Subprograms: Parameters

Programming Languages
William Killian
Millersville University
• Terminology
  • Signature
  • Formal vs. Actual
  • Parameter Correspondence

• Passing Modes
  • Semantic
  • Actual

• Passing Arrays
Terminology

Signature

*Portion of the Subprogram that tells us its interface but not its implementation.*

*Contains:*

1. Return type (if any)
2. Name
3. Count, order, and type of all Parameters

*The type signature omits the name*
Parameters

Within a definition:

```c
int magic(int a, int b, int c) {
    return b * b - 4 * a * c;
}
```

Within a call:

```c
int d = magic(a, a + 1, a * 4);
```
Parameters

Within a definition:

Called a **formal parameter**
- A “dummy” variable listed in the signature
- Used within the function’s implementation

Within a call:

Called an **actual parameter**
- Represents a value or address
- Determined at the point of execution
Parameter Correspondence

Actual Parameters must map to Formal Parameters

How do you think we should be able to map these?
Parameter Correspondence

**Actual Parameters** must map to **Formal Parameters**

*Positional*
- The first actual parameter corresponds to the first formal parameter, the second actual parameter corresponds to the second formal parameter, and so on...
- Very easy to check/implement

*Keyword*
- The *name* associated with the formal parameter must be used at the call site (where a subprogram is called)
- Parameters can appear in any order
- Intent can be clearer, but it harder to check/implement
Passing Modes
Semantic Modes

• In mode
  *Information is sent to the subprogram*

• Out mode
  *Information is retrieved from the subprogram*

• In-Out mode
  *Information is sent/received to/from the subprogram*
Semantic Modes

**Caller**
(sub (a, b, c))

In mode

**Callee**
(procedure sub (x, y, z))

Out mode

Inout mode

Call

Return

a

b

c

x

y

z
Actual Modes

- Pass-by-Value
- Pass-by-Result
- Pass-by-Value-Result
- Pass-by-Reference
- Pass-by-Name
Pass-by-Value (In mode)

• The **value** of the actual parameter is used to initialize its corresponding formal parameter
• The underlying value is usually copied from one memory cell to another

• **Advantages:**
  • Simple

• **Disadvantages:**
  • Additional storage required
  • Copied value can be large (in space)
Pass-by-Value

```c
int by_value (int x) {
    int y = 4;
    x += 4;
    return y + x;
}

int y = 3;
int z = by_value (y);
// y =
// z =
```
Pass-by-Value

Most languages do pass-by-value default:
• C
• C++
• Java
• C#
• Javascript
• Python* (for immutable types)
Pass-by-Result (Out mode)

• There is no passed initial value, but the formal parameter acts like a local variable.
• When control flow returns to the caller, copy the result back to a memory cell.

• **Advantages:**
  • Useful for when you have multiple “outputs”

• **Disadvantages:**
  • For \( \text{fun}(x, \text{list}[x]) \) should the address of \( \text{list}[x] \) be determined before or after execution?
  • Special semantics need to be determined for calls that use the same memory cell twice: \( \text{fun}(a, a) \)
Pass-by-Result

```c
void by_result (int x) {
    int y = 2;
    x = y + 4;
}

int y = 3;
by_result (y);
// y =
```
Pass-by-Value-Result (In-Out mode)

• The expected combination of pass-by-value and pass-by-result

• Also called pass-by-copy (copy ALL THE THINGS)

• Formal parameters all have local storage
Pass-by-Value-Result

Very few languages can pass-by-value-result:
• FORTRAN
Pass-by-Value-Result

```c
void by_value_result (int x) {
    int y = 4;
    x = y + 4;
}
```

```c
int y = 3;
by_value_result (y);
// y =
```
Pass-by-Reference (In-Out mode)

• Instead of passing a value, pass an access path (or memory address).
• Also called pass-by-sharing
• Advantages:
  • Lower memory footprint
• Disadvantages:
  • Slower access (must deference to retrieve value)
  • Potential side effects (multiple reference updates)
  • Aliasing isn’t a lot of fun
Pass-by-Reference

```c
 void by_reference (int x) {
     int y = 4;
     x = y + 4;
 }

int y = 3;
by_reference (y);
// y =
```
Pass-by-Reference

Most languages can pass-by-reference:

- C++
- Java (reference semantics for objects)
- C# (ref keyword)
- Python* (for mutable types)
Pass-by-Name (In-Out mode)

• Literal Text Substitution
• Formals bound at time of call
• Values bound at time of reference or assignment

• **Advantages:**
  • Provides the latest possible binding
  • Extremely flexible

• **Disadvantages:**
  • No obvious semantics by looking at it

*It was possible to do this natively in ALGOL; harder to do today due to language design decisions.*
Pass-by-Name (Jensen’s Device)

```c
#define SUM(Type, Var, Low, High, Term) ( {
  int low = (Low); \
  int high = (High); \
  Type sum = (Type)0; \
  for (Var = low; Var <= high; ++Var) { \
    sum += (Term); \
  } \
  sum;
})
```
Pass-by-Name (Jensen’s Device)

double sum = SUM(double, i, 1, 100, 1.0 / i)

// expands to:
double sum = ({
    int low = (1);
    int high = (100);
    double sum = (double)0;
    for (i = low; i <= high; ++i) {
        sum += (1.0 / i);
    }
    sum;
});
Pass-by-Name (Jensen’s Device)

double prod = SUM(double, i, 1, 100, SUM(double, j, i, 100, i * j))

// expands to
double prod = ({
    int low = (1);
    int high = (100);
    double sum = (double)0;
    for (i = low; i <= high; ++i) {
        sum += (({
            int low = (i);
            int high = (100);
            double sum = (double)0;
            for (j = low; j <= high; ++j) {
                sum += (i * j);
            }
            sum;
        }));
    }
    sum;
});
Passing Arrays as Parameters
<table>
<thead>
<tr>
<th>Languages Where Passing Arrays is Fine and Works as Expected</th>
</tr>
</thead>
<tbody>
<tr>
<td>C#</td>
</tr>
<tr>
<td>JavaScript</td>
</tr>
<tr>
<td>Lisp</td>
</tr>
</tbody>
</table>
Languages Where Passing Arrays is Weird

C

C++
Array Passing in C/C++

Definition of a stack-based array in C:

```c
int arr[5];
```

The type of `arr` is `int[5]`

Passing `arr` as a parameter will *decay* its type.
Decay means automatic conversion (loss of information) `int*`
Array Passing in C/C++

Definition of a heap-based array in C++:

```cpp
int* arr = new int[5];
```

The type of `arr` is `int*`

Passing `arr` as a parameter doesn’t change its type:

```cpp
int*
```
Array Passing in C/C++

We don’t want heap-based and stack-based arrays to interact differently, so they need to be converted to a common type

• Though it is possible to accept only stack-based arrays!
Multi-Dimensional Array Passing in C++

double m[3][4];

// valid
void mat_inverse (double (&m)[3][4])
void mat_inverse (double* m[4])

// invalid
void mat_inverse (double** m)

We decay the outer-most dimension only