

CSC 347 - Concepts of Programming Languages

Argument Passing

Instructor: Stefan Mitsch



Learning Objectives

- ❓ How should arguments to functions be passed in?
 - Understand the difference between call-by-value and call-by-reference



Argument Passing

- Consider

```
1 def f (x:String, y:Int) = x * y  
2 f ("hello", 10)
```

- `x` , `y` : *formal parameters* (or *parameters*)
- `"hello"` , `10` : *actual parameters* (or *arguments*)



Argument Passing

- Subtleties in passing arguments
- What does a call to `f` print?

```
1 def g (y:Int) : Int =  
2   y = y + 1  
3  
4 def f () =  
5   var x = 1  
6   g (x)  
7   println (x)
```



Call-By-Value (CBV)

- Most PLs use *call-by-value* (CBV) by default
- To run `g (e)`
 - i. evaluate `e` to a value `v`
 - ii. pass a *copy* of `v` to `g`
 - iii. callee changes to copy of `v` not visible to caller



Example

```
1 def g (y:Int) : Int =  
2   y = y + 1  
3  
4 def f () =  
5   var x = 1  
6   g (x)  
7   println (x)
```

Call-By-Value

- Prints 1 in a CBV PL, i.e., `x=1` after call to `g`

Call-By-Reference

- Prints 2 in a CBR PL, i.e., `x=2` after call to `g`



Call-By-Reference

- Some PLs use *call-by-reference* (CBR)
- To run `g (e)`
 - i. evaluate `e` to an l-value `r` (address!)
 - ii. pass the l-value `r` to `g`
 - iii. callee changes via `r` are visible to caller



CBR and Temporaries

- Can temporary values be passed as l-values?
- `g(x+1)` is not obviously legitimate in CBR
- Some languages reject it, some allow it
- Perl allows it



Call-By-Reference: Perl

- Perl uses CBR

```
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 }  
10  
11 f ();
```

```
1 $ perl ./cbr.pl  
2 x = 2
```



Call-By-Reference: Perl

- Perl allows temporaries!

```
1 sub g {  
2     $_[0] = $_[0] + 1;  
3 }  
4  
5 sub f {  
6     my $x = 1;  
7     g ($x + 1);  
8     print ("x = $x\n");  
9 }  
10  
11 f ();
```

```
1 $ perl ./cbr.pl  
2 x = 1
```



Call-By-Reference: Perl

- Simulate CBV by creating copies explicitly

```
1 sub g {  
2     my ($y) = @_;  
3     $y = $y + 1;  
4 }  
5  
6 sub f {  
7     my $x = 1;  
8     g ($x);  
9     print ("x = $x\n");  
10 }  
11  
12 f ();  
13 # x=1
```



Call-By-Value: C

- Simulate CBR in C by explicitly passing, receiving, accessing a pointer

```
1 void g (int *p) {  
2     *p = *p + 1;  
3 }  
4  
5 int main () {  
6     int x = 1;  
7     g (&x);  
8     printf ("x = %d\n", x);  
9     return 0;  
10 }  
11 // x=2
```



Call-By-Reference: C++

- C++ has reference types `int&`
- Unlike `int*`, creates references (aliases) implicitly

```
1 #include <iostream>
2
3 using namespace std;
4
5 void g (int& y) {
6     y = y + 1;
7 }
8
9 int main () {
10     int x = 1;
11     g (x);
12     cout << "x = " << x << endl;
13     return 0;
14 }
15 // x=2
```



Call-By-Reference: C#

- C# has reference parameters `ref int`
- Unlike `int&` must also be used by caller

```
1 using System;
2 class Test {
3     static void g (ref int y) {
4         y = y + 1;
5     }
6
7     static void Main () {
8         int x = 1;
9         g (ref x);
10        Console.WriteLine("{0}", x);
11    }
12 }
13 // 2
```



Call-By-Reference: C++

- Passing a non-lvalue

```
1 #include <iostream>
2
3 using namespace std;
4
5 void g (int& y) {
6     y = y + 1;
7 }
8
9 int main () {
10     int x = 1;
11     g (x + 1);
12     cout << "x = " << x << endl;
13     return 0;
14 }
```

```
1 $ g++ -o reference reference.cpp
2 reference.cpp: In function 'int main()':
3 reference.cpp:11:8: error: invalid initialization of non-const
4   reference of type 'int&' from an rvalue of type 'int'
5     g (x + 1);
6         ^
7 reference.cpp:5:6: error: in passing argument 1 of 'void g(int&)'
8   void g (int& y) {
9         ^
```



Call-By-Value: Java

- Java has only a restricted form of *pointers*, called *references*
 - must point to heap-allocated objects
 - cannot point to stack-allocated data
 - cannot point to primitive types
- Java references cannot be forged
 - not from integers via casting
 - not from other references via pointer arithmetic
- Objects only accessed via references
 - unlike C++
 - Java has no address-of & operator



Call-By-Value: Java

- Simulate CBR In Java
- Heap-allocated object with field of intended argument type
- Pass a reference to the object instance **by value**

```
1 class IntRef { int n; }
2
3 public class Ref {
4     static void g (IntRef r) { r.n = r.n + 1; }
5
6     public static void main (String[] args) {
7         IntRef s = new IntRef (); s.n = 1;
8         g (s);
9         System.out.println (s.n);
10    }
11 }
12 // prints 2
```



Call-By-Value: Scala

```
1 def f (x: Double) : Double =  
2   val x1 = x  
3   val x2 = x  
4   x1 - x2  
5 println ("f= " + f (Math.random()))
```



Call-By-Value: Scala

- Scala allows functions as parameters
- A function that takes no arguments can be seen as a *delayed value*, also called a *thunk*

```
1 def g (x: () => Double) : Double =  
2   val x1 = x()  
3   val x2 = x()  
4   x1 - x2  
5  
6 println ("g= " + g (() => Math.random()))
```



Call-By-Name: Scala

- Scala has a special syntax for using thunks as parameters
- *Call-by-name* parameters are non-strict

```
1 def h (x: => Double) : Double =  
2   val x1 = x  
3   val x2 = x  
4   x1 - x2  
5  
6 println ("h= " + h (Math.random()))
```



Call-By-Name: Scala

- Call-by-name can be used to create new control constructs

```
1 def myWhile (cond: => Boolean) (body: => Unit) : Unit =  
2   if (cond)  
3     body  
4     myWhile (cond) (body)
```

- Use (brackets `{}` are required)

```
1 var i = 3  
2 myWhile (i > 0) {  
3   println ("i= " + i)  
4   i = i - 1  
5 }
```



Summary

- Call-by-value: pass copies of r-values as arguments
- Call-by-reference: pass l-values as arguments
- Call-by-name: special notation to pass parameterless functions as arguments