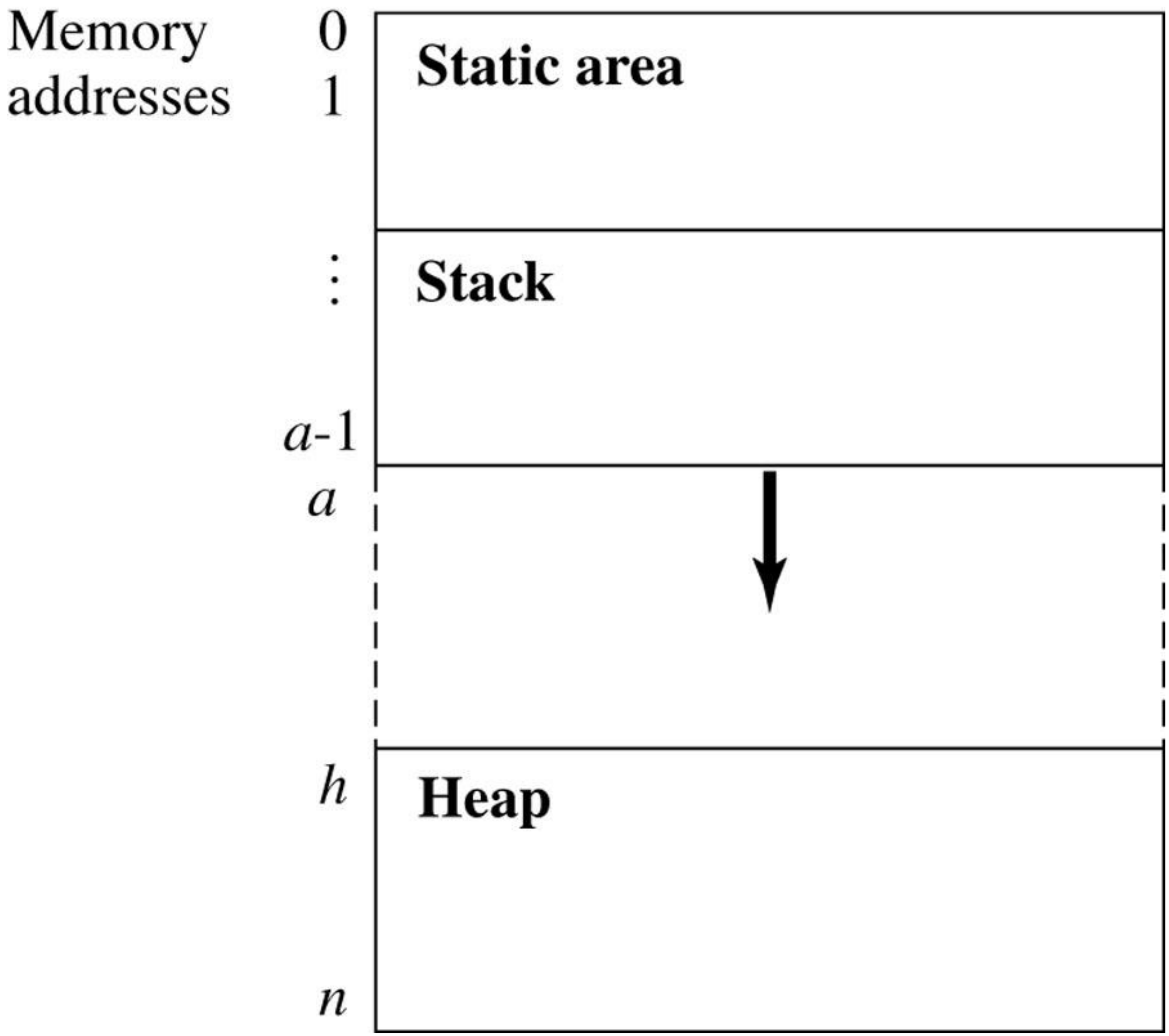
Chapter 9

Subprograms

The Structure of Run-Time Memory



Subprograms

- Two fundamental abstraction facilities in programming language:
  - Process abstraction – represented by subprograms
  - Data abstraction

- General characteristics of subprograms:
  1. A subprogram has a single entry point
  2. The caller is suspended during execution of the called subprogram
  3. Control always returns to the caller when the called subprogram's execution terminates

Subprograms

- A subprogram definition is a description of the actions of the subprogram abstraction
- A subprogram call is an explicit request that the subprogram be executed
  - A subprogram is active if, after being called, it has begun execution but has not yet completed that execution
- A subprogram header is the first line of the definition
  - Specifies that the following syntactic unit is a subprogram of some particular kind - using a special word (function, procedure, etc)
  - Provides name of subprogram
  - Specifies the list of formal parameters

