Subprograms

Subprograms

- Programming languages provide two fundamental abstraction mechanisms:

1. **Process Abstraction** → Encapsulating behavior (functions, procedures, methods).
2. **Data Abstraction** → Encapsulating data and its operations (objects, classes, ADTs).

Subprograms

- The first programmable computer, Babbage's Analytical Engine, built in the 1840s, had the capability of reusing collections of instruction cards at several different places in a program.
- In a modern programming language, such a collection of statements is written as a **subprogram**. This reuse results in savings in memory space and coding time.

Subprograms

- A subprogram (also called a procedure, function, method, or routine) is a self-contained sequence of instructions that performs a specific task within a program.

- Subprograms allow code reuse, modularity, and better organization in programming.

General Subprogram Characteristics

- Each subprogram has a single entry point.
- The calling program unit is suspended during the execution of the called subprogram, which implies that there is only one subprogram in execution at any given time.
- Control always returns to the caller when the subprogram execution terminates.

Parameter Profile

- The **parameter profile** of a subprogram contains the number, order, and types of its formal parameters.
- The **protocol** of a subprogram is its parameter profile plus, if it is a function, its return type.
- In languages in which subprograms have types, those types are defined by the subprogram’s protocol.

Parameters

- **Formal parameters** - parameters in the subprogram header.
- **Actual parameters -** Subprogram call statements must include the name of the subprogram and a list of parameters to be bound to the formal parameters of the subprogram.
- **Positional parameters**- The first actual parameter is bound to the first formal parameter and so forth.
- **Keyword parameters** - can appear in any order in the actual parameter list.

  sumer(length = my_length, list = my_array, sum = my_sum)

Procedure and Functions

- Subprograms are collections of statements that define parameterized computations.
- Functions return values and procedures do not.
- Procedures can produce results in the calling program unit by two methods:
  - (1) If there are variables that are not formal parameters but are still visible in both the procedure and the calling program unit, the procedure can change them; and
  - (2) if the procedure has formal parameters that allow the transfer of data to the caller, those parameters can be changed

Function

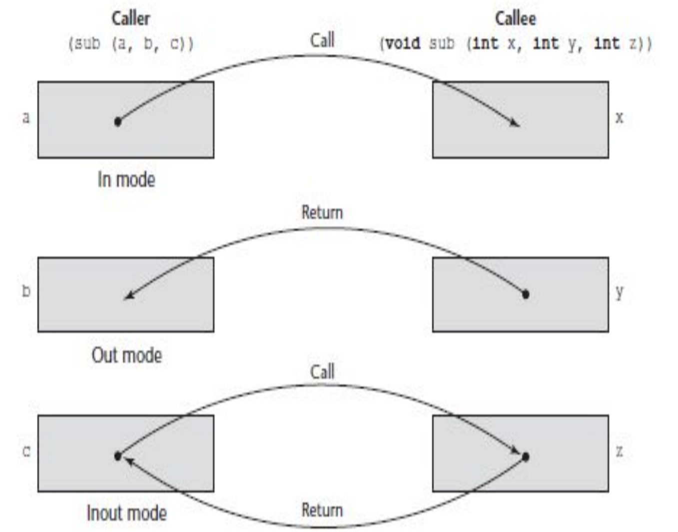
- If a function is a **faithful model**, it produces no side effects; that is, it modifies neither its parameters nor any variables defined outside the function.
- Such a function returns a value—that is its only desired effect.
- For example, the value of the expression f(x) is whatever value f produces when called with the parameter x. For a function that does not produce side effects, the returned value is its only effect.
- The functions in most programming languages have side effects.

Design Issues

- Are local variables statically or dynamically allocated?
- Can subprogram definitions appear in other subprogram definitions?
- What parameter-passing method or methods are used?
- Are the types of the actual parameters checked against the types of the formal parameters?
- If subprograms can be passed as parameters and subprograms can be nested, what is the referencing environment of a passed subprogram?
- Are functional side effects allowed?
- What types of values can be returned from functions?
- How many values can be returned from functions?
- Can subprograms be overloaded?
- Can subprograms be generic?

Parameters Passing

- Formal parameters are characterized by one of three distinct semantics models:
  - (1) They can receive data from the corresponding actual parameter;
  - (2) they can transmit data to the actual parameter; or
  - (3) they can do both.



