Faculty of Computer Science, Dalhousie University *1-Oct-2018*

CSCI 2132 — Software Development

Lecture 12: C compared to Java: Expressions and Statements

Location: Chemistry 125 Instructor: Vlado Keselj
Time: 12:35 – 13:25

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

12 C Operators, Expressions, and Statements

The syntax for operators, expressions and statements for C is very similar to that of Java. Since we assume that you know Java quite well already, we will focus on differences between C and Java.

12.1 Arithmetic Operators

As in Java, the main arithmetic binary operators are `+`, `-`, `*`, `/`, and `%`. One difference between C and Java regarding arithmetic operators is that the operands of the `%` (modulus, or remainder) operator in C can only be integer types, while in Java, the operands can also be floating-point types (find out what this operator does when it takes two floating-point variables as operands in Java). If you try to apply the modulus operator with floating-point operands in C, the result will be a compile error.

Another operator where there can be a difference between C and Java is the division (`/`) operator. We can use this operator to explain the concept of implementation-defined behaviour frequently used in the C standard. We know when both operands of division are integers, the result of division will be an integer. When both operands are positive, the result is rounded down. However, if at least one of the operands is negative the result may not be clear. For example, should the result of $-10/7$ be -1 or -2?

In the earlier C standards, this was an example of implementatino-defined behaviour. The C standard deliberately leaves parts of the language unspecified. An 'implementation', which is a C compiler on a specific platform, fills in these details. This is usually done for efficiency reasons.

Let us look at the example of integer division (when the result is negative) again. In the C89 standard or earlier standards, this is an implementation-defined behavior. CPUs typically have an instruction for division, and when one operand is negative, some CPUs will round down the result, and some CPUs will round up the result. By making this implementation-defined, compilers on a specific platform can choose to be consistent with the corresponding CPU instruction, so that only one instruction is used when integer division is performed. This maximizes efficiency, but sacrifices portability (or at least makes it harder for a programmer).

The C99 standard, however, chooses to specify this. In C99, the result of integer division is always rounded toward 0. Therefore, the result of $-10/7$ is -1. Why does C99 choose to specify this? This is because by 1999, the integer division instruction in most CPUs chooses to round toward 0, except in some old architectures. Thus, the C99 standard decided to specify this to improve portability, knowing that it is very unlikely to decrease efficiency in modern CPUs.

By the way, the result of integer division in Java is always rounded toward 0.

Assignment Operators

The assignment operators, such as =, +=, and −=, and so on, are exactly the same as in Java.

Increment and Decrement Operators

The increment (++) and decrement (−−) operators work the same as those in Java. You are expected to know the difference pre- and post- increment and decrement operators, such as `++i` and `i++`.

Expression Evaluation

As in Java, there are also rules of precedence and associativity regarding expression evaluation, when multiple operators are used in one expression. These rules allow us to break any C expression into subexpressions, to determine uniquely where the parentheses would go if the expression were fully parenthesized. However, they do not always determine the value of expression, which may depend on the order in which its subexpressions are evaluated. Let us look at the following example (a, b, c are all `int` variables):

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

There are two subexpressions. If they are evaluated from left to right, then after these two statements, the value of c is 6. If they are evaluated from right to left, the value of c is 2.

In Java, subexpressions are evaluated from left to right. However, in C, the order of subexpression evaluation is unspecified; they are not specified by the C language standard. There is a difference between *unspecified behavior* and *implementation-defined behavior:* unspecified behavior is not required to be documented by the implementation, while implementation-defined behavior is.

Back to order of subexpression evaluation: when the order matters, we are supposed to use multiple expressions. The first example above can be rewritten as:

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

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`, ...)

The operators include the relational operators '<', '>', '<=', '>='; the equality operators '==', '`!=`', and the logical operators '`!`', '`&&`', and '`||`'.