    Fortran: Subroutine Adder(parameters)

    Ada: procedure Adder(parameters)

Subprograms

- The parameter profile (signature) of a subprogram is the number, order, and types of its parameters
- The protocol of a subprogram is its parameter profile plus, if it is a function, its return type
- Subprograms can have declarations as well as definitions
- Subprogram declaration provides the subprogram's protocol but do not include their bodies
  - Function declarations in C/C++ are called prototypes

Parameters

- A formal parameter is a dummy variable listed in the subprogram header and used in the subprogram

- An actual parameter represents a value or address used in the subprogram call statement

```
void doNothing (int formal_param) {
        …
}
main() {
        int actual_param;
        doNothing(actual_param);
}
```

Parameters

- Actual/Formal Parameter Correspondence
  - ⇨ Binding of actual to formal parameters (type checking)
  1. Positional parameters
     - First actual param bound to first formal param, etc
  2. Keyword parameters
     - Name of formal param to which actual param is bound is specified with actual param

       Ada: Sumer(Length => My_Length,
       List => My_Array,
       Sum => My_Sum);

     - Advantage: order is irrelevant
     - Disadvantage: user must know the formal parameter's names

Parameters

- Default values of formal parameters
  - Allowed by C++, Fortran 95, Ada and PHP
  - Default value is used if no actual parameter is passed to the formal parameter

Ada:
```
function Compute_Pay(   Income : Float; Exemptions : Integer := 1;
                        Tax_Rate : Float ) return Float
pay := Compute_Pay (20000.00, Tax_Rate => 0.15);
```

  - C# allows methods to accept variable number of params of the same type

```
public void DisplayList(params int[] list ) {
    foreach (int nextValue in list) {
        Console.WriteLine("Next value {0}", nextValue);
    }
}
```

Procedures and Functions

- A **function** is called from within an expression and returns a result after invocation. A **procedure** is treated as an atomic statement and does not return a result after invocation.
- Procedures provide user-defined statements
- Functions provide user-defined operators
  - Value produced by function is returned to the calling code, effectively replacing the call itself

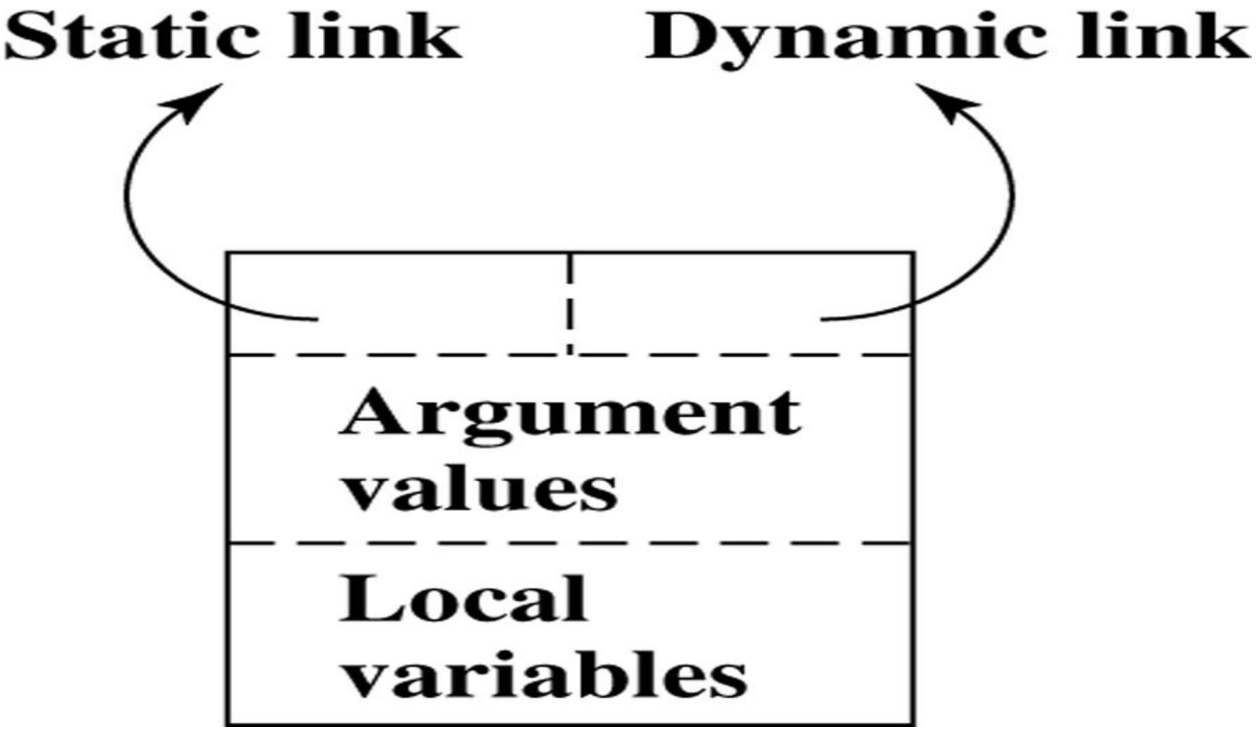
```
float power(float base, float exp)

result = 3.4 * power(10.0, x);
```

