CS49K Programming Languages

Chapter 9: Subroutines and Control Abstraction

LECTURE 11: SUBROUTINES

DR. ERIC CHOU

IEEE SENIOR MEMBER

Objectives

Subroutines and Control Abstraction

SECTION 1

Functional Abstraction

- A function performs a specified task, given stated preconditions and postconditions
- It has a name, parameters and a return value
- It may be used “by name” as long as …
  - appropriate values or objects are passed to it as parameters,
  - its preconditions are met
  - its return value is used in an appropriate context
- In this sense we have “abstracted” the function and “hidden” its implementation details

A Set of Rules

- Code Section
- Subroutines
- Procedure
- Operation
- Function
- Method
- Lambda Expression

Advantages

- **Modularization:** Decomposing a complex programming task into simpler steps
- **Code Reuse:** Reducing duplicate code within a program
- **Packaging:** Enabling reuse of code across multiple programs
- **Team Work:** Dividing a large programming task among various programmers, or various stages of a project
- **Abstraction:** Hiding implementation details
- **Readability:** Improving traceability

Functionality

Visibility **Generic** **Parameter**

```
public static<T> max(T variableA, T variableB)
{
    /* code of the function */
    …

}
```

Scope **Method Signature**

Note: Scope cannot be specified only by the static keyboard. In here, we only indicate the scope of the method: whether it is a class method or instance method. For the scope of variables, there are other ways to identify it.

