

Subroutines and Control Abstraction

---

Textbook, Chapter 8

1

---

---

---

---

---

---

---

Subroutines and Control Abstraction

---

- Mechanisms for process abstraction
- Single entry (except FORTRAN, PL/I)
- Caller is suspended
- Control returns to caller

2

---

---

---

---

---

---

---

Design Issues

---

- Syntax, type checking
- Parameter passing mechanism
- Static or dynamic allocation of locals
- Static or dynamic scope
- Environment of parameter functions
- Overloading
- Generics
- Separate compilation

3

---

---

---

---

---

---

---

## Syntax

- **FORTRAN**

```
SUBROUTINE ADDER (parameters)
```

- **Ada**

```
procedure ADDER (parameters)
```

- **C**

```
void ADDER (parameters)
```

4

---

---

---

---

---

---

---

## More Syntax

- **Positional or keyword parameters**

- **Default values**

- **Ada example:**

```
function F (X : FLOAT;  
           Y : INTEGER := 1;  
           Z : FLOAT) RETURN FLOAT;
```

```
I := F (3.14, Z => 2.71);
```

5

---

---

---

---

---

---

---

## Type Checking

- **Are argument types checked against parameter types?**

- **If functions are passed as arguments, is the function type checked?**

- **Kernighan & Richie C:**

```
int foo (int (*f)()) { ... }  
int bar (int i, int j) { ... }  
foo (bar);
```

6

---

---

---

---

---

---

---

## Parameter-Passing Methods

- Call-by-Value (Copy-In, Eager Eval.)
- Call-by-Result (Copy-Out)
- Copy-In-Copy-Out
- Call-by-Reference
- Call-by-Name
- Call-by-Need (Lazy Evaluation)

7

---

---

---

---

---

---

---

---

## Call-by-Value

- Allocate memory for parameter
- Evaluate argument
- Initialize parameter with value
- Call procedure
- Nothing happens on return
  
- Used in C, C++, Pascal, Lisp, ML, etc.

8

---

---

---

---

---

---

---

---

## Call-by-Value Example

```
int a = 1;
void foo (int x) {
    // a and x have same value,
    // changes to a or x don't
    // affect other variable
}

// argument can be expression
foo (a + a);
// no modifications to a
```

9

---

---

---

---

---

---

---

---

## Call-by-Result

- Argument must be variable
- Allocate memory for parameter
- Don't initialize parameter
- Call Procedure
- ...
- Copy parameter into argument var.
- Return from procedure

10

---

---

---

---

---

---

---

---

## Call-by-Result Example

```
int a = 2;
void foo (int x) {
    // x is not initialized,
    // changes to a or x don't
    // affect other variable
}
// argument must be variable
foo (a);
// a was modified
```

11

---

---

---

---

---

---

---

---

## Call-by-Value-Result (Copy-In-Copy-Out)

- Combination of previous two
- Copy argument value on call
- Copy result on return
  
- Used by Ada for parameters of primitive type in in-out mode

12

---

---

---

---

---

---

---

---

## Call-by-Value-Result Example

```
int a = 3;
void foo (int x) {
    // a and x have same value
    // changes to a or x don't
    // affect each other
}
// argument must be variable
foo (a);
// a might be modified
```

13

---

---

---

---

---

---

---

---

## Call-by-Reference

- Evaluate argument into temporary
- Parameter is alias for location of tmp
- Call procedure
- Nothing happens on return
  
- Used by FORTRAN before 1977,  
Pascal var parameter

14

---

---

---

---

---

---

---

---

## Call-by-Reference Example

```
int a = 4;
void foo (int x) {
    // a and x reference same location
    // changes to a and x
    // affect each other
}
// argument can be an expression
foo (a);
// a might be modified
```

15

---

---

---

---

---

---

---

---

## Call-by-Ref in FORTRAN IV Implementations

```
SUBROUTINE FOO (I)
  I = 5
  RETURN

  FOO (10)
  J = 10
```

**J gets value 5!**

16

---

---

---

---

---

---

---

---

## Call-by-Name

- Don't evaluate argument
- Create closure to evaluate argument
- Call procedure
- Eval arg by calling parameter closure
- Nothing happens on return
  