- C-based languages
  - have only functions (but they can behave like procedures)
  - Can be defined to return no value if the return type is void

Local Referencing Environments

- Local variables: variables defined inside subprograms
  - their scope is the body of subprogram in which they are defined
    - Stack-dynamic: bound to storage when subprogram begins execution, unbound when its execution terminates
      - Advantages:
        1. Support for recursion
        2. Storage for local variables of active subprogram can be shared with local variables of inactive subprograms
      - Disadvantages:
        1. Allocation/deallocation time
        2. Indirect addressing (indirectness because the place in stack where a particular local variable is stored can only be determined at run time)
        3. Subprograms cannot be history sensitive
           - Cannot retain data values of local variables between calls
    - Static: bound to storage at compile-time

Structure of a Called Method's Stack Frame (Activation record

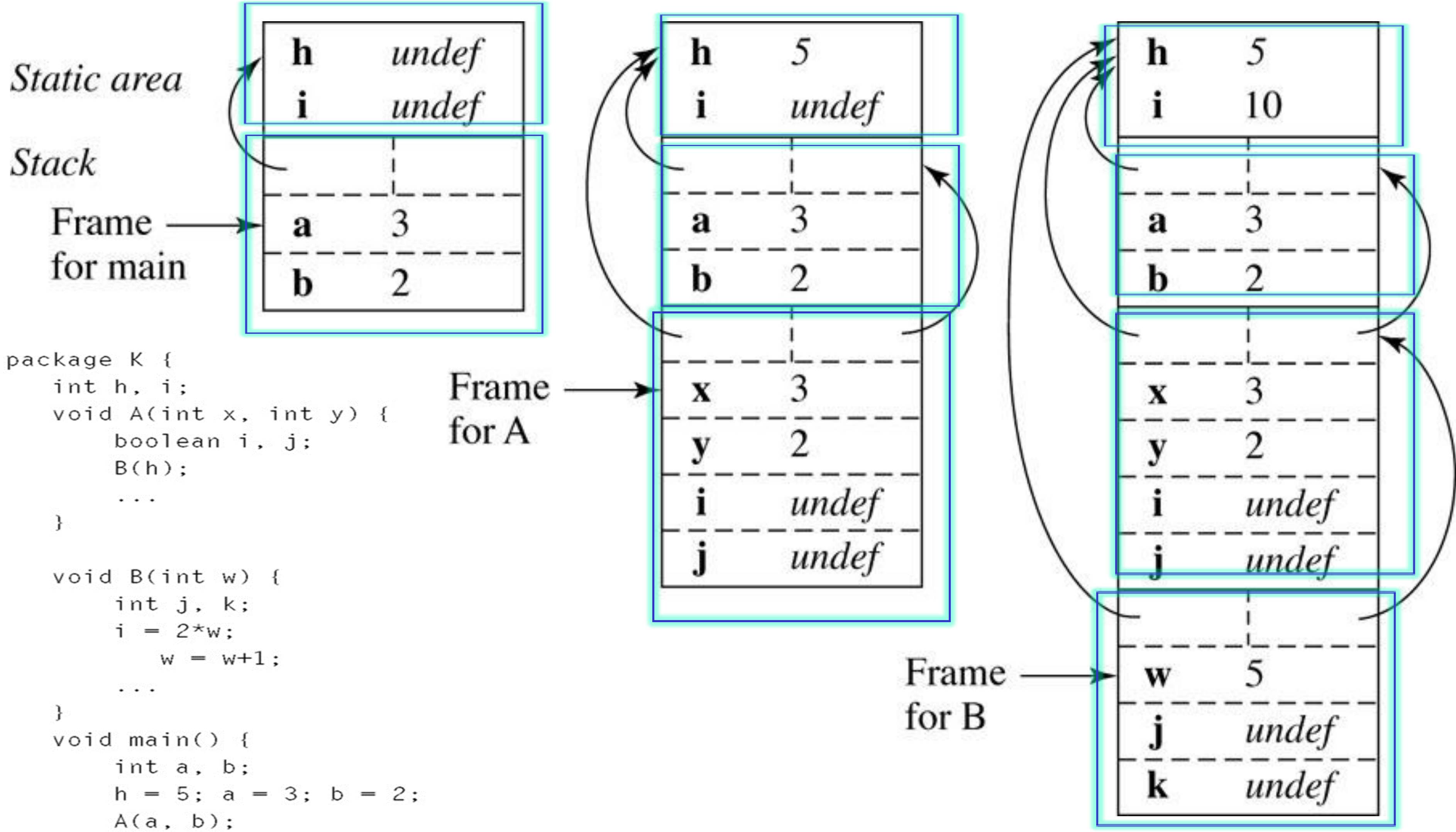


- Run-time activation of subprograms that are managed with a stack of Activation Record Instances (ARIs).