Topics in This Chapter

1. Call Stack
2. Calling Sequence
3. Parameter Passing
4. Exception Handling
5. Coroutines

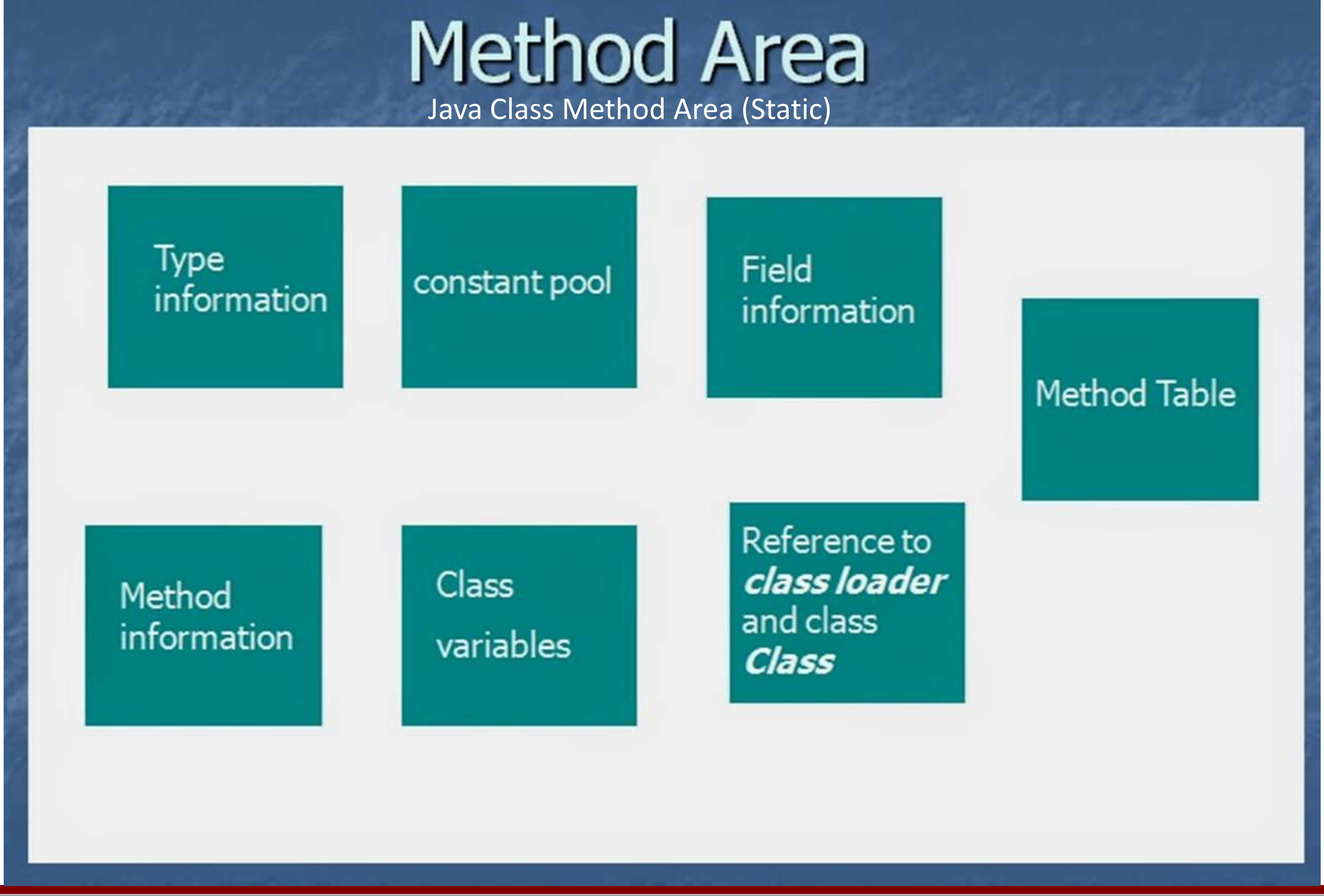
Stack Layout

SECTION 2

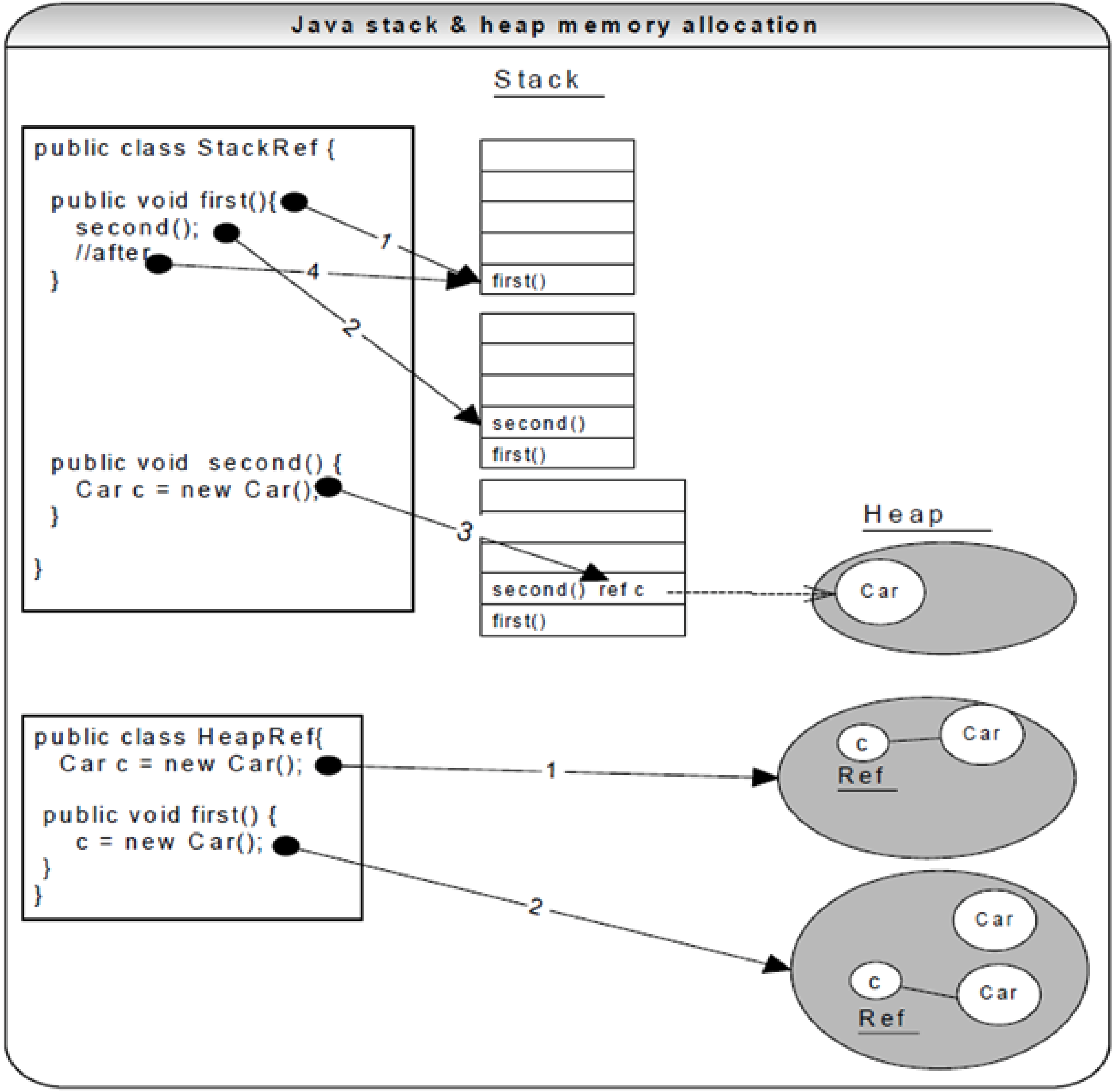
Review Of Stack Layout

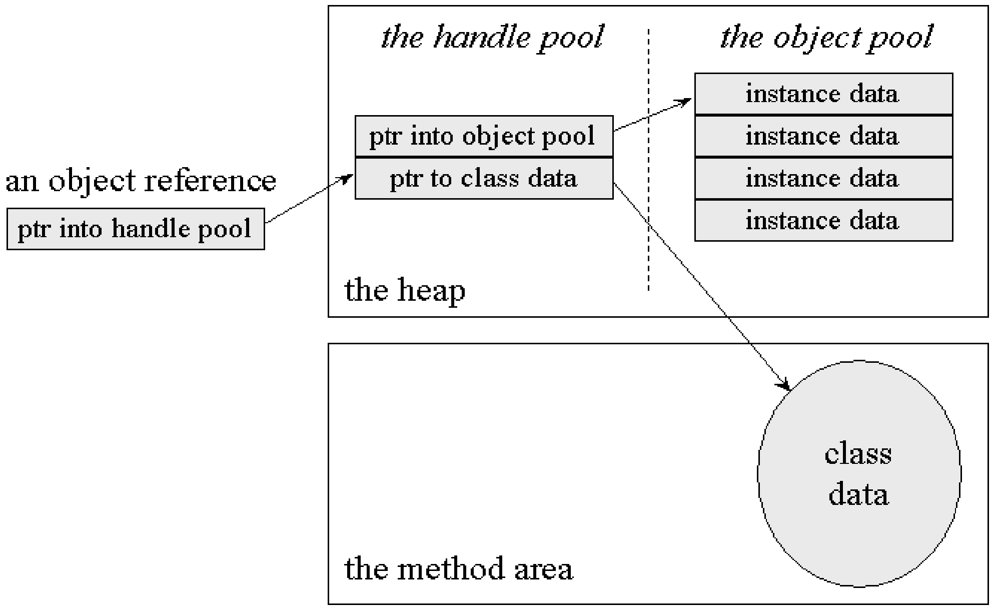
Allocation strategies

- **Static**
  - Code
  - Globals
  - Own variables
  - Explicit constants (including strings, sets, other aggregates)
  - Small scalars may be stored in the instructions themselves



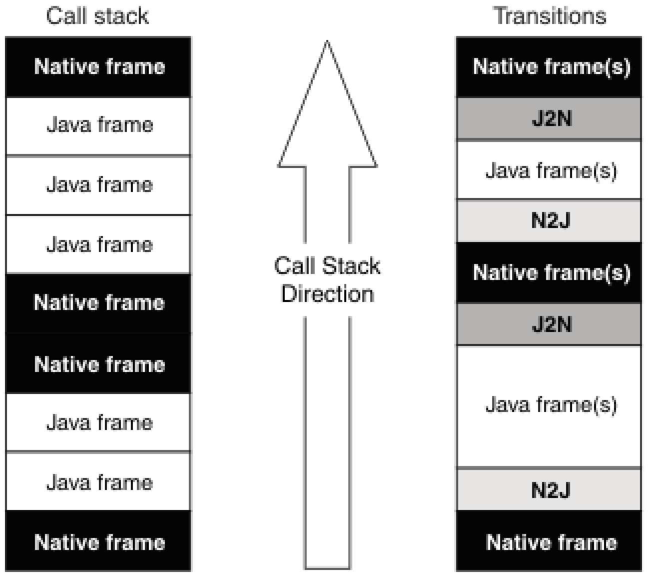
Java Class Method Area (Static)


Java stack & heap memory allocation
Stack
public class StackRef {
public void first(){
second();
//after
}
public void second() {
Car c = new Car();
}
}
first()
second()
first()
second() ref c
first()
Heap
Car
public class HeapRef{
Car c = new Car();
public void first() {
c = new Car();
}
}
c
Car
Ref
Car
c
Car
Ref

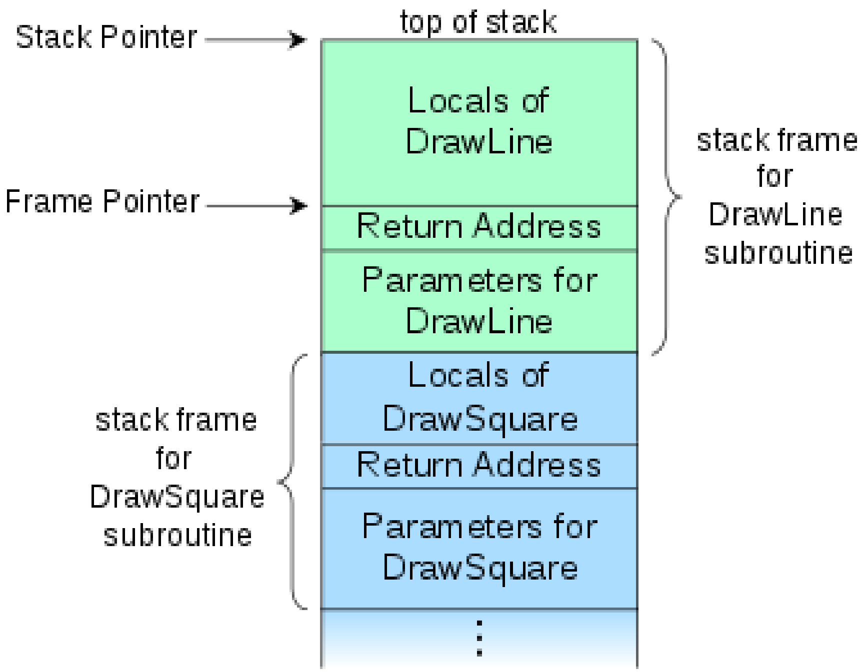

the handle pool
the object pool
instance data
instance data
instance data
instance data
ptr into object pool
ptr to class data
an object reference
ptr into handle pool
the heap
class data
the method area

Class Method Area In Stack (Dynamic)