- These models are called **in mode**, **out mode**, and **inout mode**, respectively

Parameters Passing

- Pass-by-Value – Adv: fast Dis: Additional storage.
- Pass-by-Result – same as above.
- Pass-by-Value-Result (sometimes called pass-by-copy)
- Pass-by-Reference – Adv: Duplicate space is not required Dis: Indirect addressing, Erroneous changes may be made.
- Pass-by-name - complex

Pass by result

```
procedure compute_square(x : integer, result : integer RESULT)
begin
    result := x * x;   // Assign a value to 'result'
end;

a = 5;
b = 0;    // 'b' is ignored as it's pass-by-result

compute_square(a, b);

print(b);   // Output: 25
```

Example

```
#include <stdio.h>

int a=2;
int b=1;

void fun(int x, int y) {
    b=x+y;
    x=a+y;
    y=b+x;
}

void main() {
    fun(b,b);
    printf("%d %d\n",a,b);
}
```

Example

```
void main()
{
    int x = 5;
    foo (x,x);
    print (x);
}

void foo (int a, int b)
{
    a = 2 * b + 1;
    b = a - 1;
    a = 3 * a - b;
}
```

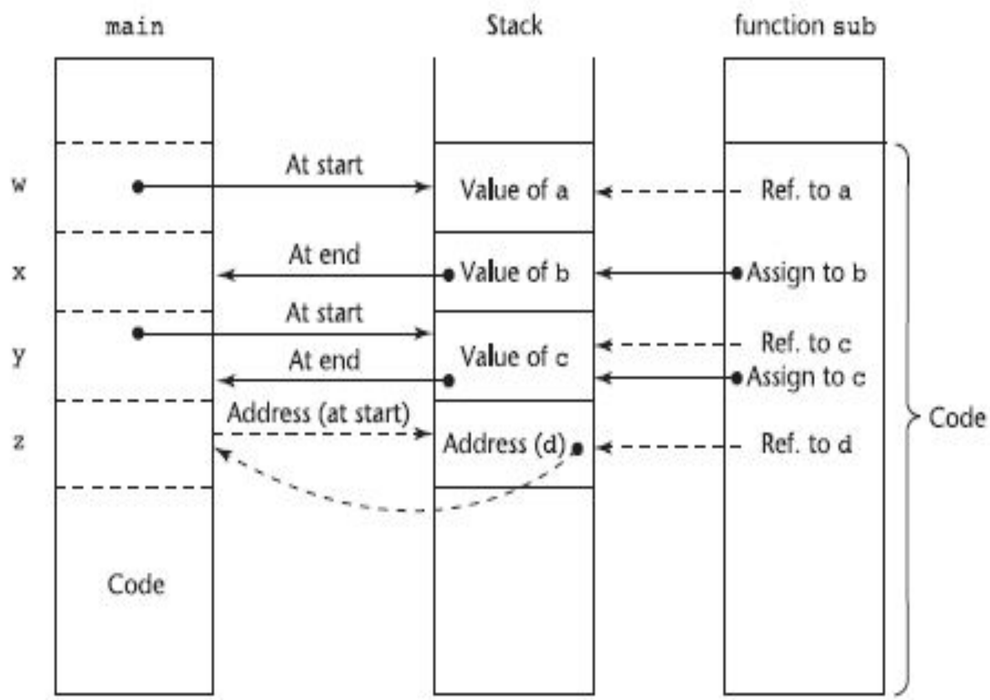
Pass by name

```
#include <stdio.h>
void swap(int x, int y);
int main() {
    // Write C code here
    int a=1, b=3;
    swap (a,a+1);
    printf("a %d b %d\n",a,b);

    return 0;
}

void swap(int x, int y)
{
    int temp;
    temp =x;
    x=y;
    y=temp;
    printf("x %d y %d\n",x,y);
}
```

Implementing Parameter-Passing Methods



Parameters That Are Subprograms

- The environment of the call statement that enacts the passed subprogram (**shallow binding**)
  - Sub2 from sub4 => 4
- The environment of the definition of the passed subprogram (**deep binding**)
  - Sub2 from sub1 => 1
- The environment of the call statement that passed the subprogram as an actual parameter (**ad hoc binding**)
  - Sub2 from sub3 => 3

Note: Consider the execution of sub2 when it is called in sub4.

```
function sub1() {
  var x;
  function sub2() {
    alert(x);  // Creates a dialog box with the value of x
    };
  function  sub3() {
    var  x;
    x = 3;
    sub4(sub2);
    };
  function  sub4(subx) {
    var  x;
    x = 4;
    subx();
    };
  x = 1;
  sub3();
  };
```