- The dynamic link is a pointer to the base of the activation record instance of the caller. In static-scoped languages, this link is used to provide traceback information when a run-time error occurs. In dynamic-scoped languages, the dynamic link is used to access nonlocal variables.
- The static link is a pointer to Static area.

Example Program with Methods and Parameters

```
package K {
    int h, i;
    void A(int x, int y) {
        boolean i, j;
        B(h);
        ...
    }

    void B(int w) {
        int j, k;
        i = 2*w;
           w = w+1;
        ...
    }
    void main() {
        int a, b;
        h = 5; a = 3; b = 2;
        A(a, b);
        ...
    }
}
```

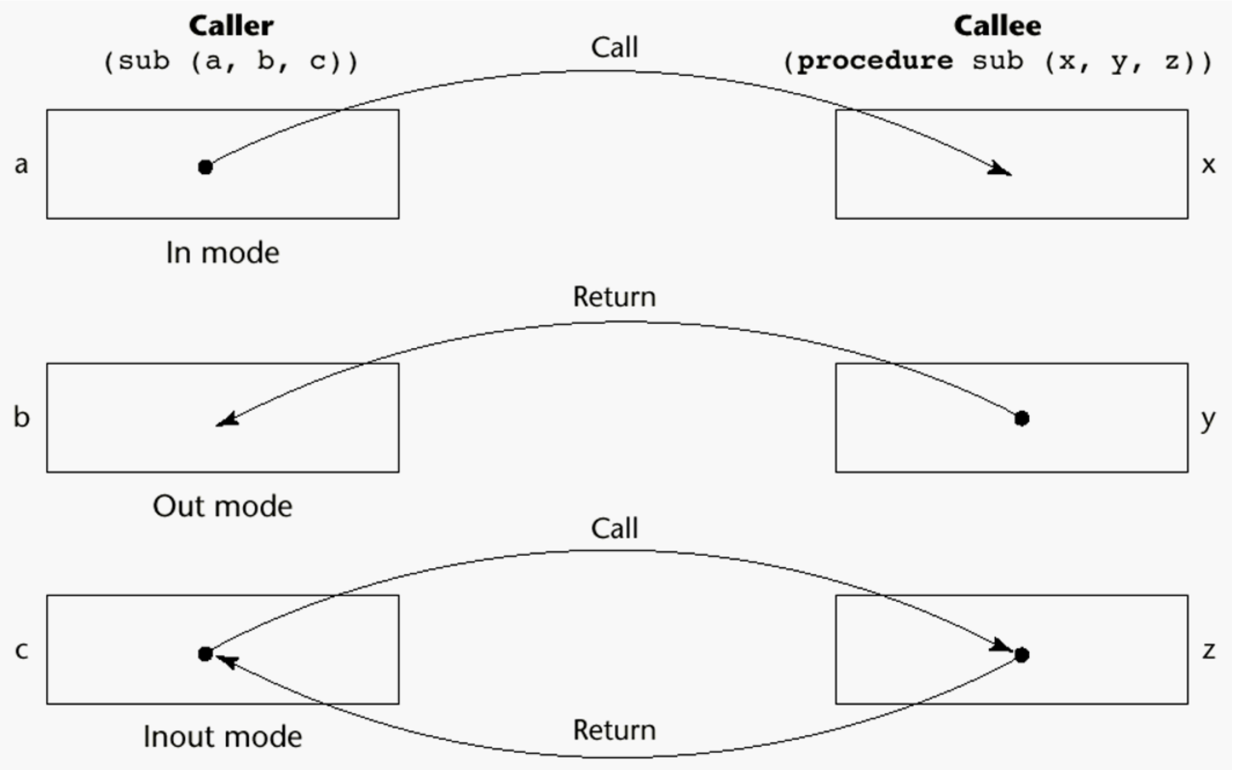
Run-Time Stack with Stack Frames for Method Invocations




```
package K {
   int h, i;
   void A(int x, int y) {
       boolean i, j;
       B(h);
       ...
   }

   void B(int w) {
       int j, k;
       i = 2*w;
          w = w+1;
       ...
   }
   void main() {
       int a, b;
       h = 5; a = 3; b = 2;
       A(a, b);
       ...
   }
}
```