- Used by ALGOL-60, Simula-67
- Similar to macro expansion (e.g, TeX)

17

---

---

---

---

---

---

---

---

## Call-by-Name Example

```
int a = 5;
void foo (int x) {
  // x is a function
  // to get value of argument,
  // evaluate x() when value is needed
}
// argument can be an expression
foo (a + a);
// no modifications to a
```

18

---

---

---

---

---

---

---

---

## Call-by-Need (Lazy Evaluation)

- Similar to Call-by-Name
- Argument evaluated only once
- Result kept in temporary
- Behavior differs with side effects
  
- Used by Haskell, Miranda

19

---

---

---

---

---

---

---

---

## Example Lazy Evaluation

```
int diverge () {  
    return diverge();  
}  
  
int if_then_else (int c, int t, int e) {  
    return c ? t : e;  
}  
  
int i = if_then_else(1, 42, diverge());
```

20

---

---

---

---

---

---

---

---

## Static vs. Dynamic Locals

- **Static**
  - efficient
  - no time for (de)allocation required
  - retain value across function calls
- **Dynamic (on the stack)**
  - required for recursive functions
  - on modern hardware cheap access
  - let compiler do the optimizations

21

---

---

---

---

---

---

---

---

## Implementation of Parameter Passing

- Allocate activation record in registers or on the stack
- Copy values into activation record
- Copy pointers for Call-by-Reference
- Encapsulate lazy args. in closure
- Branch to function code

22

---

---

---

---

---

---

---

---

## Environment of Function Parameters

- Function is passed down

```
int x = 1; // static scoping
int foo () { return x; }
int bar (int (*f) ()) {
    int x = 2; // dynamic scoping
    return f();
}
int i = bar (foo);
```

23

---

---

---

---

---

---

---

---

## Static vs. Dynamic Scope

- Static scope
  - non-local variables are looked up in statically enclosing scope
  - Need static link for nested functions
  - i gets value 1
- Dynamic scope
  - non-locals are looked up along dynamic call chain
  - i gets value 2

24

---

---

---

---

---

---

---

---

## Environment of Function Parameters

- **Function is passed up**

```
(define (add n)
  (lambda (m) (+ n m)))
```

```
(define add1 (add 1))
(define add5 (add 5))
```

```
(define i (add1 10))
(define j (add5 10))
```

25

---

---

---

---

---

---

---

---

## Implementation of Closures

- **Functions are represented as pair**

- pointer to code
- pointer to environment

- **Activation records might be on heap**

- **Needs garbage collection**

- E.g., we don't know when to reclaim the activation record of add

26

---

---

---

---

---

---

---

---

## Garbage Collection

- **No explicit free or delete**

- **When memory runs out, GC**

- traverses life data structures starting with global variables and stack
- marks reachable data
- scans heap to find unreachable data
- collects unreachable data in free list
- may compact heap

27

---

---

---

---

---

---

---

---

## Overloading vs. Multimethods

- Multiple methods/functions with same name, different arg. types

```
int foo (int);  
int foo (float);
```

- Overloading
  - method selection at compile time
- Multimethods
  - method selection at run time

28

---

---

---

---

---

---

---

---

## Overloaded Operators

- Allow redefining operators

- Ada

- function “\*” . . .

- C++

- operator \*
- allows redefinition of [], -, (), etc.

29

---

---

---

---

---

---

---

---

## Generic Functions

- Functions with type parameter

```
template <class Type>  
Type max (Type n, Type m) {  
    return n > m ? n : m;  
}
```

- Instantiated at compile time
- Better than using preprocessor

30

---

---

---

---

---

---

---

---

## Separate Compilation

- **Compiler needs type information of other compilation units**
- **C++: use #include**
- **Java: search in current directory**

31

---

---

---

---

---

---

---

---