# Recursion

Lecture 09

Instructor: Dr. Cong Pu, Ph.D.

*cong.pu@okstate.edu*

*Adapted partially from Data Structures and Algorithms in Java, M.T. Goodrich, R.Tamassia and M. H. Goldwasser, Sixth Edition, Wiley; Data Structures and Algorithms in C++, Adam Drozdek, 4th Edition, Cengage Learning*

# Recursive Definitions

- Two parts of a ***recursive definition***:
  - ***anchor*** or ***ground case*** (also sometimes called the ***base case***)
    - establish the basis for all the other elements of the set
  - ***inductive clause***
    - establish rules for the creation of new elements in the set
- For example, define the set of natural numbers:
  1. $0 \in \mathbf{N}$ (**anchor**)
  2. if $n \in \mathbf{N}$, then $(n + 1) \in \mathbf{N}$ (**inductive clause**)
  3. there are no other objects in the set **N**
  - there may be other definitions

# Recursive Definitions (cont.)

- Recursive definitions serve two purposes:
  - *generating* new elements
  - *testing* whether an element belongs to a set
- In the case of ***testing***
  - ***reducing*** the problem to an even *simpler* problem
  - and so on
  - until it is reduced to the ***anchor*** problem

(you already have solution for anchor problem!!)

- E.g., is 23 a natural number?
  - 1 + 22, 1 + 1 + 21, 1 + 1 + 1 + 20, …

# Recursive Definitions (cont.)

- The *recursive definition* of the factorial function, !:

$$n! = \begin{cases} 1 & \text{if } n = 0 \quad \text{(anchor)} \\ n \cdot (n-1)! & \text{if } n > 0 \quad \text{(inductive clause)} \end{cases}$$

  - So, **3!** = 3 · **2!** = 3 · 2 · **1!** = 3 · 2 · 1 · **0!** = 3 · 2 · 1 · 1 = 6

undesirable feature: to determine the value of current element ($s_n$), we have to compute the values of all of the previous elements ($s_1, \ldots, s_{n-1}$,)

- Find a formula that is equivalent to the recursive one without referring to previous values
  - for factorials, we can use $n! = \prod_{i=1}^{n} i$
  - frequently non-trivial and often quite difficult to achieve

# Recursive Definitions (cont.)

- From the standpoint of *computer science*,
  - ***recursion*** occurs frequently in *language definitions* as well as *programming*
- The translation from specification to code is fairly straightforward;
  - e.g., a factorial function in C++:

```
unsigned int factorial (unsigned int n){
   if (n == 0)
      return 1;
   else return n * factorial (n – 1);
}
```

- Most modern programming languages incorporate mechanisms
  - support the use of recursion, making it transparent to the user
  - recursion on computers are implemented using the **run-time stack**

# Function Calls and Recursive Implementation

- What kind of information must we keep track of when a function is called?
  - if the function has **parameters**??
    - need to be initialized to their corresponding arguments
  - where to resume the calling function once the called function is complete
    - **return address**
  - since functions can be called from other functions,
    - keep track of **local variables** for scope purposes
  - don't know in advance how many calls will occur,
    - **stack**, an efficient location to save information
    - e.g., dynamic allocation using the run-time stack

# Function Calls and Recursive Implementation

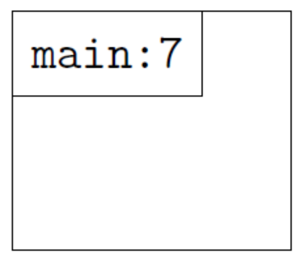
```
def f(x,y):
    x += y
    print x
    return x

def main():
    n = 4
    out = f(n,2)
    print out

main()
```

at the beginning, main is call

- create a new *stack frame*
- main has no parameters
    - *stack frame* is *empty*

# Function Calls and Recursive Implementation

```
def f(x,y):
    x += y
    print x
    return x

def main():
    n = 4
    out = f(n,2)
    print out

main()
```

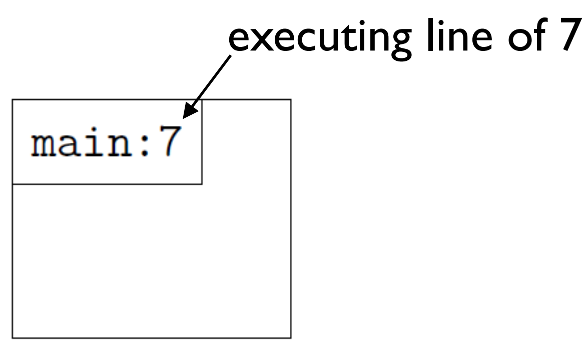



at the beginning, main is call

- create a new *stack frame*
- main has no parameters
  - *stack frame* is *empty*

