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

```
def f (x:String, y:Int) = x * y
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?

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

void f () {
  int x = 1;
  g (x);
  print (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

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

void f () {
  int x = 1;
  g (x);
  print (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

```
sub g {
    $_[0] = $_[0] + 1;
}

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

f ();
```

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

Call-By-Reference: Perl

- Perl allows temporaries!

```
sub g {
    $_[0] = $_[0] + 1;
}

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

f ();
```

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

Call-By-Reference: Perl

- Simulate CBV by creating copies explicitly

```
sub g {
    my ($y) = @_;
    $y = $y + 1;
}

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

f ();
# x=1
```

Call-By-Value: C

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

```c
void g (int *p) {
  *p = *p + 1;
}

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

Call-By-Reference: C++

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

```
#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;
}
// x=2
```

Call-By-Reference: C#

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

```
using System;
class Test {
  static void g (ref int y) {
    y = y + 1;
  }

  static void Main () {
    int x = 1;
    g (ref x);
    Console.WriteLine("{0}", x);
  }
}
// 2
```

Call-By-Reference: C++

- Passing a non-lvalue

```
#include <iostream>

using namespace std;

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

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

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

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**

```
class IntRef { int n; }

public class Ref {
  static void g (IntRef r) { r.n = r.n + 1; }

  public static void main (String[] args) {
    IntRef s = new IntRef (); s.n = 1;
    g (s);
    System.out.println (s.n);
  }
}
// prints 2
```

Call-By-Value: Scala

```
def f (x: Double) : Double =
  val x1 = x
  val x2 = x
  x1 - x2
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*

```
def g (x: () => Double) : Double =
  val x1 = x()
  val x2 = x()
  x1 - x2

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

```
def h (x: => Double) : Double =
  val x1 = x
  val x2 = x
  x1 - x2

println ("h= " + h (Math.random()))
```

Call-By-Name: Scala

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

```
def myWhile (cond: => Boolean) (body: => Unit) : Unit =
  if (cond)
    body
    myWhile (cond) (body)
```

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

- Brackets are required here

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