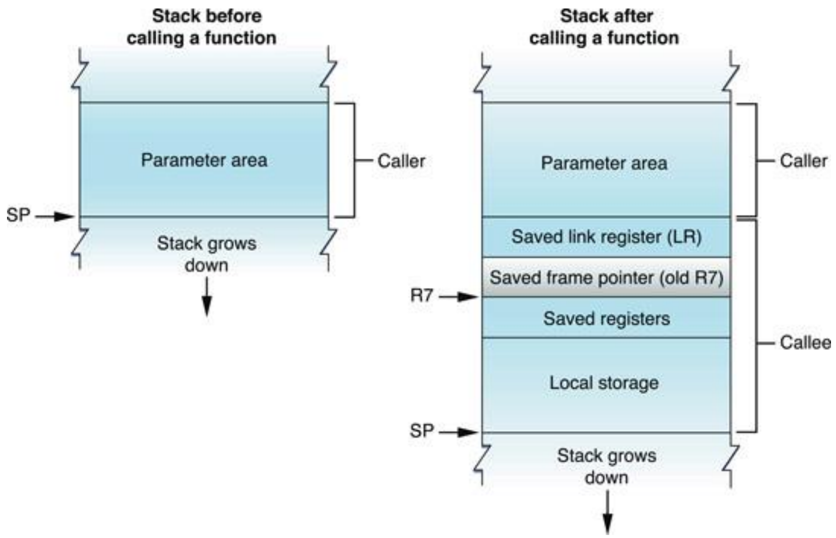
Growth of Call Stack



C Call Stack



Assembly Call Stack



Review Of Stack Layout

Allocation strategies

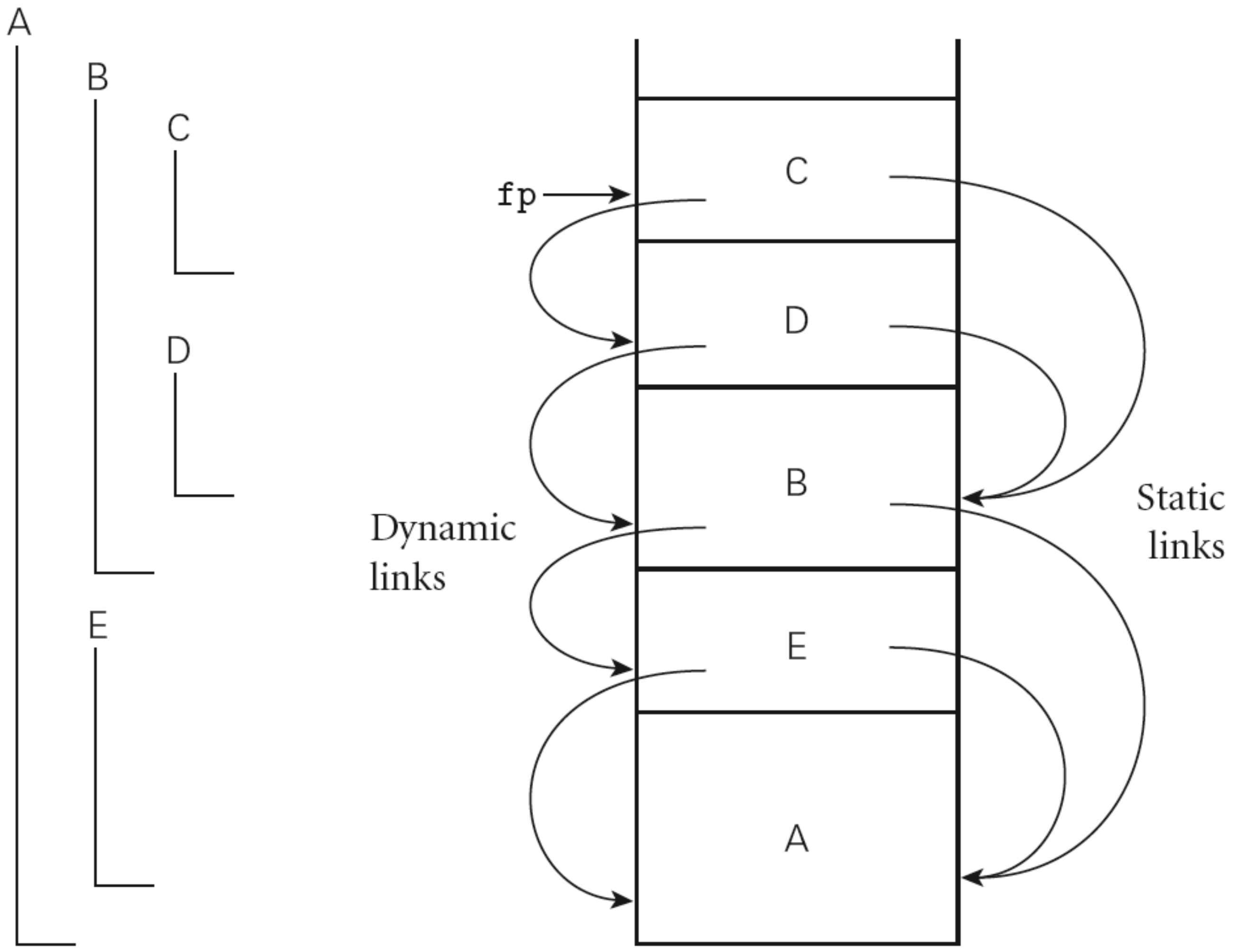


Figure 9.1 **Example of subroutine nesting, taken from Figure 3.5.** Within B, C, and D, all five routines are visible. Within A and E, routines A, B, and E are visible, but C and D are not. Given the calling sequence A, E, B, D, C, in that order, frames will be allocated on the stack as shown at right,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

Review Of Stack Layout

Allocation strategies (2)

- Stack
    - parameters
    - local variables
    - temporaries
    - bookkeeping information
- Heap
    - dynamic allocation

Review Of Stack Layout

Contents of a Stack Frame

- bookkeeping
    - return PC (dynamic link)
    - saved registers
    - line number
    - saved display entries
    - static link
- arguments and returns
- local variables
- temporaries

Calling Sequences

SECTION 3

Calling Sequences

- Maintenance of stack is responsibility of **calling sequence** and **subroutine prolog (call)** and **epilog (return)**
  - space is saved by putting as much in the prolog and epilog as possible
  - time **may** be saved by putting stuff in the caller instead, where more information may be known
    - e.g., there may be fewer registers IN USE at the point of call than are used SOMEWHERE in the callee

Task Must be Done

Prologue (Call):

