1, expr2,* and so on
- Example: `x = (a=3, b=4, c=5);`

# Goto Statement

- Not in Java, although it is a reserved word
- Unconditional jump to any other statement in the same function
- Syntax to define a label
  *identifier*`:` *statement*
  Example: `loop: i++;`
- Syntax of the goto statement:
  `goto` *identifier*
  Example: `goto loop;`

# Example with goto

```
#include <stdio.h>

int main() {
  int i = 1;

  loop: printf("%d\n", i);

  i++;
  if (i <= 10)
    goto loop;
  return 0;
}
```

# Some notes about goto

- Used to be popular (e.g., Basic, Fortran)
- Excessive use leads to “spaghetti” code
  - hard to understand and maintain
- Discouraged in *structured programming*
- Excluded from Java (although kept as a reserved word)
- In C: Jumps within a function

# Typical Uses for goto

- Machine-generated code
- Jumping out of several nested loops and switch-statements; e.g.:

```
while (...) {
  switch(...) {
    ...
    goto loop_done;
  }
  ...
}

loop_done: ...
```

# null Statement

- Simply a semicolon: `;`
- Example:

```
for (d = 2; d < n && n % d != 0; d++)
  ;
if (d < n)
  printf("%d is not a prime number\n");
```

- More efficient loop:

```
for (d = 2; d*d <= n && n % d != 0; d++)
  ;
```

- The same effect of empty loop body:

```
for (d = 2; d*d <= n && n % d != 0; d++)
  { }
```

# C Basic Types: Integer Types

- type `int` with optional preceding specifiers
- First specifier: signed and unsigned
- unsigned is always non-negative
- signed uses first bit to indicate negative number
- Java does not have specifier signed/unsigned
- Second specifier: short and long
- Determines memory size

# Combinations of Integer Specifiers

- Order not important; e.g., all of these are valid:

```
short int
unsigned short int
int
unsigned int
long int
unsigned long int
long unsigned int
signed long int
```

- We can omit ‘int’; e.g.:

```
long i = 1L; short s = 0;
```

# Range of integers

- Typically an ‘int’ corresponds to one machine word
- Example: size of int = 2 bytes (on an old CPU)
- unsigned int range:
  from 0 to $2^{16} - 1 = 65,535$
- signed int range:
  from $-2^{15} = -32,768$ to $2^{15} - 1 = 32,767$
- Explanation:
  - Numbers between 10000000 00000000 and 11111111 11111111 are used for negative numbers
- 2’s complement representation