The C standard C99 introduced the type `bool` that is used similarly to Java `boolean` as the result of logic operations. Before C99, the type for the result of logic operations was `int` and it still can be used since the use of `bool` is not mandatory. When using `int`, the false value of an expression is 0, while if the expression is true, the value is 1. The control and logical statements treat 0 as false, and anything other than 0 as true. This feature allows to have code such as the following code for calculating factorial of a number stored in `n`:

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

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) { ... }
  ```

This is the source of many errors for programmers who do not pay attention to this. For example, the expression `a < i < b` is valid in C (it is invalid in Java). However, it is not equivalent to `(a < i) && (i < b)`. Why? Suppose that `a = 5`, `i = 4`, and `b = 2`. Then `a < i < b` is equivalent to `(a < i) < b`. Since `a < i` is false, its value is 0. Thus this expression is evaluated as `0 < b`, which is 1 (true). However, `(a < i) && (i < b)` is 0.

Another common trap is to use = instead of ==. This type of error is usually caught by a compiler in Java, but not in C. For example, in

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

we probably want to compare if the value of `x` is equal to the value of `a + b`, but actually we are evaluating `a + b`, assigning the value to `x`, and the condition is true if this value is not zero.

"Short-circuit" evaluation applies to `&&` and `||`, as in Java. Consider the following example:

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

It is valid due to short-circuit evaluation of `&&`. If the short-circuit evaluation is not provided, it would be problematic code. Why?

C99 Standard: `_Bool`

- C99 allows optional boolean type (similar to C++); e.g.:

```
#include <stdbool.h>
_Bool flag;
flag = 5;       /* flag becomes actually 1 */
bool flag1;     /* same as _Bool, a macro */
flag  = true;   /* macro for 1 */
flag1 = false; /* macros for 0 */
```

There is no boolean type in C89 or earlier, since `int` is used as boolean—0 is interpeted as false, and anything else as true. This allows for expressions such as:

```
int i = 1, j = -1;
if ( i + j )
   print "not a zero\n";
```

In C99, there is a boolean type called `_Bool`. For example, you can define

```
_Bool flag;
```

This type is an integer type that can only be assigned 0 or 1. Thus if we write the following statement:

```
flag = 5;
```

the flag will actually have the value 1, and not 5.

In C99, the `<stdbool.h>` header defines the following three macros:

- `bool`, a macro for `_Bool`
- `true`, a macro for 1
- `false`, a macro for 0

Thus, if we use `#include <stdbool.h>` in our program, we can write code like

```
bool flag;
flag = true;
```

which looks exactly the same as the bool type in C++.

Control Structures

The syntax of the selection and iteration statements, including if, switch, while, do-while and for, is the same as the syntax of those in Java. The only difference is that the value of test expression is integer. Any non-zero value is treated as true, and 0 is treated as false. With this in mind, you can use these statements the same way you do in Java.

There are some statements that we can use to exit from a loop. One statement is the break statement, which is the same as the unlabeled break statement in Java. The other one is the continue statement, which is the same as the Java continue statement.

These two statements are also called jump statements, as they transfer control from one position of the program to another position.

An overview of 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

A jump statement that is not in Java is the goto statement. It can be used to jump to any statement in the same function, provided that the statement has a label. Unlike other jump statements, which provide some form of branching in a program, the goto statement provides unconditional jump to another statement. To use goto, we need to learn how to introduce a label and how to use the goto statement itself. To define a label for a C statement, we follow this syntax:
identifier`:` *statement*
Example: `loop: i++;`
In the above syntax, a label is an identifier placed at the beginning of a statement. With the label defined, we can follow the syntax below to write a goto statement:
`goto` *identifier*
Example: `goto loop;`

Let's see a simple example. In this example, we print the numbers from 1 to 10. We would normally use a for-loop or while-loop for this, but we can also use goto:

```
#include <stdio.h>

int main() {
  int i = 1;

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

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

The goto statement is explicitly excluded from Java, so that it is still a reserved word, but without use in Java, in order to prevent programmers to accidentally use it. The goto statement is quite old, and it has a very close

equivalent in the machine languages. It is present in the assembly languages, Basic, Fortran, etc. It used to be a quite popular command, but it was realized that its excessive use leads to code which is hard to read and maintain—the so-called "spaghetti" code. A new programming paradigm—the *structured programming*—emerged and the use of goto was strongly discouraged. This is the main reason that the goto statement was excluded from Java.