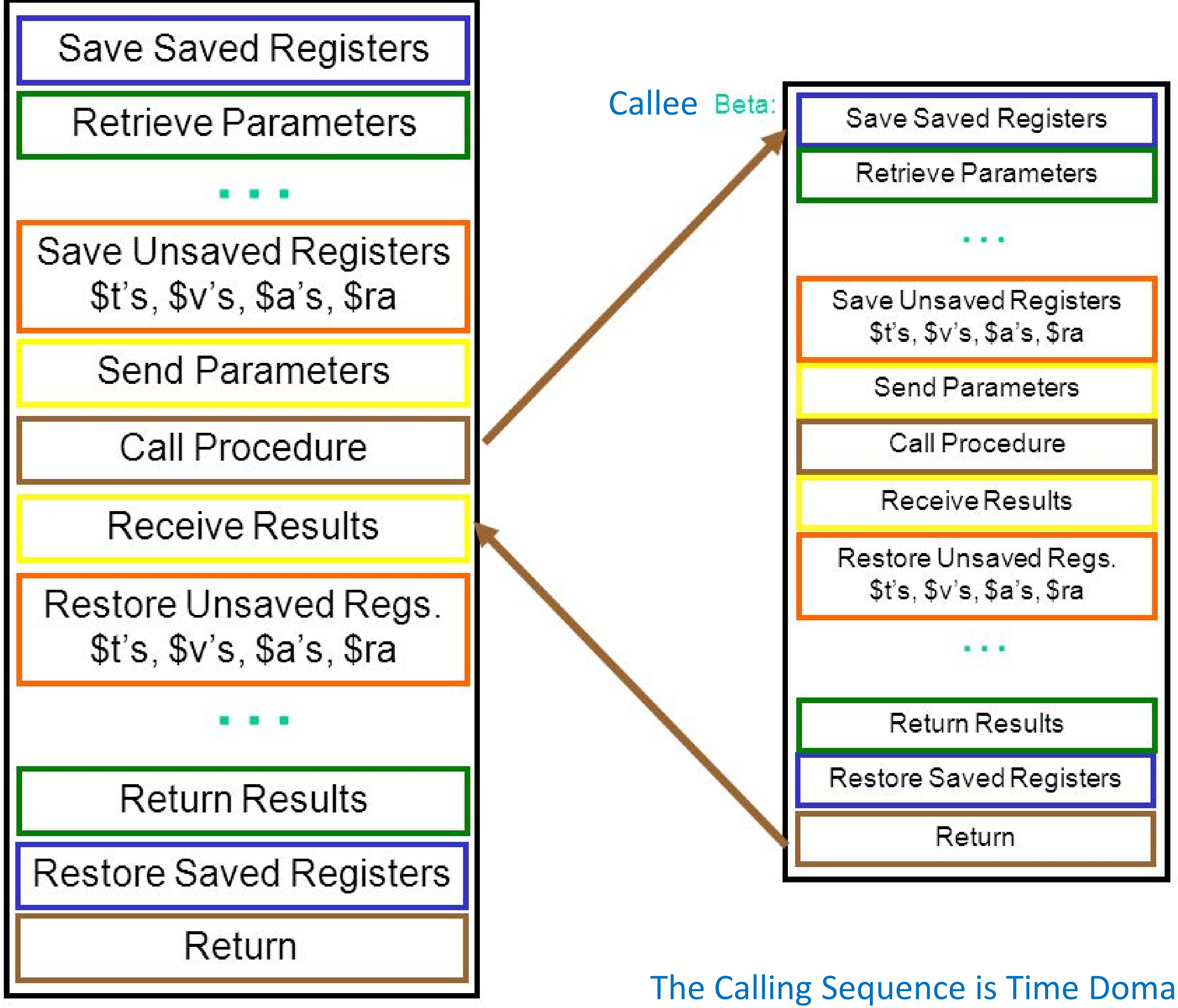
- Passing Parameters
- Saving the Return Address
- Changing the Program Counters
- Changing the Stack Pointer (Call Stack)
- Saving Registers
- Changing Frame Pointer to Refer to the New Frame
- Executing the Initialization Code for New Objects

Epilog (Return):

- Passing the Return Parameters or Function Values
- Executing the Finalization Code for the Local Objects
- Deallocating the Stack Frame
- Restoring other Stored Registers
- Restoring Program Counter

Structure of a Procedure

Caller Alpha:



The Calling Sequence is Time Domain not Spatial Domain.

Calling Sequences

- The ideal approach is to save those registers that are both live in the caller and needed for other purpose in the Callee.
- Hard to determine this intersecting set.
- Common strategy is to divide registers into **caller-saves** and **callee-saves** sets. (Of equal size)
  - caller uses the "callee-saves" registers first
  - "caller-saves" registers if necessary
- Local variables and arguments are assigned fixed OFFSETS from the stack pointer or frame pointer at compile time
  - some storage layouts use a separate arguments pointer
  - the VAX architecture encouraged this

More convenient names for registers

Register	Name	Use
R0	$0	Constant 0
R1	$at	Reserved Temp.
R2	$v0	Return Values
R3	$v1	
R4	$a0	Procedure arguments
R5	$a1	
R6	$a2	
R7	$a3	
R8	$t0	Caller Save Temporaries: May be overwritten by called procedures
R9	$t1	
R10	$t2	
R11	$t3	
R12	$t4	
R13	$t5	
R14	$t6	
R15	$t7	
R16	$s0	Callee Save Temporaries: May not be overwritten by called pro-cedures
R17	$s1	
R18	$s2	
R19	$s3	
R20	$s4	
R21	$s5	
R22	$s6	
R23	$s7	
R24	$t8	Caller Save Temp
R25	$t9	
R26	$k0	Reserved for Operating Sys
R27	$k1	
R28	$gp	Global Pointer
R29	$sp	Callee Save Stack Pointer
R30	$fp	Frame Pointer
R31	$ra	Return Address

