Control Abstraction and Parameter Passing

Read: Scott, Chapter 9.1-9.3

(lecture notes cover mostly 9.3)

Lecture Outline

- Control Abstraction
- Parameter Passing Mechanisms
  - Call by value
  - Call by reference
  - Call by value-result
  - Call by name
  - Call by sharing

Abstraction

- Abstraction: hiding unnecessary low-level detail
- Data abstraction: types
  - Type integer is an abstraction
  - Type struct Person is an abstraction
- Control abstraction: subroutines
  - A subroutine abstracts away an algorithm
  - A subroutine provides an interface: name, argument types, return type: e.g., `int binarySearch(int a[], int v)`
- Classes/objects in OO, Abstract Data Types (ADTs) are a higher level of abstraction

Subroutines

- Other terms: procedures and functions
- Modularize program structure

- Argument: information passed from the caller to the callee (also called actual parameter or actual argument)
- Parameter: local variable in the callee, whose value is received from the caller (also called formal parameter)

Parameter Passing Mechanisms

- How does the caller pass information to the callee?
- Call by value
  - C, Pascal, Ada, Algol68
- Call by reference
  - Fortran, C++, Pascal var params, sometimes Cobol
- Call by value-result (copy-in/copy-out)
  - Ada
- Call by name (outmoded)
  - Algol60
- Discussion applies to value model for variables

Parameter Passing Modes

- Most languages use a single parameter passing rule
  - E.g., Fortran, C
- Other languages allow different modes, in other words, programmer can choose different parameter passing rules in different contexts
  - E.g., C++ has two parameter passing mechanisms: swap(int &i, int &j) vs. swap(int i, int j)
  - Pascal too

Call by Value

- Value of argument is copied into parameter location

```
m,n : integer;
procedure R(k,j : integer)
begin
    k := k+1;
    j := j+2;
end R;
…
m := 5;
n := 3;
R(m,n);
write m,n;
```

By Value:

k	j
~~5~~	~~3~~
6	5

Output:

5 3

Call by Reference

- Argument is an l-value; l-value is passed to the parameter

```
   m,n : integer;
   procedure R(k,j : integer)
   begin
       k := k+1;
       j := j+2;
   end R;
…
   m := 5;
   n := 3;
   R(m,n);
   write m,n;
```

Value update happens in storage of caller, while callee is executing

k,m	j,n
~~5~~	~~3~~
6	5

Output:

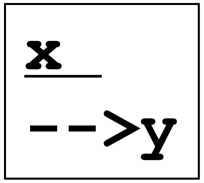
6 5

Call by Value vs. Call by Reference

- Call by value
  - Advantage: safe
  - Disadvantage: inefficient
- Call by reference
  - Advantage: more efficient
  - Disadvantage: may be unsafe due to aliasing
  - Aliasing (memory aliasing) occurs when two or more different names refer to the same memory location
    - E.g., `m` in `main`, and `k` in `R` are aliases for the same memory location during the call to `R`

Aliasing: Call by Reference

```
  y: integer;
  procedure P(x: integer)
  begin
      x := x + 1;
      x := x + y;
  end P;
…
  y := 2;
  P(y);
  write y;
```



x

-->y

During the call, **x** and **y** are two different names for the same location!

x,y

~~2~~

~~3~~

6

Output:

6

No Aliasing: Call by Value

```
    y: integer;
    procedure P(x: integer)
    begin
        x := x + 1;
        x := x + y;
    end P;
…
    y := 2;
    P(y);
    write y;
```

<u>x</u>

~~2~~

~~3~~

5

<u>y</u>

2

Output:
2

More Aliasing with Call by Reference

```
j,k,m : integer;
procedure Q(a,b : integer)
begin
   b := 3;
   a := m * a;
end Q;
...
s1: Q(m, k);
...
s2: Q(j, j);
```

Global-formal aliases:
`<m,a>` `<k,b>` associations during call to `Q` at `s1`

Formal-formal aliases:
`<a,b>` during call at `s2`

Questions

- Aliasing is an important concept in programming

- Can you think of other examples of aliasing?

- Why memory aliasing is considered dangerous?
- Can you think of other ways for creating memory aliasing?

Memory Aliasing is Dangerous

- One part of the program can modify a location through one alias, breaking invariants/expectations of other parts that use different aliases to the same location

- In general, we cannot know whether **`x->f`** and **`y->f`** are aliases to the same location
  - We "err" on the safe side
  - Aliasing makes reasoning about code hard
  - Aliasing prevents compiler optimization

Readonly Parameters

- What are some defenses against unwanted modification through aliases?
  - const parameters are an important paradigm in C/C++

```
log(const huge_struct &r) { … }
…
log(my_huge_struct);
```

Readonly Parameters

- const can be tricky…

```
log(const huge_struct * r) {
   r->f = 0; // NOT OK
}
```

vs.

```
log(huge_struct * const r) {
   r->f = 0; // OK
}
```

Readonly Parameters

```
class C {
 int f;
public:
   int get() const
     { return f; }
   int set(int g)
     { f = g; }
};
```

More on Call by Reference

- What happens when someone uses an expression argument for a call-by-reference parameter?
  - **`(2*x)`**?