The goto statement in C can be used to jump from any point in a function to any other point. We can jump in or out of a loop, but we cannot jump to another function. For this reason, it is allowed to use the same label identifier in different functions. The following is an example showing a few different goto jumps:

```
#include <stdio.h>

int main() {
  int i=0;

  i = 100;
  goto L;
  while (i<10) {
  L:
    if (i>20) i = 5;
    printf("%d\n", i++);
  }
  f();
  return 0;
}

int f() {
  int i=20;

  i = 100;
  goto L;
  while (i<30) {
  L: L1:           /* we can have two labels */
    if (i>40) i = 25;
    printf("%d\n", i++);
  }
  return 0;
}
```

A question is whether we should use goto or not. On the one hand, goto makes the code hard to read and modify, as it can jump to anywhere in the same function. On the other hand, sometimes goto can be used to improve efficiency. The following is an example in which using goto improves efficiency:

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

loop_done: ...
```

Here we cannot replace goto with break. This is because this statement is inside switch, and if we use break here, it

will only break out of the switch, not the while loop. To avoid using goto, we thus have to use additional variables as flags, and add them into the test statement of while. This would be less efficient.

There are some other cases for which using goto is more efficient. You can also see goto statements frequently in machine-generated code, such as compilers that choose to generate C code for high-level languages. Thus goto is indeed useful.

For this course, in order to develop good programming habit, use goto only when you need jump out of nested loops (including switch).

null Statement

The null statement is simply a semicolon (`;`), which is the same as an empty statement in Java.

C programmers often use null statements, especially when the body of a loop is empty. For example, the following code snippet tests whether a number is a prime number:

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

Actually, a more efficient loop would be:

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

To make code readable, it is advisable to put the null statement on a line by itself. The same effect can be achieved with:

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

To summarize, in this and the previous lectures, we learned the difference between C and Java for operators, expressions and control structures. This way we can start writing C programs that use these features right away.

13 C Basic Types

Integer Types

We already learned that we can use keyword ‘int’ to declare variables. First, we will learn that we can add specifiers before int to declare new integer types.

The first set of specifiers are ‘signed’ and ‘unsigned’. An unsigned int is always non-negative, while in a variable of type signed int, 1 bit is used to indicate whether this variable is negative or not. Therefore, the maximum values that a signed int and an unsigned int can take are different. These two specifiers are not in Java.

The second set of specifiers are ‘short’ and ‘long’, and they are used to indicate the sizes (i.e., the number of bits required to store a variable of said type) of the integer type. Thus integer types come in different sizes. We will learn more about the sizes later.

With these specifiers, we can define different integer types. The following are six distinct combinations, from the smallest in size to the largest in size:

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

The order of specifiers does not matter. For example, the following two integer types are equivalent:

```
unsigned long int
long unsigned int
```

We can also drop the keyword int when a specifier is present. That is, we can abbreviate the names of integers types by dropping 'int'. This is popular but not required. For example, these two integers types are equivalent

```
unsigned long
unsigned long int
```

The two integer types below are also equivalent:

```
long
long int
```

This is different from Java. In Java, you have to use 'long' to declare a long integer and you cannot declare variables of 'long int'. In C, both are valid.

The range of values represented by each integer type varies between machines. For the type int, typically one machine word is used to store an int variable.

Let's take a 16-bit machine for example (this would be as old as an IBM 286). A typical compiler for such a machine would use 16 bits to represent an int value. The most significant bit will be used to indicate whether this value is negative or not. That is, this bit is 1 if and only if this value is negative. Thus, the following 2^{15} values can be used to store 0 and non-negative integers: 0000000000000000, 0000000000000001, 0000000000000010, ..., 0111111111111111. In this way, the maximum value of int is $2^{15} - 1 = 32,767$. Similarly, 2^{15} values can be used to store negative integers. Hence the minimum value would be $-2^{15} = -32,768$.

For the same machine, an unsigned int would also typically use one word. Since all the 16 bits can be used to represent the values, the maximum value is $2^{16} - 1 = 65,535$.

Why does not C require int variables to be stored in a certain fixed number of bits, regardless of the platform (Java does require an int to be 32-bit long)? The reason is that for C, efficiency is important. In the computer organization course, we will learn that a word is a fixed group of bits that are handled as a unit by the CPU instruction set. Thus, with the flexibility granted by the C standard, C compilers can choose to store an int variable in a machine word, regardless of the word size, to maximize efficiency.