Parameter Passing: Semantic Models

- Semantic models for formal parameters
  - In mode – can receive data from corresponding actual parameters
    - Actual value is either copied to caller, or an access path is transmitted
  - Out mode – can transmit data to actual parameters
  - Inout mode – can do both receive/transmit data



Parameter Passing

- Pass-by-value
- Pass-by-result
- Pass-by-value-result
- Pass-by-reference
- Pass-by-name

Parameter Passing: Implementation

- Pass by value (in mode)
  - Value of actual parameter is used to initialize formal parameter, which acts as a local variable

```
void foo (int a) {
        a = a + 1;
}
void main() {
      int b = 2;
      foo(b);
 }
```

  - Normally implemented by copying actual parameter to formal parameter
  - Can also be implemented by transmitting access path to the value of actual parameter as long as cell is write protected
  - Disadvantages:
    - Requires more storage (duplicated space)
    - Cost of the moves (if the parameter is large)

Parameter Passing: Implementation

- Pass by result (out mode)
  - Local's value is passed back to the caller
  - No value transmitted to the subprogram
  - Formal parameter acts as local variable, but just before control is transferred back to caller, its value is transmitted to actual parameter
  - Disadvantages:
    1. If value is copied back (as opposed to access paths), need extra time and space
    2. Pass-by-result can create parameter collision

       e.g. procedure sub1(y: int, z: int);

       ...

       sub1(x, x);

       - Value of x in the caller depends on order of assignments at the return

Parameter Passing: Implementation

- Pass by value-result (or pass-by-copy)
  - Combination of pass-by-value and pass-by-result
  - Formal parameter acts as local variable in subprogram
  - Actual parameter is copied to formal parameter at subprogram entry and copied back at subprogram termination
  - Share disadvantages of pass-by-result and pass-by-value
    - Requires multiple storage for parameters
    - Requires time for copying values
    - Problems with parameter collision

Parameter Passing: Implementation

- Pass by reference (or pass-by-sharing)
  - transmits an access path (e.g., address) to the called subprogram
  - Called subprogram is allowed to access actual parameter in the calling program unit
  - Advantage:
    - passing process is efficient (no copying and no duplicated storage)
  - Disadvantages:
    - Slower accesses to formal parameters due to additional level of indirect addressing
    - Allows aliasing

```
void fun (int &first, int &second);
…
fun(total, total);
```

Parameter Passing: Implementation

- Pass-by-reference
  - Collisions due to array elements can also cause aliases

```
void fun(int &first, int &second)
fun(list[i], list[j]);      /* where i=j */
void fun1(int &first, int *a);
fun1(list[i], list);
```

  - Collisions between formal parameters and nonlocal variables that are visible

```
int *global;
void main() {
        extern int *global;
        …
        sub(global);
        …
}
```

```
void sub(int *param) {
        extern int *global;
        …
}
```

Parameter Passing: Implementation

- Pass by Name
  - Another type of inout mode
  - Actual parameter is textually substituted for the corresponding formal parameters
    - Actual binding of value and address is delayed until formal parameter is assigned or referenced
  - Advantage:
    - flexibility of late binding
  - Disadvantage:
    - very expensive related to other parameter passing
      - Not used in any widely used language
  - Another Example:
    - Used at compile time by macros, and for generic subprograms in C++

Pass-by-value

```
int m=8, i=5;
foo(m);
print m;    # prints 8
            # since m is passed by-value
...
proc foo (byval b) {
   b = i + b;
   # b is byval so it is essentially a local variable
   # initialized to 8 (the value of the actual back in
   # the calling environment)
   # the assignment to b cannot change the value of m back
   # in the main program
}
```

Pass-by-reference

```
int m=8, i=5;
foo(m);
print m;    # prints 13
            # since m is passed by-reference
...
proc foo (byref b) {
   b = i + b;
   # b is byref so it is a pointer back to the actual
   # parameter back in the main program (containing 8
  initially)
   # the assignment to b actually changes the value in m
  back
   # in the main program
   # i accesses the variable in the main via scope rules
}
```

Pass-by-value-result

```
int m=8, i=5;
foo(m);
print m;    # prints 13
            # since m is passed by-value-result
...
proc foo (byvres b) {
   b = i + b;
   # b is byves so it copies value of the actual
   # parameter (containing 8 initially)
   # new value of b is copied back to actual parameter
   # in the main program
   # i accesses the variable in the main via scope rules
}
```

Pass-by-name

```
array A [1..100] of int;
int i=5;
foo(A[i],i);
print A[i]; # prints A[6]
...         # so prints 7

# good example
proc foo (name B,name k) {
   k = 6;
   B = 7;
}

# text substitution does this
proc foo {
   i = 6;
   A[i] = 7;
}
```

```
array A [1..100] of int;
int i=5;
foo(A[i]);
print A[i]; # prints A[5]
...         # not sure what

# a problem here...
proc foo (name B) {
   int i = 2;
   B = 7;
}

proc foo {
   int i = 2;
   A[i] = 7;
}
```