Parameters That Are Subprograms

```
function sub1() {
  var x;
  function sub2() {
   alert(x); // Creates a dialog box with the value of x
  };
  function sub3() {
   var x;
   x = 3;
   sub4(sub2);
  };
  function sub4(subx) {
    var x;
    x = 4;
   subx();
  };
x = 1;
sub3();
};
```

Consider the execution of `sub2` when it is called in `sub4`. For shallow binding, the referencing environment of that execution is that of `sub4`, so the reference to `x` in `sub2` is bound to the local `x` in `sub4`, and the output of the program is `4`. For deep binding, the referencing environment of `sub2`'s execution is that of `sub1`, so the reference to `x` in `sub2` is bound to the local `x` in `sub1`, and the output is `1`. For ad hoc binding, the binding is to the local `x` in `sub3`, and the output is `3`.

Calling Subprograms Indirectly

- The call to the subprogram is made through a pointer or reference to the subprogram, which has been set during execution before the call is made.

- The two most common applications of indirect subprogram calls are
  - for event handling in graphical user interfaces, which are now part of nearly all Web applications, as well as many non-Web applications, and
  - for callbacks, in which a subprogram is called and instructed to notify the caller when the called subprogram has completed its work.

Calling Subprograms Indirectly

int myfun2 (int, int); // A function declaration

int (*pfun2)(int, int) = myfun2; // Create a pointer and initialize it to point to myfun2

pfun2 = myfun2; // Assigning a function's address to a pointer

Calling Subprograms Indirectly

```
#include <stdio.h>

// Function declaration
int myfun2(int a, int b) {
    return a + b; // Example function: returns sum of two integers
}

int main() {
    // Declare function pointer and assign myfun2's address
    int (*pfun2)(int, int) = myfun2;

    // Call the function using the function pointer
    int result = pfun2(5, 10);
    int resultx = (*pfun2)(5, 9);

    // Print the result
    printf("Result of myfun2(5, 10): %d\n", result);
    printf("Result of myfun2(5, 10): %d\n", resultx);

    return 0;
}
```

Closure

```
outer = function() {
  var a = 1;
  var inner = function()
  {
    console.log(a);
  }
  return inner; // this returns a function
}
var fnc = outer(); // execute outer to get inner
fnc();
```

Normally when a function exits, all its local variables are blown away. However, if we return the inner function and assign it to a variable *fnc* so that it persists after outer has exited, all of the variables that were in scope when inner was defined also persist. The variable a has been closed over -- it is within a closure.

Closure in Python

- In Python, a closure is typically a function defined inside another function.
- This inner function grabs the objects defined in its enclosing scope and associates them with the inner function object itself.
- The resulting combination is called a closure.

Cont..

```python
def fun(a):
    # Outer function that remembers the value of 'a'
    def adder(b):
        # Inner function that adds 'b' to 'a'
        return a + b
    return adder  # Returns the closure

# Create a closure that adds 10 to any number
val = fun(10)

# Use the closure
print(val(5))
print(val(20))
```

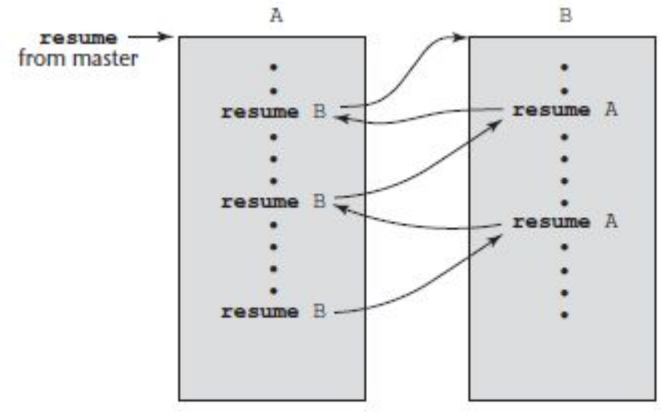
Coroutines

- **A coroutine** is a special kind of subprogram. A **coroutine** is a **special type of function** that can be **paused and resumed** at specific points during execution. Unlike regular functions that execute from start to finish, coroutines can **yield control** to another coroutine and later resume execution **from where they left off**.

- Rather than the master-slave relationship between a caller and a called subprogram that exists with conventional subprograms, caller and called coroutines are more equitable.

- In fact, the coroutine control mechanism is often called the **symmetric unit control model**.

- Coroutines can have multiple entry points, which are controlled by the coroutines themselves.



