CS 461

Programming Language Concepts

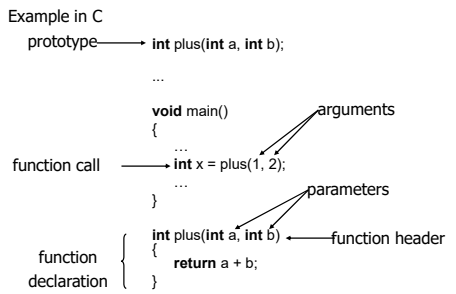
Gang Tan
Computer Science and Engineering
Penn State University

1

Functions and Procedures

2

Terminology



3

Parameters vs. Arguments

Parameters (AKA formal parameters, formal arguments): names in the declaration of a function header

Arguments (AKA actual parameters, actual arguments): variables/expressions passed to a function during a function call

4

Parameter-Argument Matching

- Usually by number and by position
  - Suppose f has two parameters, then any call to f must have two arguments, and they must match the corresponding parameters' types.
- Exceptions
  - Python/Ada/OCaml/C++
    - arguments can have default values
    - Python example:

```
>>> def myfun(b, c=3, d="hello"):
        return b + c
>>> myfun(5,3,"hello")
>>> myfun(5,3)
>>> myfun(5)
```


5

Parameter-Argument Matching

- Exceptions:
  - Arguments and parameters can be linked by name
  - Python example:

```
>>> def myfun(a, b, c):
        return a-b
>>> myfun(2, 1, 43)
1
>>> myfun(c=43, b=1, a=2)
1
>>> myfun(2, c=43, b=1)
1
```


6

Parameter Passing

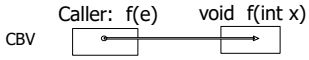
- How values are passed between arguments and parameters?
- Different parameter passing mechanisms
  - Call by value (CBV, AKA pass by value)
  - Call by result (CBR)
  - Call by value-result (CBVR)
  - Call by reference (CBR)
  - Call by name (CBN)

7

7

Call By Value

- Compute the *value* of the argument at the time of the call and copy that value to storage for the corresponding parameter
  - Copy-in semantics
    - At start of call, argument's value is computed and copied into parameter's storage



  - Example: void f(int x) {…}; f(3+5)

8

8

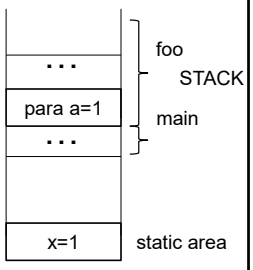
Call By Value: Storage Allocation

1. At start of the call, arguments are evaluated to their values
2. Storage allocated for parameters on AR
3. Argument values copied to storage for parameters AR
4. AR destroyed when callee returns

9

Call By Value

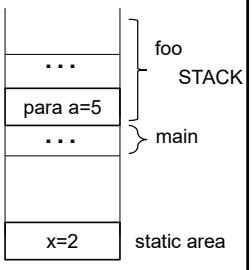
```
int x=1;
int foo (int a) {
    x = 2;
    a = 5;
    return x+a;
}
void main() {
    foo(x); //result?
    print(x); //result?
}
```



10

Call By Value

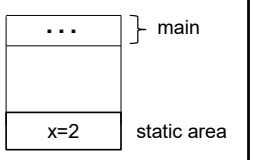
```
int x=1;
int foo (int a) {
    x = 2;
    a = 5;
    return x+a;
}
void main() {
    foo(x); //result?
    print(x); //result?
}
```



11

Call By Value

```
int x=1;
int foo (int a) {
    x = 2;
    a = 5;
    return x+a;
}
void main() {
    foo(x); //result?
    print(x); //result?
}
```



12

Call By Value

- Arguments and parameters have separate storage
  - Their values may diverge
  - Call by value doesn't allow the callee to modify an argument's value.
- Does the following C program swap the values of arguments?

```
void swap (int a, int b) {
  int temp=a;
  a = b;
  b = temp;
}
… x =1; y = 2; swap(x,y); …
```

- All arguments in C and Java are passed by value.
  - But pointers can be passed to allow argument values to be modified.
  - E.g., void swap(int *a, int *b) { … }