Caller Save

“temp” or $t registers

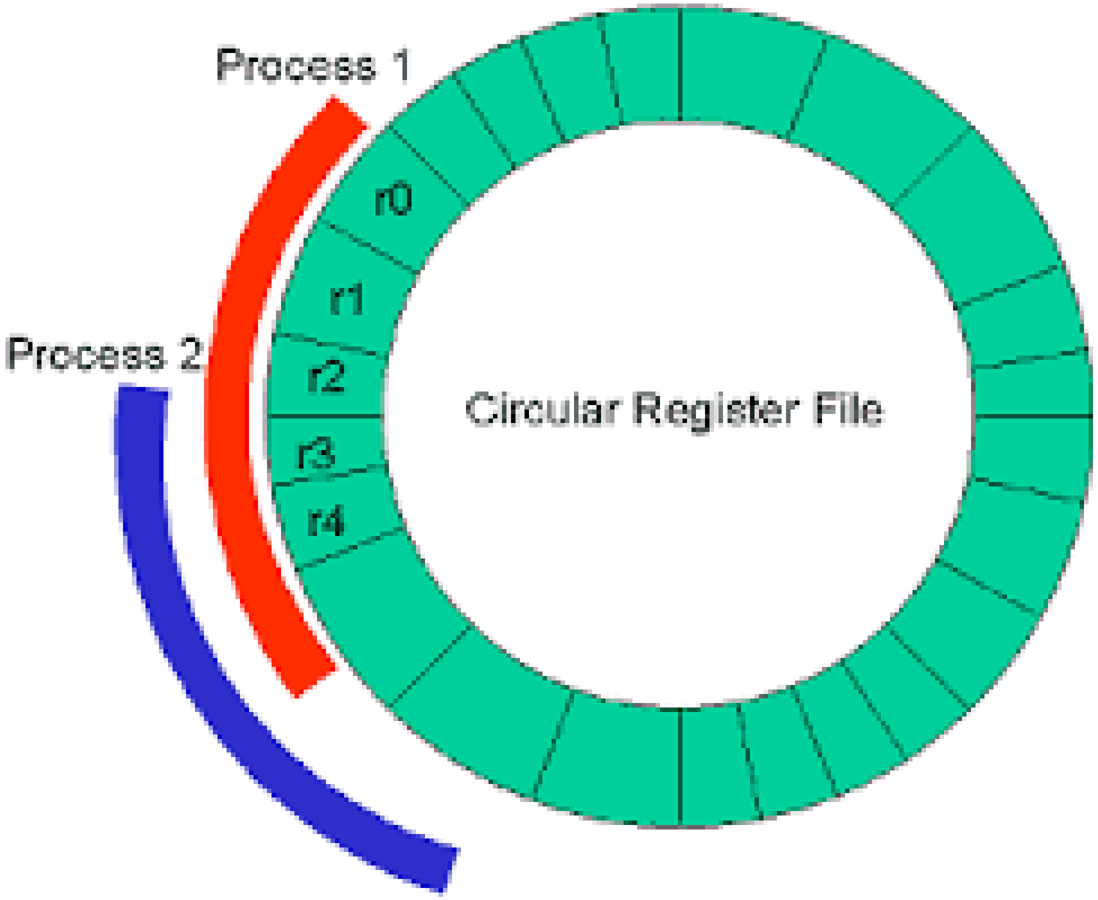
Callee Save

“save” or $s registers

Reg

Caller-Save
Callee-Save
Both save to memory (Stack)
A lot of Memory Cycles

Register Windows on RISC Machine



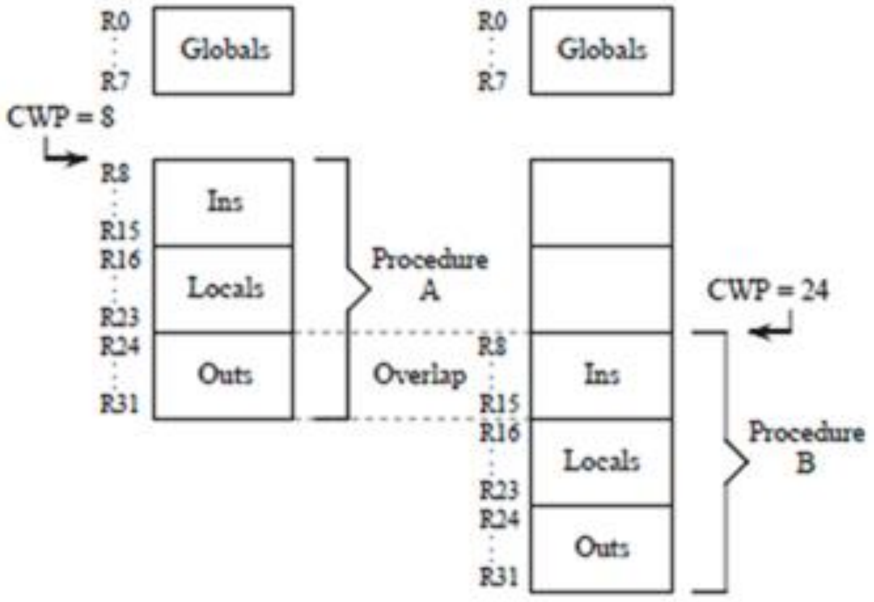
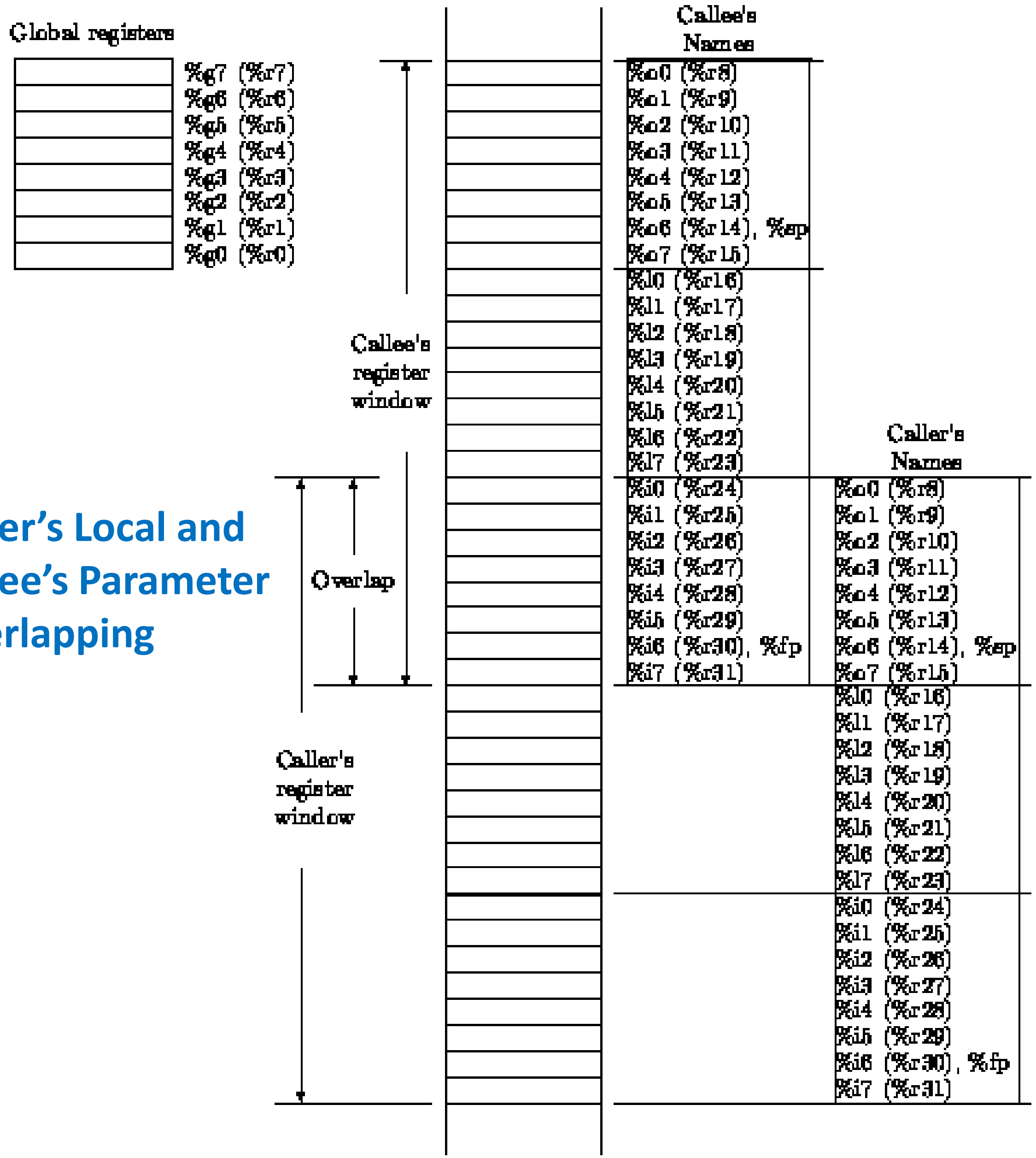


Figure 10-8 Overlapping register windows.

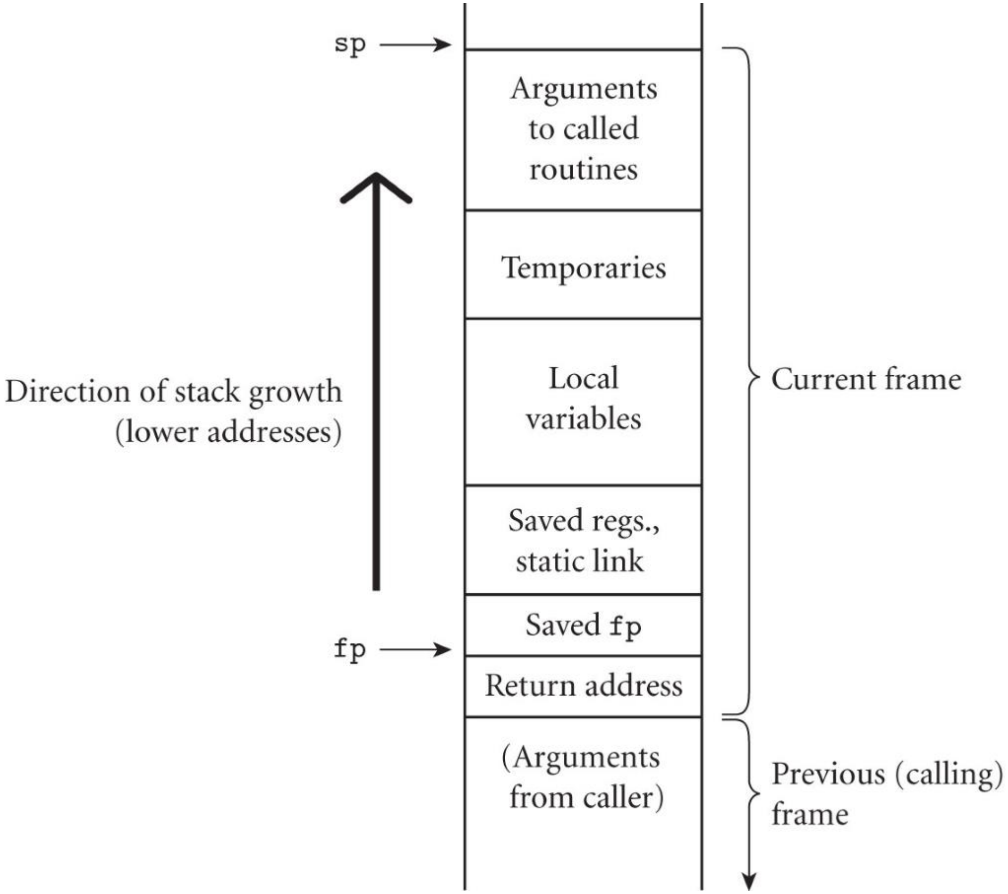
Caller’s Local and Callee’s Parameter Overlapping