(a)

```
# A Python program to generate numbers in a
# range using yield

def rangeN(a, b):
    i = a
    while (i < b):
        yield i
        i += 1 # Next execution resumes
               # from this point
for i in rangeN(1, 5):
        print(i)
```

Activation Record

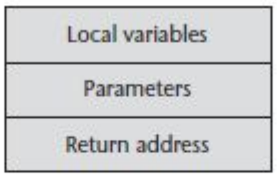
- An activation record is a contiguous block of storage that manages information required by a single execution of a procedure.
- When you enter a procedure, you allocate an activation record, and when you exit that procedure, you de-allocate it.
- Basically, it stores the status of the current activation function. So, whenever a function call occurs, then a new activation record is created and it will be pushed onto the top of the stack. It will remain in stack till the execution of that function.
- So, once the procedure is completed and it is returned to the calling function, this activation function will be popped out of the stack.
- If a procedure is called, an activation record is pushed into the stack, and it is popped when the control returns to the calling function.

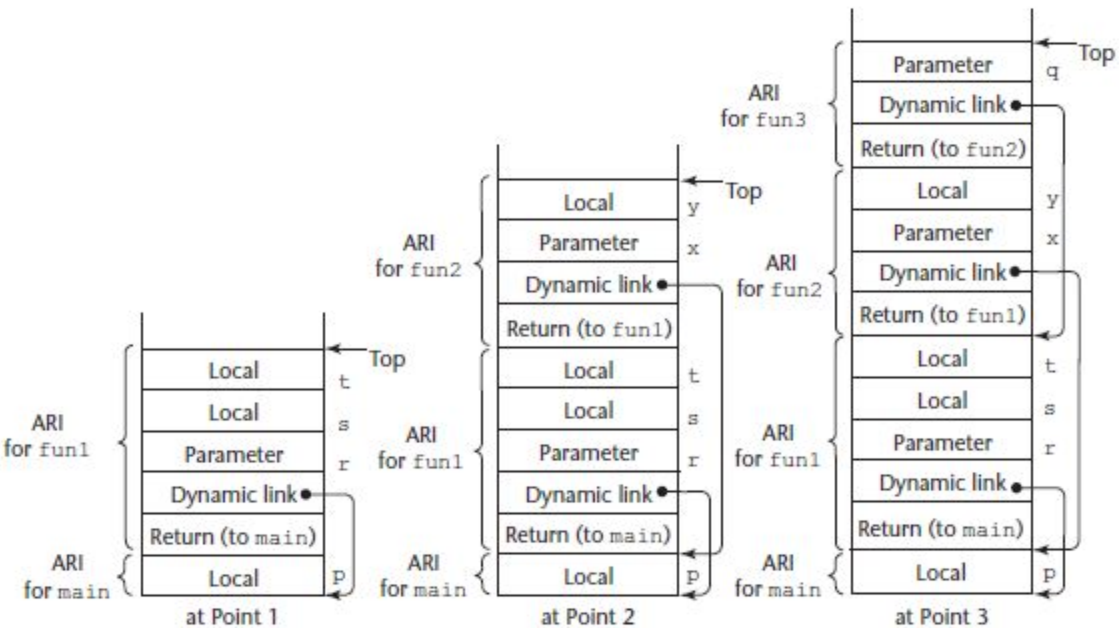
Cont..

- Activation Record includes some fields which are –
  Return values, parameter list, control links, access links, saved machine status, local data, and temporaries.

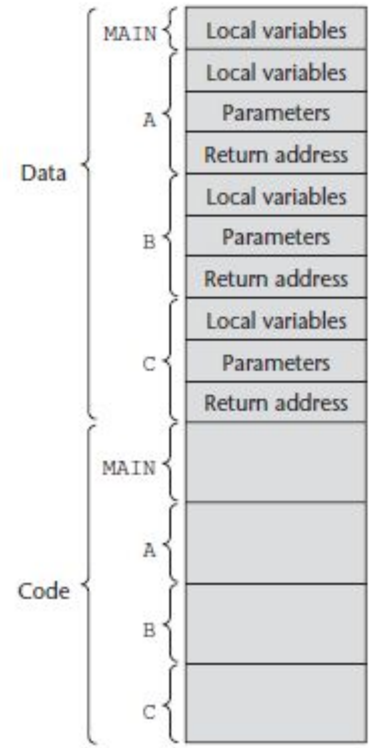
Activation Record

Activation Record





ARI = activation record instance



Multidimensional Arrays as Parameters

```
void fun(int matrix[][10]) { . . . }
void main() {
    int mat[5][10];
    . . .
    fun(mat);
    . . .
}
```

- For row as well as column

```
void fun(float *mat_ptr, int num_rows, int num_cols)
```