Parameter Passing in PL

- Fortran
  - Always use inout-mode semantics model of parameter passing
  - Before Fortran 77, mostly uses pass-by-reference
  - Later implementations mostly use pass-by-value-result
- C
  - mostly pass by value
  - Pass-by-reference is achieved using pointers as parameters

```
int *p = { 1, 2, 3 };
void change( int *q) {
        q[0] = 4;
}
main() {
    change(p);    /*  p[0] = 4 after calling the change function */
}
```

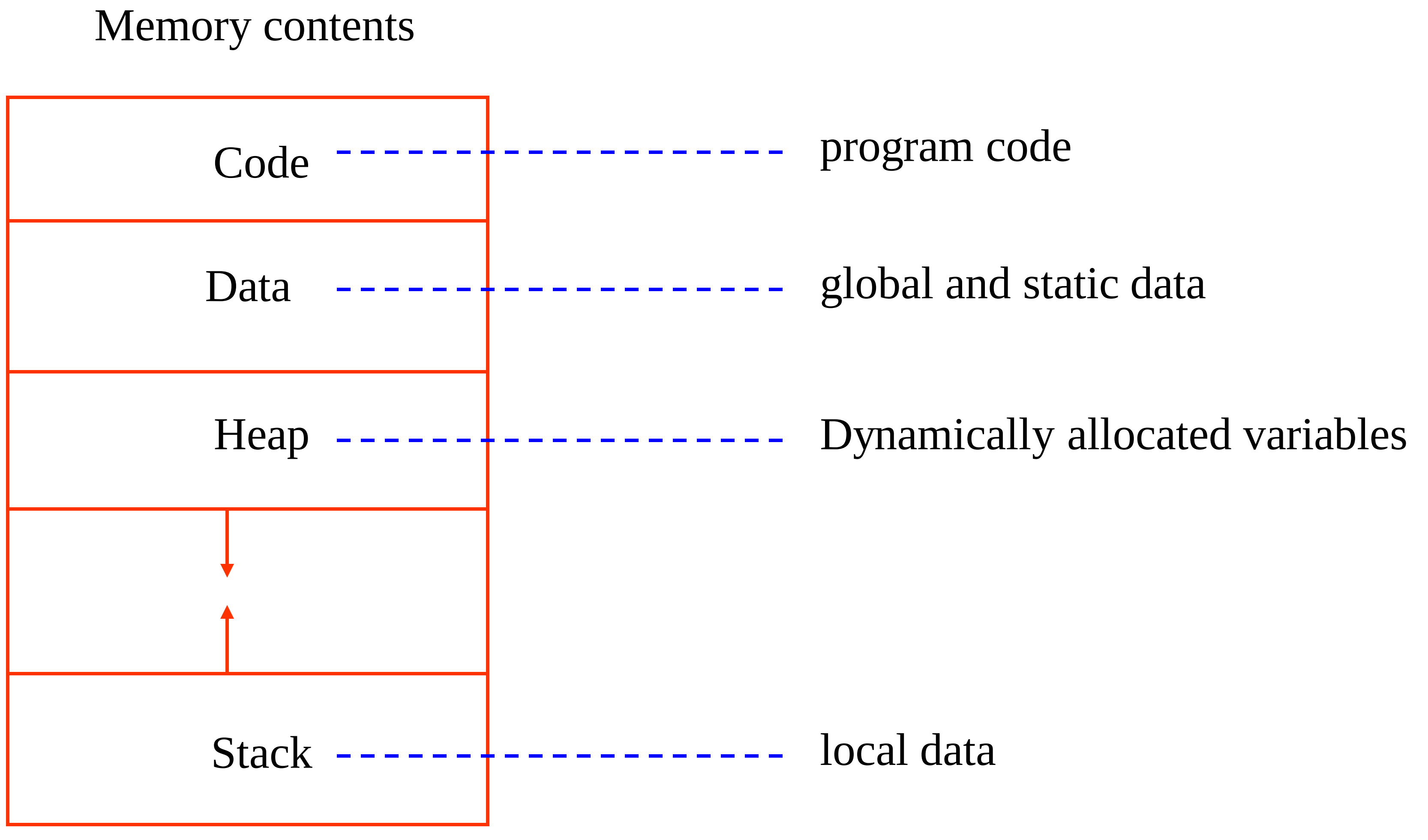
Parameter Passing in PL

- C++
  - includes a special pointer type called a reference type

```
void GetData(double &Num1, const int &Num2) {
    int temp;
    for (int i=0; i<Num2; i++) {
            cout << “Enter a number: “;
            cin >> temp;
            if (temp > Num1)
                { Num1 = temp; return; }
    }
```

  - Num1 and Num2 are passed by reference
  - const modifier prevents a function from changing the values of reference parameters
  - Referenced parameters are implicitly dereferenced
  - Why do we need a constant reference parameter?