Operation	Syntax	Operation implemented
save caller's register window	save rs_1, rs_2, rd	res = reg[rs_1]+reg[rs_2] CWP = (CWP−1) % NWINDOWS reg[rd] = res
	save rs_1, $siconst_{13}$, rd	res = reg[rs_1]+$siconst_{13}$ CWP = (CWP−1) % NWINDOWS reg[rd] = res
restore caller's register window	restore rs_1, rs_2, rd	res = reg[sr_1]+reg[sr_2] CWP = (CWP+1) % NWINDOWS reg[rd] = res
	restore rs, $siconst_{13}$, rd	res = reg[rs]+$siconst_{13}$ CWP = (CWP+1) % NWINDOWS reg[rd] = res
	restore	CWP = (CWP+1) % NWINDOWS

A Typical Calling Sequence

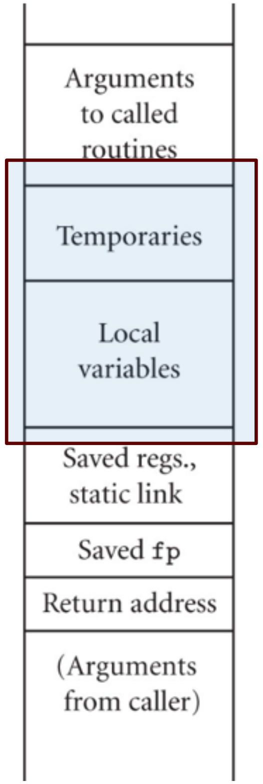
To maintain this stack layout, the calling sequence might operate as follows.
The caller
1. saves any caller-saves registers whose values will be needed after the call
2. computes the values of arguments and moves them into the stack or registers
3. computes the static link (if this is a language with nested subroutines), and passes it as an extra, hidden argument
4. uses a special subroutine call instruction to jump to the subroutine, simultaneously passing the return address on the stack or in a register
In its prologue, the callee
1. allocates a frame by subtracting an appropriate constant from the sp
2. saves the old frame pointer into the stack, and assigns it an appropriate new Value
3. saves any callee-saves registers that may be overwritten by the current routine (including the static link and return address, if they were passed in registers)
After the subroutine has completed, the epilogue
1. moves the return value (if any) into a register or a reserved location in the stack
2. restores callee-saves registers if needed
3. restores the fp and the sp
4. jumps back to the return address
Finally, the caller
1. moves the return value to wherever it is needed
2. restores caller-saves registers if needed



Calling Sequences (LLVM on ARM)

Caller

- saves into the “local variable and temporaries” area any caller-saves registers whose values are still needed
- puts up to 4 small arguments into registers r0-r3
- puts the rest of the arguments into the argument build area at the top of the current frame
- does b1 or b1x, which puts return address into register lr, jumps to target address, and (optionally) changes instruction set coding



Low Level Virtual Machine (LLVM)

LLVM is written in C++ and is designed for compile-time, link-time, run-time, and "idle-time" optimization of programs written in arbitrary programming languages. Originally implemented for C and C++, the language-agnostic design of LLVM has since spawned a wide variety of front ends: languages with compilers that use LLVM include ActionScript, Ada, C#, Common Lisp, Crystal, D, Delphi, Fortran, OpenGL Shading Language, Halide, Haskell, Java bytecode, Julia, Lua, Objective-C, Pony, Python, R, Ruby, Rust, CUDA, Scala, and Swift.

Calling Sequences (LLVM on ARM)

- **In prolog, Callee**
  - pushes necessary registers onto stack
  - initializes frame pointer by adding small constant to the sp placing result in r7
  - subtracts from sp to make space for local variables, temporaries, and arg build area at top of stack
- **In epilog, Callee**
  - puts return value into r0-r3 or memory (as appropriate)
  - subtracts small constant from r7, puts result in sp (effectively deallocates most of frame)
  - pops saved registers from stack, pc takes place of lr from prologue (branches to caller as side effect)

Calling Sequences (LLVM on ARM)

- After call, Caller
  - moves return value to wherever it's needed
  - restores caller-saves registers lazily over time, as their values are needed
- All arguments have space in the stack, whether passed in registers or not
- The subroutine just begins with some of the arguments already cached in registers, and 'stale' values in memory

Calling Sequences (LLVM on ARM)

- This is a normal state of affairs; optimizing compilers keep things in registers whenever possible, flushing to memory only when they run out of registers, or when code may attempt to access the data through a pointer or from an inner scope

Calling Sequences (LLVM on ARM)

- Many parts of the calling sequence, prologue, and/or epilogue can be omitted in common cases
  - particularly LEAF routines (those that don't call other routines)
    - leaving things out saves time
    - simple leaf routines don't use the stack - don't even use memory – and are exceptionally fast

Parameter Passing I

Parameter Mode and Call by Value

SECTION 4

Parameter Passing

Formal Parameter

Method Declaration
```
public void setText(String text)
```

Method Invocation
```
lblMyName.setText("Del Searls");
```

Actual Parameter

Java: reference data being passed.
(Passing Modes)

Constant Pool or Heap

"Del Seals"

Parameter Passing

- Parameter passing mechanisms have four basic implementations (Parameter Modes)
  - value
  - value/result (copying)
  - reference (aliasing)
  - closure/name
- Many languages (e.g. Ada, C++, Swift) provide value and reference directly

Parameter Passing

Passed by Value and Passed by Reference

- C/C++: functions
  - parameters passed by value (C)
  - parameters passed by reference can be simulated with pointers (C)

```
void proc(int* x,int y){*x = *x+y } …
proc(&a,b);
```

  - or directly passed by reference (C++)

```
void proc(int& x, int y) {x = x + y }
proc(a,b);
// a will be changed after proc function.
```

Value and Reference Parameters

```
x : integer                          -- global
procedure foo(y : integer)
      y := 3
      print x
. . .
x := 2
foo(x)
print x
```

Call by Value: foo(2)

y= 2 (parameter passing)

y= 3 (assignment)

x no change -> print 2

Call by Reference: x is passed

y = 2 (by assignment.)

X = 2 (by sharing data with y

y = 3 (by assignment.)

x also equals to 3.

x changed

- If y is passed to foo by value, then the assignment inside foo has no visible effect—y is private to the subroutine—and the program prints 2.
- If y is passed to foo by reference, then the assignment inside foo changes x—y is just a local name for x—and the program prints 3 twice.

Emulating Call by Reference in C:

```
void swap(int *a, int *b) { int t = *a; *a = *b; *b = t; }
...
swap(&v1, &v2);
```

Parameter Passing

Input Parameters and Output Parameters (Return Value)

- Ada goes for semantics: who can do what
  - *In*: callee reads only
  - *Out*: callee writes and can then read (formal not initialized); actual modified
  - *In out*: callee reads and writes; actual modified
- Ada in/out is always implemented as
  - value/result for scalars, and either
  - value/result or reference for structured objects

Pass by Sharing

- In a language with a reference model of variables (ML, Ruby), pass by *sharing (reference*) is the obvious approach
- It's also the only option in Fortran
  - If you pass a constant, the compiler creates a temporary location to hold it
  - If you modify the temporary, who cares?
- **Call by Reference and Call by Sharing:**
  - They are the same in the sense that both can change the value of the parameter variable.
  - They are different that parameter in call-by-reference can point to another object (change reference), but call-by sharing cannot.

