Subroutines: abstraction and its implementation

3.2.1-3.2.2, 9.1,9.2, 9.3

Subroutine

- An abstraction
- Parameter passing
- Implementation of subroutine on a physical machine

Subroutine: abstraction

- Subroutine is a natural abstraction we use everyday
- Example – Beef steak recipe
  - **Ingredients**
    - 1 small onion, chopped / 7 cloves garlic / 1/2 cup olive oil / 1/2 cup vinegar / 1/2 cup soy sauce / 2 tablespoons chopped fresh rosemary / 2 tablespoons Dijon-style prepared mustard / 2 teaspoons salt / 1 teaspoon black pepper / 1 (2 pound) tri-tip steak
  - **Directions**
    - Place onion, garlic, olive oil, vinegar, soy sauce, rosemary, mustard, salt, and pepper into the bowl of a food processor. Process until smooth. Place steak in a large resealable plastic bag. Pour marinade over steaks, seal, and refrigerate for about 3 hours.
    - Preheat the grill for high heat.
    - Brush grill grate with oil. Discard marinade, and place steak on the prepared grill. Cook for 7 minutes per side, or to desired doneness.

- Example – collect mail
  - Tell the clerk you come to pick you your mail
  - (The clerk will get the mail and pass it to you)
- In these examples you abstract away the details from “high level of actions”. At the abstract level, you only care about what the actions do.

- When to solve computational problems, we need a lot of such abstractions to
  - help us organize our thoughts and program.
  - Reduce the size of the program.

- Subroutine is such an abstraction
- You have subroutines in almost every program.

Subroutine

- Subroutine
  - Has name, parameters (and possibly return value)
    - A subroutine that returns a value is called a *function.*
    - A subroutine that does not return a value is called *a procedure.*
  - Call (use) a subroutine
    - parameters in the call are *actual parameters.*
  - Define a subroutine
    - Parameters in the definition are *formal parameters.*

Use/define subroutine

- Example: square of a number

Parameter passing

- A parameter can be declared as
  - Pass-by-value
    - Value of the actual parameter is copied to formal parameter, when calling a function
  - Pass-by-result
    - Value of the formal parameter is copied back to the actual parameter after the function returns
  - Pass-by-value-result
    - Value of actual parameter -> formal parameter, when calling, **and** value of formal parameter -> actural parameter after the function returns
  - Pass-by-reference
    - The actual parameter and the formal one have the same reference
  - Pass-by-name
    - Every time a formal parameter is used, the actual parameter will be re-evaluated

Pass by value vs reference vs value-result

- Example

```
int m=9, i=5;
foo(m);
print(m);

proc foo(int b) {
  b := b+5
}
```

Pass by name

```
array A[1..100] of int;
int i=5;
foo(A[i], i);
print A[i];
#print A[6] whose value is 7
#GOOD example
proc foo(name B, name k){
  k=6;
  B=7;
}
#text sub does this
proc foo{
  i=6;
  A[i]=7;
}
```

Return values for functions

- One way to use the function name

```
function  add_five ( value1 : integer ) : integer;
begin
    add_five := value1 + 5;
end;
```

- Another way: using **return**

Discussion

- When the parameter is a big “object”
- Variable length of parameters
  - prinf(“%s%s%s”, x, y, z)
- A function can return a function in C.

Implementation

- How to implement subroutines on a physical machine?
  - Call a subroutine
    - Control should be transferred to the subroutine
    - Parameters should be passed to the subroutine
    - After the subroutine is done, the control should be returned
  - Define a subroutine
    - Depending on how “call a subroutine” is processed

- Example

```
int m=9, i=5;
foo(m, 5);
print(m);

proc foo(int m, int n) {
  int x[12];
       m := m+5;
       n := m + n;
}
```

- Implementation of call/def subroutine
  - Major difficulty is the binding of local variables
  - If no recursion is allowed
    - Use static memory is alright
  - Otherwise
    - Use the stack memory

- Implementation details on call subroutine – calling sequence
  - Push a new frame onto the stack. The frame contains
    - The values of the actual parameter
    - Save the frame pointer (fp)
    - bookkeeping information such as return address
  - “Jump” to the address of the subroutine
  - After the subroutine “returns”, restore fp and pop out the frame
  - More details on stack layout can be found in Fig 9.2 of book. However, the calling sequence here is different from the one in the book.

- Implementation details of defining subroutine
  - Using fp to access the parameters and local variables
  - After the subroutine is finished, “jump” to the return address stored in the current frame of the stack

Example

- Implementation of the following code including call sequence and subroutine definition.

```
int m=9, i=5;
foo(m);
print(m);

proc foo(int b) {
       b := b+5
}
```

Summary

- Subroutine is a way of *abstraction* which helps solve complex problems
- Subroutine can be elegantly implemented on a physical machine by using the stack memory.