Implementing Parameter Passing



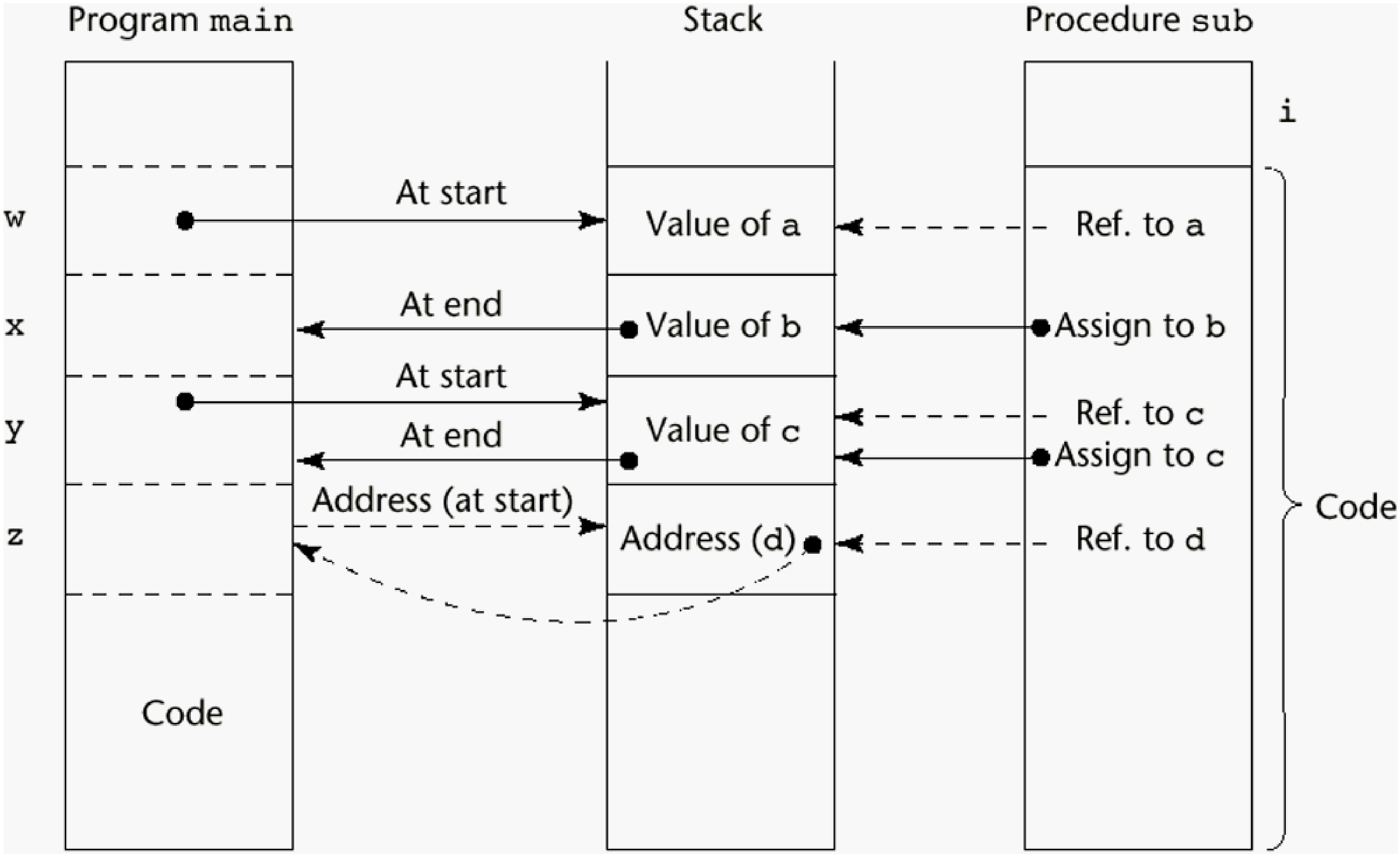
Implementing Parameter Passing

- Pass by Value
  - Values copied into stack locations
  - Stack locations serve as storage for corresponding formal parameters
- Pass by Result
  - Implemented opposite of pass-by-value
  - Values assigned to actual parameters are placed in the stack, where they can be retrieved by calling program unit upon termination of called subprogram
- Pass by Value Result
  - Stack location for parameters is initialized by the call and then copied back to actual parameters upon termination of called subprogram