13

13

Call By Result

- Copying the final value computed for the parameter out to the argument at the end of the call
  - No copying of the initial value
- Copy-out mechanism
  - before returning control to caller, final value for the parameter is copied to the argument

Caller: f(e) void f(int x)

14

14

Call By Result

- Notes:
  - The argument must have an address
  - The parameter is initialized by the callee
    - The callee doesn't care about the initial value of the argument
    - Used as a mechanism for the callee to return a value

15

15

Call by Value-Result

- Two steps:
  1. Copying the argument's value into the parameter at the beginning of the call
  2. Copying the computed result back to the corresponding argument at the end of the call
- Copy-in and copy-out
  - at start of call, argument's value is computed and is copied into parameter's storage
  - before returning control to caller, final value for the parameter is copied to the argument

Caller: f(e) void f(int x)

16

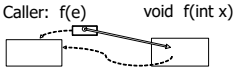
16

Call By Reference

- Compute the *address* of the argument at the time of the call and assign it to the parameter
  - During function execution, the argument and the parameter are alias
  - Argument must have an address (l-value in C/C++)
- C++ example

```
int h=10;
void B(int &w) {
     int i;
     i = 2*w;
     w = w+1
}
… B(h) …
```

Caller: f(e) void f(int x)



17

17

Discussion

- In the absence of aliasing, call by value-result is equivalent to call by reference
  - Def of alias: different names refer to the same storage location

18

18

Call By Name

- Textually substitute the argument for each occurrence of the parameter in the function's body
  - without computing the value of the argument first
  - can be viewed as macro expansion
  - originally used in Algol 60
- C macros
  - #define SQUARE(x) ((x) * (x))

19

19

Call By Name

- However, error prone
  - #define SQUARE(x) (x * x)
    - What is SQUARE(2+3)?
  - *Example: Jensen's device*
    - *one macro could compute (depending on arguments)*
      - *product of two numbers*
      - *sum of an array*
      - *dot product of two arrays*
- Not many languages use it
  - Examples: Algol 60
  - Haskell uses call by need, a variant of call by name

20

20

Considerations when choosing parameter-passing mechanisms

- minimize access to data
  - use pass-by-value if no data need be returned
  - use pass-by-result if no data need be sent to callee
  - Call by reference or value-result otherwise
- only use pass-by-reference when needed
  - can accidentally change the value of the parameter
- large arrays/objects usually pass-by-reference
  - avoids copying the entire array

21

21

Parameter passing in major languages: C

- Essentially pass-by-value
- Pass-by-reference
  - simulated using pointer values
  - pointer notation
    - int *p
      - declares p to be a pointer to an int
    - &x
      - provides the address of variable x
    - *p
      - dereferences a pointer
      - get the value of the variable pointed at by p

22

22

Parameter passing in major languages: C++

- same as C, but
- also has Call by reference
  - these are like pointers, but implicitly dereferenced
  - true pass-by-reference
  - void swap(int& a, int& b) {
    int temp = a;
    a = b;
    b = temp;
    }
    swap(x,y);
  - note, int &a, int &b can also be used

23

23

Parameter passing in major languages: Java

- Pass-by-value
- primitive data types: values are copied
- object and array parameters
  - Pass references values
  - still Call by value, but it's the reference values being passed

24

24

Example why Java is Call by value

```
Dog aDog = new Dog("Max");
foo(aDog);
aDog.getName().equals("Fifi"); // false

public void foo(Dog d) {
  d.getName().equals("Max"); // true
  d=new Dog ("Fifi");
}
```

- If Java used Call by reference, then after the call the test would be true
- Java: no real way to accomplish swap() with two **int** parameters
  - Workaround: pass in an array with two elements

25

25

Parameter passing in major languages: Ada

- Does not specify parameter passing implementations
- Can specify each parameter as:
  - **in**: can be read but not modified
    - implementation can be Call by value or Call by reference depending on size of parameter
  - **out**: cannot be read until value set by the callee function
    - initial value is never used
    - argument get the final value
    - implementation can be Call by result or Call by reference depending on the size of parameter
  - **in out**: can be read and modified
    - implementation either Call by value-result or Call by reference