# Function Calls and Recursive Implementation

```
1    def f(x,y):
2        x += y
3        print x
4        return x
5
6    def main():
7        n = 4
8        out = f(n,2)
9        print out
10
11    main()
```

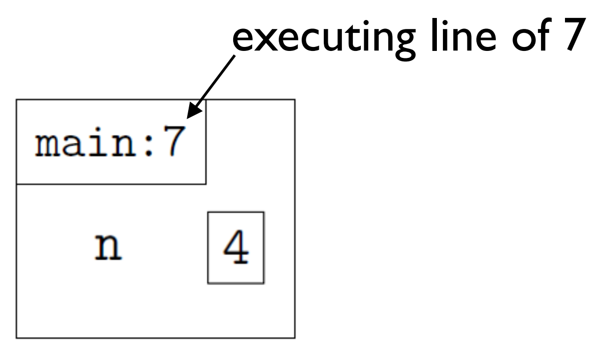



when line of 7 of main is executed

- n is set to 4
- draw a box with a label and put content

# Function Calls and Recursive Implementation

```
1     def f(x,y):
2         x += y
3         print x
4         return x
5
6     def main():
7         n = 4
8         out = f(n,2)
9         print out
10
11     main()
```

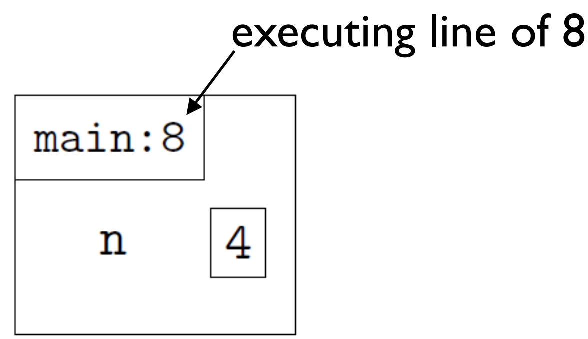



when line of 7 of main is executed

- n is set to 4
- draw a box with a label and put content

# Function Calls and Recursive Implementation

```
1    def f(x,y):
2        x += y
3        print x
4        return x
5
6    def main():
7        n = 4
8        out = f(n,2)
9        print out
10
11    main()
```

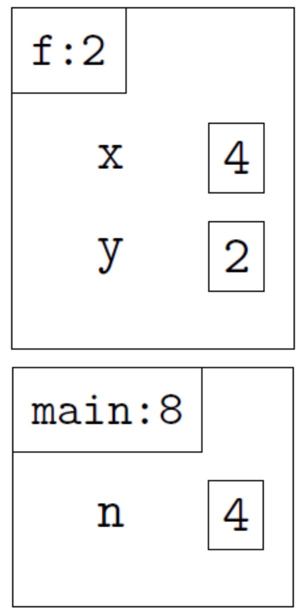



when line of 8 of main is executed

- f is called
    - first determine the value of arg, n
        - n is 4; (2nd arg is 2)
- create a new *stack frame* containing arg values

# Function Calls and Recursive Implementation

```
1     def f(x,y):
2         x += y
3         print x
4         return x
5
6     def main():
7         n = 4
8         out = f(n,2)
9         print out
10
11     main()
```

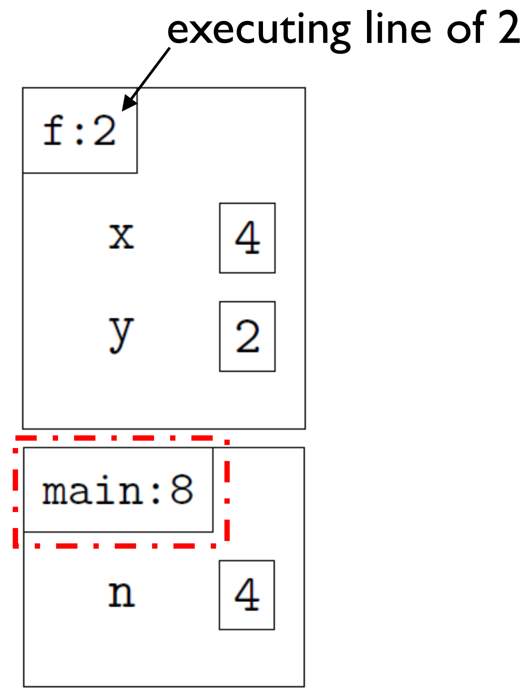



when line of 8 of main is executed

- f is called
  - first determine the value of arg, n
    - n is 4; (2nd arg is 2)
- create a new *stack frame* containing arg values