Lecture Outline

- Control Abstraction
- Parameter Passing Mechanisms
  - Call by value
  - Call by reference
  - Call by value-result
  - Call by name
  - Call by sharing

Call by Value-Result

- Argument is copied in into the parameter at entry, parameter is copied out into the argument at exit

```
 m,n : integer;
 procedure R(k,j : integer)
 begin
     k := k+1;
     j := j+2;
 end R;
…
 m := 5;
 n := 3;
 R(m,n);
 write m,n;
```

By Value-Result

k	j
~~5~~	~~3~~
6	5

Output:

```
6    5
```

Call by Value-Result

```
  c : array [1..10] of integer;
  m,n : integer;
  procedure R(k,j : integer)
  begin
      k := k+1;
      j := j+2;
  end R;

/* set c[i] = i */
  m := 2;
  R(m, c[m]);
  write c[1], c[2], …, c[10];
```

k	j
~~2~~	~~2~~
3	4

What element of **c** has its value changed? **c[2]**? **c[3]**?

Call by Value-Result

```
  …
/* set c[i] = i */
  m := 2;
  R(m, c[m]);
  write c[1], c[2], …, c[10];
```

k	j
~~2~~	~~2~~
3	4

What element of **c** has its value changed? **c[2]**? **c[3]**?

One possible implementation is to copy arguments from left to right and re-evaluate the l-value at exit. This will produce **m=3** and **c[3]=4.**

Exercise

- Write a program that produces different result when the parameter passing mechanism is call by value, call by reference, or call by value-result

Exercise

```
   y: integer;
   procedure P(x: integer)
   begin
        x := x + 1;
        x := x + y;
   end P;
…
   y := 2;
   P(y);
   write y;
```

By Value Output:

2

By Reference Output:

6

By Value-Result Output:

5

Call by Name

- An expression argument is not evaluated at call.

It is evaluated within the callee, if needed.

```
  c : array [1..10] of integer;
  m : integer;
  procedure R(k,j : integer)
  begin
      k := k+1;
      j := j+2;
  end R;
/* set c[i] to i */
  m := 2;
  R(m, c[m]);
  write m,c[m]
```

```
m := m + 1
```

```
c[m] := c[m] + 2
```

m	c[ ]
~~2~~	1 2 ~~3~~ 4 5 6 7 8 9 10
3	1 2 5 4 5 6 7 8 9 10

Call by Name

- Call by name (Algol 60)
  - Case1: Argument is a variable
    - Same as call by reference
  - Case2: Argument is an expression
    - E.g., expressions **`c[m], f(x,y), x+z`**, etc.
    - Evaluation of the argument is deferred until needed
    - Argument is evaluated in the caller's environment – the expression goes with a THUNK (a closure!) which carries the necessary environment
    - Generally inefficient
    - Difficult to implement

Call by Name vs. Call by Value

- Recall reduction strategies in the λ-calculus
  - What reduction strategy corresponds to call by name?
    - Normal order reduction
  - What reduction strategy corresponds to call by value?
    - Applicative order reduction

Lecture Outline

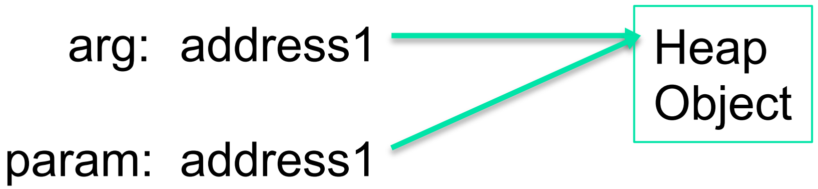
- Control Abstraction
- Parameter Passing Mechanisms
  - Call by value
  - Call by reference
  - Call by value-result
  - Call by name
  - Call by sharing

Reference Model for Variables

- So far, discussion applied to the value model for variables
- What is the parameter passing mechanism in languages that use the reference model for variables? Neither call by value, nor call by reference make sense for languages with the reference model
  - Call by sharing: argument reference (address) is copied into parameter. Argument and parameter references refer to the same object

Reference Model for Variables

- How does call by sharing relate to call by value?
  - Similarities?
  - Differences?

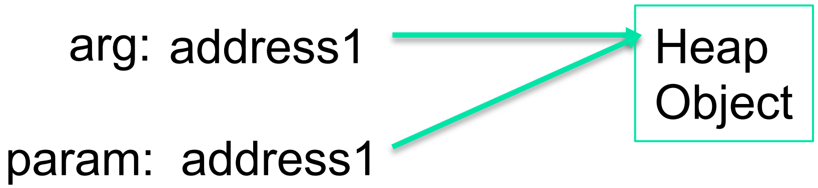


arg: address1

param: address1

Heap Object

- How does call by sharing relate to call by reference?
  - Similarities?
  - Differences?



arg: address1

param: address1

Heap Object

Immutability

- Immutability is a "defense" against unwanted mutation due to sharing
- In Scheme, methods are pure
- In Python, there are immutable datatypes
- In Java, not much… There is no const-like construct to protect the referenced object
  - final disallows re-assignment of a variable

```
final Point p = new Point();
p = q; // NOT OK
p.x = 0; r.y = 0; // ALL OK
```

Immutability

- Software engineering principles that help protect against unwanted mutation due to “sharing”
  - Avoid representation exposure (rep exposure)
  - Design immutable ADTs
  - Write specifications that emphasize immutable parameters
    - E.g., modifies: none

Exercise

- Construct a program which prints different result when parameter passing mechanism is
  - Call by value
  - Call by reference
  - Call by value-result
  - Call by name

The End