Concepts of Programming Languages
Lecture Notes: Week 6—L-Values and Argument Passing

Stefan Mitsch

School of Computing, DePaul University
smitsch@depaul.edu

Midterm Exam (90min)

L-Values (20min)

The terms *l-value* and *r-value* refer to the location in an assignment. Considering a declaration **var** x: Int = 0, what is the meaning of x in x=x+1? First, we notice that x occurs in two different positions: left of the assignment operator (as an l-value—a storage *location*) and to the right of the assignment operator (as an r-value—a *read* from a storage location). On the right-hand side, x denotes a *value* that is read from a storage location, whereas on the left-hand side x denotes that storage location itself that gets modified. The essential feature of a location (a l-value) is that it has some content (an r-value).

In C, this distinction is quite obvious (and also not) because we can take the address of an l-value to obtain a pointer to that location, and we can dereference the pointer to modify the contents (the r-value) of the location, and we can take addresses of pointers.

```
1  int x = 5;
2  int y = 6;
3  int *p = &x;
4  int *q = &y;
5  int **r = &p;
6  r = &q;
7  q = p;
8  **r = 7;
```

Some language constructs are restricted to being r-values and cannot be evaluated as locations, even though their values might be stored temporarily during evaluation.

```
1  int *p = &(x + y); /* not allowed */
2  (x + y) = 7;       /* not allowed */
```

Some l-values may require r-mode evaluation when computing the locations. For example, array access, like arr[n+2] = 7, pointer arithmetic, field access obj.field = 7, and combinations of field and array accesses, may first perform some computation in order to determine a location.

Argument Passing (60)

So far, we defined methods in the form **def** f(x: String, y: Int) = x + y; in this form x and y are *formal parameters* (or just parameters) that, on function execution, hold *actual*

parameters (or *arguments*). In this section, we take a deeper look at alternative designs for passing arguments to functions.

Call-by-value. ⚙ As a starting example, let's determine what the C code below prints as value of x. Try to find out for yourself before you read on. □

```
1  void g (int y) {
2    y = y + 1;
3  }
4
5  void f () {
6    int x = 1;
7    g (x);
8    printf("%d\n", x);
9  }
```

The code above will print `x=1`, because C, like most programming languages, uses *call-by-value* by default, so that we pass a copy of the value of x to function g, which then locally modifies this value without modifying the outer location x. The steps in executing function $g(x)$ are

1. evaluate x to a value v
2. pass a copy of v (a new location!) to function g
3. the changes to that copy are not visible to the caller

It becomes more obvious when considering a call $g(x+1)$, where we pass the incremented value of $x+1$ to function g (this incremented value will be in a temporary anonymous location).

Call-by-reference. The complementary approach to call-by-value is *call-by-reference*, as for instance used in Perl. The steps for passing arguments by reference on execution of $g(x)$ are:

1. evaluate x to an l-value l
2. pass l to function g (g now has an *alias* of x)
3. the changes to the content of the location l are visible to the caller

```
1  sub g {
2      $_[0] = $_[0] + 1;
3  }
4
5  sub f {
6      my $x = 1;
7      g ($x);
8      print ("x = $x\n");
9  }
```

Simulating call-by-value and call-by-reference. In a call-by-value language we can simulate call-by-reference. For instance, in C we can use pointers to give functions access to shared memory locations.

```c
void g (int *p) { // pass pointer by value
  *p = *p + 1;    // modify pointed-to location
}

int main () {
  int x = 1;
  int *q = &x;
  g (q);
  printf ("x = %d\n", x); // prints 2
  return 0;
}
```

In a call-by-reference language that supports deep-copying of values we can simulate call-by-value. For example, in Perl it is good coding practice to create local copies of the function arguments before operating on their values.

```perl
sub g {
    my ($y) = @_; // create explicit copy
    $y = $y + 1;
}

sub f {
    my $x = 1;
    g ($x);
    print ("x = $x\n"); // prints 1
}
```

C++ adds reference types that keep call-by-reference explicit in the function declaration (like C with pointers), but implicit when calling functions.

```c++
#include <iostream>
 using namespace std;

void g (int& y) {
  y = y + 1;
}

int main () {
  int x = 1;
  g (x);
  cout << "x = " << x << endl;
  return 0;
}
```

Worksheets and Assignments (15min)

Assignment steps

1. Complete Worksheet X - Y and the accompanying quizzes on D2L
2. Complete the instructions in `todo.scala`

We use the remainder of the class to start working on the worksheets and assignments.