26

26

Ada Example

```
procedure A_Test (A, B: in Integer; C: out
Integer) is
begin
   C := A + B;
end A_Test;
```

27

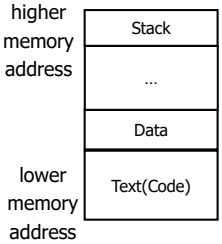
27

Function calls and returns

28

28

Process Memory Region



- Text: static code
- Stack: program execution stacks
  - Support function calls and returns
- Data
  - initialized global and static variables
  - storage for uninitialized variables (BSS)
  - Heap: dynamically allocated data (malloc, new)

29

29

Activation records

- Required storage
  - parameters
  - local variables
  - return address
  - functional return value (for functions only)
  - status info about caller (save registers)
- Activation record
  - block of information associated with a function activation
    - including its parameters and local variables
    - data that can change when function is executed
  - is only relevant while a function is active

30

30

Stack of activation records

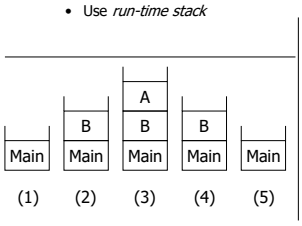
- Activation records are created dynamically
  - pushed onto the stack in called order
- For each call
  - create an activation record for the callee
    - Push it to the stack
    - Store information such as local variables and parameters
  - When the call returns
    - Destroy the corresponding activation record
- Every function that is active has an activation record on the stack
- Need an activation record for each call!
  - In the case of recursion, each call correspond to a separate activation record

31

31

Run-time Stack

- To support recursion, need a new activation record for each time a function is called.
  - Dynamic allocation of parameters and local variables
  - Use *run-time stack*




```
procedure Main is
begin
(1)   B(6);
(5) end;
procedure A(X:Integer) is
begin
(3)   ...
end;
procedure B(Y:Integer) is
begin
(2)   A(Y);
(4) end;
```

32

Stack and frame pointers

- sp (stack pointer): top of the stack
- fp (frame pointer): bottom of the top activation record
  - used to destroy the current activation record upon return from the function
  - an activation record may not have a statically known size

33

33

Activation records components

- return address
  - address of instruction following the call
  - often pointer to code segment of caller and offset address
- parameters
- local variables
  - only dynamically allocated ones
  - statically allocated variables are stored elsewhere
    - often with code
- return value
  - the value (if any) returned by the function

34

34

Activation records components

- dynamic link
  - saved frame pointer
  - points to the bottom of the caller's activation record
- static link
  - For statically scoped languages
  - Pointer to the frame of the lexically surrounding function
  - For finding visible non-local variables
- saved registers

35

35

Dynamic and Static Links Example

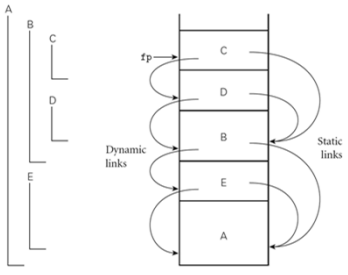


Figure 9.1 Example of subroutine nesting, taken from Figure 3.5. Within B, C, and D, all five routines are visible. Within A and E, routines A, B, and E are visible, but C and D are not. Given the calling sequence A, E, B, D, C, in that order, frames will be allocated on the stack as shown at right, with the indicated static and dynamic links.

36

36

Caller and Callee-Saves Registers

- Processor registers are storage that quickly accessible to CPU
  - eax, ebx, …
- Caller-saves registers
  - Registers saved by the caller if they store information that is needed across the function call
  - Callee can assume values in such registers can be destroyed
- Callee-saves registers
  - Callee has to save/restore their values if they are used in the callee
  - Caller can assume values in such registers are preserved across the function call

37

37

Activation records support recursive functions

- *Why do recursive functions require that local variables be dynamically allocated?*
  - we do not know until run-time how deep the recursion will go, thus we cannot know how many copies we will need
  - if the language is not recursive, activation records can be statically allocated
    - but this may waste memory, because some functions may never be called

38

38