# Function Calls and Recursive Implementation

```
1     def f(x,y):
2         x += y
3         print x
4         return x
5
6     def main():
7         n = 4
8         out = f(n,2)
9         print out
10
11     main()
```

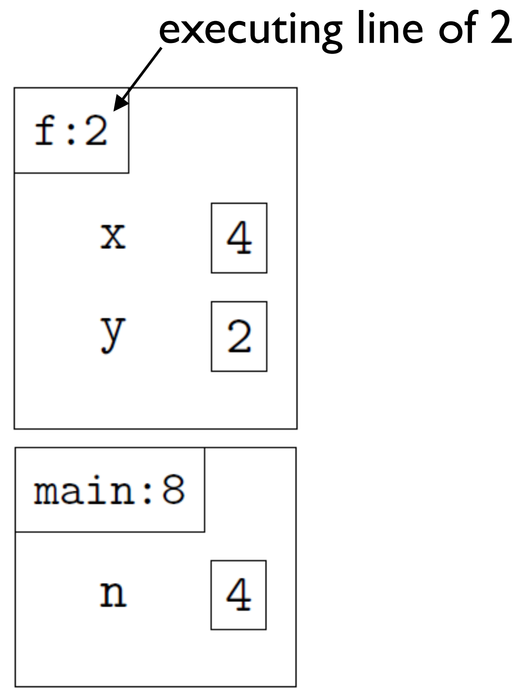



***Note***:

- the stack frame for main is keeping track of where we were in that function
  - when f is done, we will return to that line

# Function Calls and Recursive Implementation

```
1     def f(x,y):
2         x += y
3         print x
4         return x
5
6     def main():
7         n = 4
8         out = f(n,2)
9         print out
10
11     main()
```

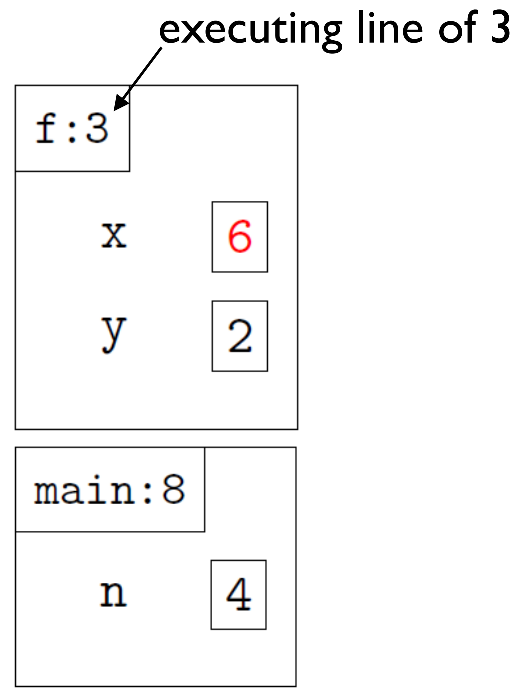



when line of 2 is executed

- update x

# Function Calls and Recursive Implementation

```
1     def f(x,y):
2         x += y
3         print x
4         return x
5
6     def main():
7         n = 4
8         out = f(n,2)
9         print out
10
11     main()
```