Implementing Parameter Passing

- Pass by Reference
  - Regardless of type of parameter, put the address in the stack
  - For literals, address of literal is put in the stack
  - For expressions, compiler must build code to evaluate expression before the transfer of control to the called subprogram
    - Address of memory cell in which code places the result of its evaluation is then put in the stack
  - Compiler must make sure to prevent called subprogram from changing parameters that are literals or expressions
  - Access to formal parameters is by indirect addressing from the stack location of the address

Implementing Parameter Passing



Main program calls sub(w,x,y,z) where w is passed by value, x is passed by result, y is passed by value-result, and z is passed by reference

Implementing Parameter Passing

- Pass by Name
  - run-time resident code segments or subprograms evaluate the address of the parameter
  - called for each reference to the formal
  - Very expensive, compared to pass by reference or value-result

Subprogram Names as Parameters

- Issues:
  1. Are parameter types checked?
     - Early Pascal and FORTRAN 77 do not; later versions of Pascal and FORTRAN 90 do
     - Ada does not allow subprogram parameters
     - Java does not allow method names to be passed as parameters
     - C and C++ - pass pointers to functions; parameters can be type checked
  2. What is the correct referencing environment for a subprogram that was sent as a parameter?
     - Environment of the call statement that enacts the passed subprogram
       - Shallow binding
     - Environment of the definition of the passed subprogram
       - Deep binding
     - Environment of the call statement that passed the subprogram as actual parameter
       - Ad hoc binding (Has never been used)

Subprogram Names as Parameters

```
function sub1() {
    var x;
    function sub2() {
        alert(x);   ←
    };
    function sub3() {
        var x;
        x = 3;
        sub4(sub2);
    }
    function sub4(subx) {
        var x;
        x = 4;
        subx();
    };
    x = 1;
    sub3();
};
```

Shallow binding:

- Referencing environment of sub2 is that of sub4

Deep binding

- Referencing environment of sub2 is that of sub1

Ad-hoc binding

- Referencing environment of sub2 is that of sub3

Overloaded Subprograms

- A subprogram that has the same name as another subprogram in the same referencing environment
- Every version of the overloaded subprogram must have a unique protocol
  - Must be different from others in the number, order, or types of its parameters, or its return type (if it is a function)
- C++, Java, Ada, and C# include predefined overloaded subprograms – e.g., overloaded constructors in C++
- Overloaded subprograms with default parameters can lead to ambiguous subprogram calls

```
void foo( float b = 0.0 );
void foo();
…
foo();          /* call is ambiguous; may lead to compilation error */
```

Generic (Polymorphic) Subprograms

- Polymorphism:
  - Increase reusability of software
  - Types:
    - Ad hoc polymorphism = Overloaded subprogram
    - Parametric polymorphism
      - Provided by a subprogram that takes a generic parameter that is used in a type expression
      - Ada and C++ provide compile-time parametric polymorphism

Generic Subprograms

```
template <class Type>
void generic_sort(Type list[], int len) {
        int top, bottom;
        Type temp;
        for (top = 0; top < len - 2; top++)
          for (bottom = top + 1; bottom < len - 1; bottom++) {
            if (list[top] > list[bottom]) {
              temp = list [top];
              list[top] = list[bottom];
              list[bottom] = temp;
             }  //** end of for (bottom ...
}  //** end of generic_sort

float flt_list[100];
...
generic_sort(flt_list, 100); // Implicit instantiation
```

Generic Method in Java

```
public class GenericMethodTest {
  // generic method printArray
  public static < E > void printArray( E[] inputArray ) {
    // Display array elements
    for(E element : inputArray) {
      System.out.printf("%s ", element);
    }
    System.out.println();
  }

  public static void main(String args[]) {
    // Create arrays of Integer, Double and Character
    Integer[] intArray = { 1, 2, 3, 4, 5 };
    Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4 };
    Character[] charArray = { 'H', 'E', 'L', 'L', 'O' };

    System.out.println("Array integerArray contains:");
    printArray(intArray);   // pass an Integer array

    System.out.println("\nArray doubleArray contains:");
    printArray(doubleArray);   // pass a Double array

    System.out.println("\nArray characterArray contains:");
    printArray(charArray);   // pass a Character array
  }
}
```

Generic Class in Java

```
public class Box<T> {
  private T t;

  public void add(T t) {
    this.t = t;
  }

  public T get() {
    return t;
  }

  public static void main(String[] args) {
    Box<Integer> integerBox = new Box<Integer>();
    Box<String> stringBox = new Box<String>();

    integerBox.add(new Integer(10));
    stringBox.add(new String("Hello World"));

    System.out.printf("Integer Value :%d\n\n", integerBox.get());
    System.out.printf("String Value :%s\n", stringBox.get());
  }
}
```