Read-Only Parameters

const Parameters in C

```
void append_to_log(const huge_record* r) { ...
...
append_to_log(&my_record);
```

- Parameter received value from the caller.
- At caller, the parameter variable works like a local variable if it is called by value. It works like a global variable if it is called by reference.

Parameter Passing II

Call by Reference, Name and Closure

SECTION 5

References in C++

- Programmers who switch to C after some experience with Pascal, Modula, or Ada (or with call-by-sharing in Java or Lisp) are often frustrated by C's lack of reference parameters.
- One can always arrange to modify an object by passing its address, but then the formal parameter is a pointer, and must be explicitly dereferenced whenever it is used.
- C++ addresses this problem by introducing an explicit notion of a **reference**.
- Reference parameters are specified by preceding their name with an ampersand in the header of the function:

Reference Variables:

```
void swap(int &a, int &b) { int t = a; a = b; b = t; }
```

```
int i;
int &j = i;
...
i = 2;
j = 3;
cout << i;              // prints 3
```

j share the same int with i.
j is an alias of i

C++ break the notion of & and *.
Sharing address by assigning reference

Simplify Code with an In-line Alias

equivalent

```
{
  element *e = &ruby.chemical_composition.elements[1];
  e->name ="Al";
  e->atomic_number=13;
  e->atomic_weight = 26.98154;
  e->metallic = true;
}
```

```
{
  element& e = ruby.chemical_composition.elements[1];
  e.name ="Al";
  e.atomic_number=13;
  e.atomic_weight = 26.98154;
  e.metallic = true;
}
```

function

std::operator<< (string)

<string>

```
ostream& operator<< (ostream& os, const string& str);
```

Insert string into stream

Inserts the sequence of characters that conforms value of *str* into *os*.

This function overloads `operator<<` to behave as described in ostream::operator<< for c-strings, but applied to string objects.

Parameters

`os`

ostream object where characters are inserted.

`str`

string object with the content to insert.

Return Value

The same as parameter *os*.

Returning a reference from a function

The overloaded << and >> operators return a reference to their first argument, which can in turn be passed to subsequent << or >> operations.

```
cout << a << b << c;
```

is short for

```
((cout.operator<<(a)).operator<<(b)).operator<<(c);
```

Without references, << and >> would have to return a pointer to their stream:

```
((cout.operator<<(a))->operator<<(b))->operator<<(c);
```

or

```
*(*(cout.operator<<(a)).operator<<(b)).operator<<(c);
```

This change would spoil the cascading syntax of the operator form:

```
*(*(cout << a) << b) << c;
```

R-value Reference string&& str;

```
// lvalues:
//
int i = 42;
i = 43; // ok, i is an lvalue
int* p = &i; // ok, i is an lvalue
int& foo();
foo() = 42; // ok, foo() is an lvalue
int* p1 = &foo(); // ok, foo() is an lvalue

// rvalues:
//
int foobar();
int j = 0;
j = foobar(); // ok, foobar() is an rvalue
int* p2 = &foobar(); // error, cannot take the address of an rvalue
j = 42; // ok, 42 is an rvalue
```

R-value (rvlaue, r-value)
Denotes temporary objects which don’t have a name. (object/body)

String str = “String”;

*a (pointer), &a (reference: lvalue), &&a (object/body: rvalue)

R-value Reference

Copy Constructor (Create object or copy reference)

r-value reference allow an argument that would normally be considered an **r-value** to be passed to a function by reference.

```
obj o2 = o1; // obj o2 = new obj(o1);
```

- Computer will initialize o2 by calling obj's copy constructor method, passing o1 as argument. The parameter of obj's copy constructor would be a constant reference (const obj&). The body of the constructor would inspect to decide how to initialize o2.
- If the state is mutable, the constructor will generally need to allocate and initialize a copy (not for “”string” in Java).

```
obj o3 = foo(“hi, mom”);  // return a temporary
                          // reference t
```

*a (pointer), &a (reference: lvalue), &&a (object/body: rvalue)

R-value Reference(&a, &&a)

move constructor

Move Constructor: obj && (double ampersand, no const)

Implicitly move

```
obj::obj(obj&& other){
    payload = other.payload;
    other.payload = nullptr;
} // move constructor - other is a
  //reference to a reference
```

A move constructor of class T is a non-template constructor whose first parameter is `T&&`, `const T&&`, `volatile T&&`, or `const volatile T&&`, and either there are no other parameters, or the rest of the parameters all have default values.

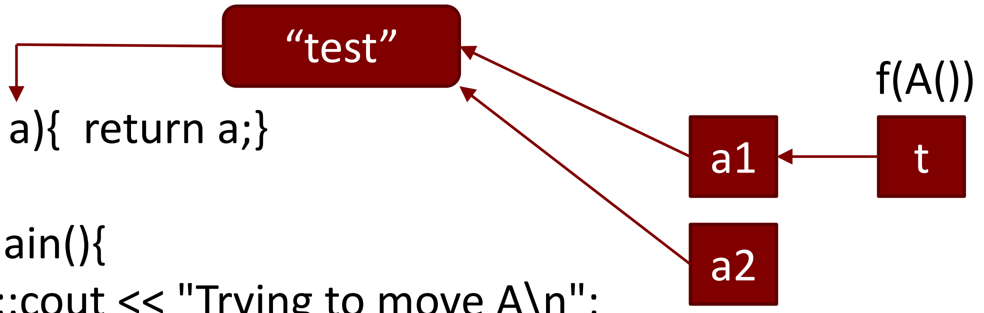
R-value Reference(&a, &&a)

move constructor

```
struct A{
    std::string s;
    A() : s("test") { }
    A(const A& o) : s(o.s) { std::cout << "move failed!\n"; }
    A(A&& o) noexcept : s(std::move(o.s)) { }
};

A f(A a){  return a;}

int main(){
  std::cout << "Trying to move A\n";
  A a1 = f(A()); // move-construct from rvalue temporary
  A a2 = std::move(a1); // move-construct from xvalue
}
```



- Programmer may know that a value will never be used after passing it as a parameter, but compiler may not know. Programmer can wrap the value in a call to move function (standard library).
- **move** generate no code: it is, in effect, a cast. Behavior undefined if the program actually does contain a subsequent use of o3.

```
obj o4 = std::move(o3);
```

*a (pointer), &a (reference: lvalue), &&a (object/body: rvalue)

Call by Closure

Subroutine as a Parameter

Closure: a reference to a subroutine, together with its referencing environment. (compare to C-Macro)

```
#define PORTIO asm
        {
        asm mov al, 2
        asm mov dx, 0xD007
        asm out dx, al
        }
```

```
type int_func is access function (n : integer) return integer;
type int_array is array (positive range<>) of integer;
begin
   for i in A'range loop
         A(i) := f(A(i));
   end
end apply_to_A;
      …
      k: integer := 3;      - in nested scope
      …
      function add_k (m: integer) return integer is
        begin
            return m+k;
        end add_k;
        …
        apply_to_A (add_k'access, B);
```

Other Call-by-Closure

(function as parameter)

First class subroutines in Scheme:

```
(define apply-to-L (lambda (f l)
    (if (null? l) '()
        (cons (f (car l)) (apply-to-L f (cdr l))))))
```

First class subroutines in ML:

```
fun apply_to_L(f, l) =
    case l of
        nil    => nil
      | h :: t => f(h) :: apply_to_L(f, t);
```

Sub-rountine Pointer in C/C++:

```
void apply_to_A(int (*f)(int), int A[], int A_size) {
    int i;
    for (i = 0; i < A_size; i++) A[i] = f(A[i]);
}
```

- Subroutines are routinely passed as parameters (and returned as results) in functional languages. A list-based version of `apply_to_A` would look some-thing like this in **Scheme**.

Call by Name

- Call-by name is an old Algol technique
  - Think of it as call by textual substitution (procedure with all name parameters works like macro) - what you pass are hidden procedures called **THUNKS**
- Call by Name: Algo-60 and Simula.
- Call by Need: Miranda, Haskell, R

Parameter mode	Representative languages	Implementation mechanism	Permissible operations	Change to actual?	Alias?
value	C/C++, Pascal, Java/C# (value types)	value	read, write	no	no
`in`, `const`	Ada, C/C++, Modula-3	value or reference	read only	no	maybe
`out`	Ada	value or reference	write only	yes	maybe
value/result	Algol W	value	read, write	yes	no
`var`, `ref`	Fortran, Pascal, C++	reference	read, write	yes	yes
sharing	Lisp/Scheme, ML, Java/C# (reference types)	value or reference	read, write	yes	yes
r-value ref	C++11	reference	read, write	yes*	no*
`in out`	Ada, Swift	value or reference	read, write	yes	maybe
name	Algol 60, Simula	closure (thunk)	read, write	yes	yes
need	Haskell, R	closure (thunk) with memoization	read, write†	yes†	yes†

Figure 9.3 **Parameter-passing modes.** Column 1 indicates common names for modes. Column 2 indicates prominent languages that use the modes, or that introduced them. Column 3 indicates implementation via passing of values, references, or closures. Column 4 indicates whether the callee can read or write the formal parameter. Column 5 indicates whether changes to the formal parameter affect the actual parameter. Column 6 indicates whether changes to the formal or actual parameter, during the execution of the subroutine, may be visible through the other. *Behavior is undefined if the program attempts to use an r-value argument after the call. †Changes to arguments passed by need in R will happen only on the first use; changes in Haskell are not permitted.

Parameter Passing III

Special Parameters

SECTION 6

Special Purpose Parameters

1. Conformant Arrays
2. Default (Optional) Parameters
3. Named Parameters
4. Variable Numbers of Arguments

Conformant Arrays

- A formal array parameter whose shape is finalized at run time (in a language that usually determines shape at compile time), is called a **conformant**, or open, array parameter.

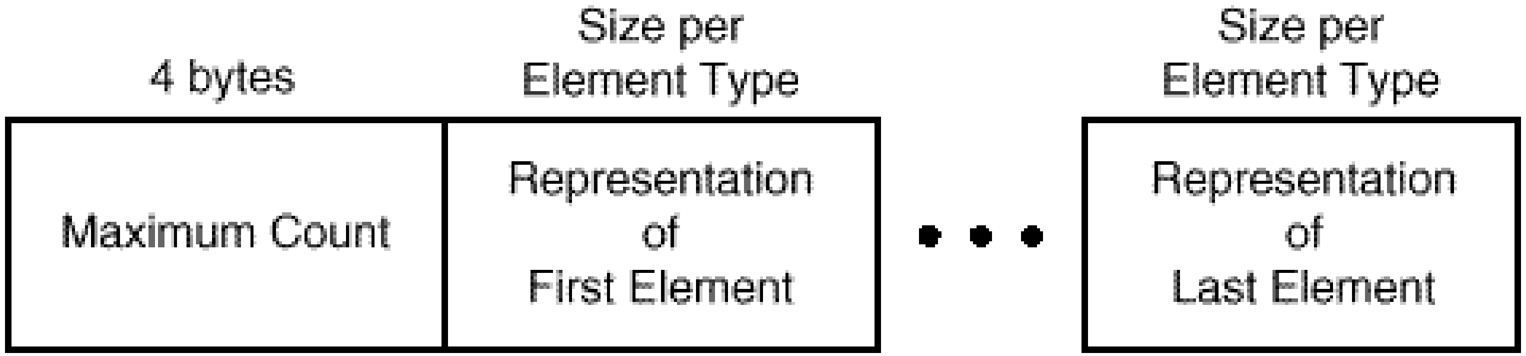


Figure Uni-dimensional Conformant Array Representation

Default (Optional) Parameters

Ada:

```
type field is integer range 0..integer'last;
type number_base is integer range 2..16;
default_width : field       := integer'width;
default_base  : number_base := 10;
procedure put(item  : in integer;
              width : in field       := default_width;
              base  : in number_base := default_base);
```

- One common use of default parameters is in I/O library routines.
- The declaration of **default_width** uses the built-in type attribute width to determine the maximum number of columns required to print an integer in decimal on the current machine.

- Any formal parameter that is “assigned” a value in its subroutine heading is optional in Ada. In our **text_IO** example, the programmer can call put with one, two, or three arguments.
- No matter how many are provided in a particular call, the code for put can always assume it has all three parameters.
- The implementation is straightforward: in any call in which actual parameters are missing, the compiler pretends as if the defaults had been provided; it generates a calling sequence that loads those defaults into registers or pushes them onto the stack, as appropriate.

Named Parameters

In position:

```
put(item => 37, base => 8);
```

Out of position:

```
put(base => 8, item => 37);
```

Mixed:

```
put(37, base => 8);
```

```
format_page(columns => 2,
    window_height => 400, window_width => 200,
    header_font => Helvetica, body_font => Times,
    title_font => Times_Bold, header_point_size => 10,
    body_point_size => 11, title_point_size => 13,
    justification => true, hyphenation => false,
    page_num => 3, paragraph_indent => 18,
    background_color => white);
```

- In all of our discussions so far we have been assuming that parameters are positional: the first actual parameter corresponds to the first formal parameter, the second actual to the second formal, and so on.
- In some languages, including Ada, Common Lisp, Fortran 90,Modula-3, and Python, this need not be the case. These languages allow parameters to be named.

Parameter Passing IV

Return

SECTION 7

Function Returns

Return Statement (return value is an out-going parameter)

Return the result of an expression:

return expression;

Return the value of a variable:

rtn = expression;

…

return rtn;

Return as the break of a function:

return;

- The syntax by which a function indicates the value to be returned varies greatly.
- In Algol 60, Fortran, and Pascal, a function specifies its return value by executing an assignment statement whose left-hand side is the name of the function. (abandoned by other languages). [**no return statement**]
- A function that has figured out what to return but doesn't want to return yet can always assign the return value into a temporary variable, and then return it later.

Function Returns

Incremental computation of a return value

Ada:

```
type int_array is array (integer range <>) of integer;
    -- array of integers with unspecified integer bounds
function A_max(A : int_array) return integer is
rtn : integer;
begin
    rtn := integer'first;
    for i in A'first .. A'last loop
        if A(i) > rtn then rtn := A(i); end if;
    end loop;
    return rtn;
end A_max;
```

- Here **rtn** must be declared as a variable so that the function can read it as well as write it. Because **rtn** is a local variable, most compilers will allocate it within the stack frame of **A_max**.
- The return statement must then perform an unnecessary copy to move that variable's value into the return location allocated by the caller.

Function Returns

Explicitly named return values in SR

SR:

```
procedure A_max(ref A[1:*]: int) returns rtn : int
    rtn := low(int)
    fa i := 1 to ub(A) ->
        if A[i] > rtn -> rtn := A[i] fi
    af
end
```

- Some languages eliminate the need for a local variable by allowing the result of a function to have a name in its own right.

- Here **`rtn`** can reside throughout its lifetime in the return location allocated by the caller.

- A similar facility can be found in **Eiffel**, in which every function contains an implicitly declared object named **Result**. This object can be both read and written, and is returned to the caller when the function returns.

Function Returns

Multivalue returns

Python:

```
def foo():
    return 2, 3
...
i, j = foo()
```

- ML, its descendants, and several scripting languages allow a **`Multivalue returns`** function to return a **tuple** of values.

- Many languages place restrictions on the types of objects that can be returned from a function. In Algol 60 and Fortran 77, a function must return a **scalar** value.
- In Pascal and early versions of Modula-2, it must return **a scalar or a pointer**.
- Most imperative languages are more flexible: Algol 68, Ada, C, Fortran 90, and many (nonstandard) implementations of Pascal allow functions to return **values of composite type**.