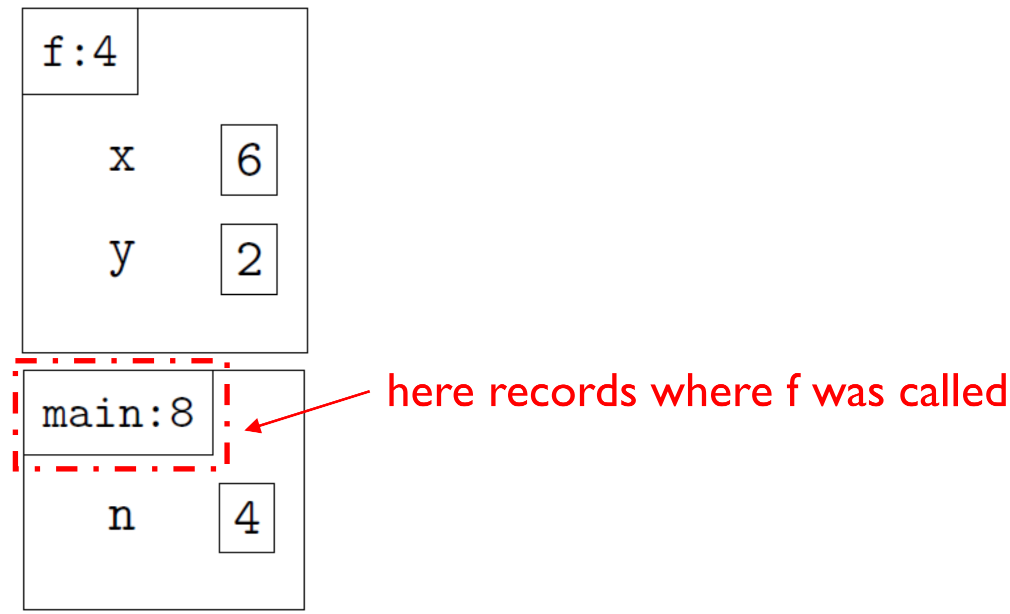



when line of 3 is executed

- print x

# Function Calls and Recursive Implementation

```
1     def f(x,y):
2         x += y
3         print x
4         return x
5
6     def main():
7         n = 4
8         out = f(n,2)
9         print out
10
11     main()
```

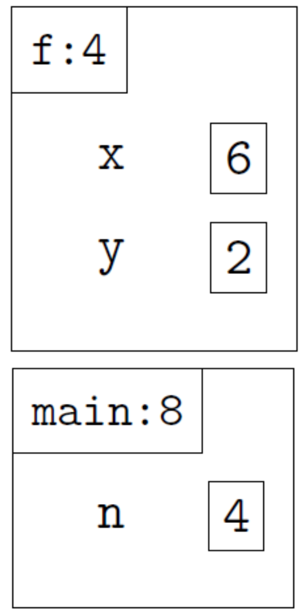



when line of 4 is executed

- return x to the place (line of 8) where f was called
  - out has value 4

# Function Calls and Recursive Implementation

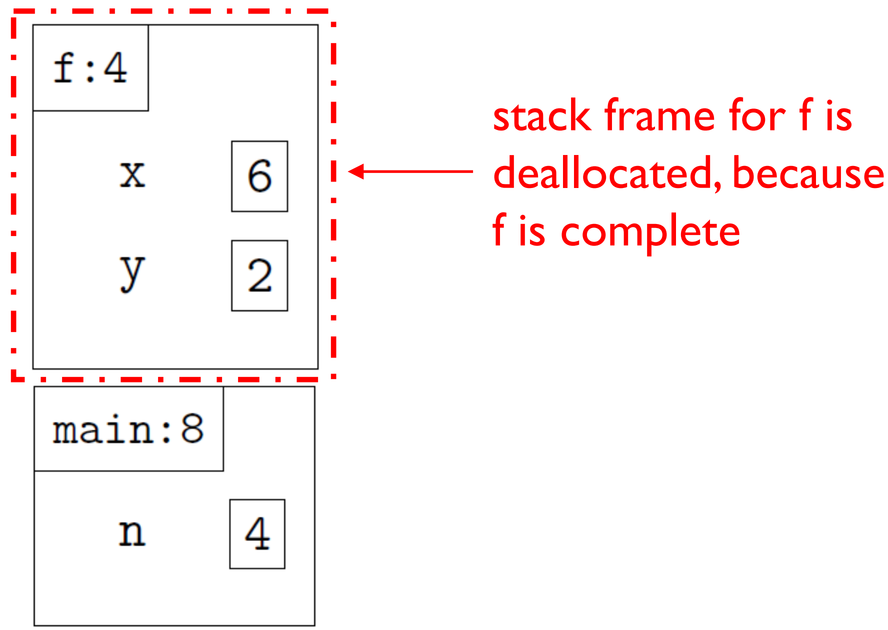
```
1     def f(x,y):
2         x += y
3         print x
4         return x
5
6     def main():
7         n = 4
8         out = f(n,2)
9         print out
10
11     main()
```




line of 8 is where f was called, so this is the place where f is supposed to be returned

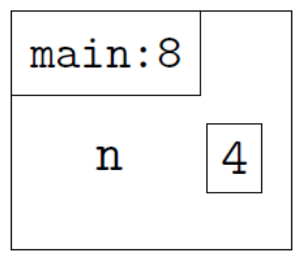
# Function Calls and Recursive Implementation

```
def f(x,y):
    x += y
    print x
    return x

def main():
    n = 4
    out = f(n,2)
    print out

main()
```



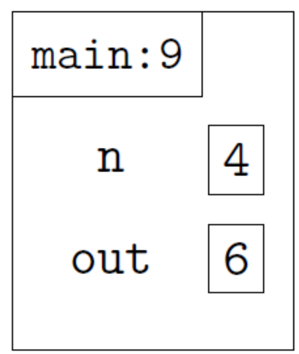
# Function Calls and Recursive Implementation

```
1    def f(x,y):
2        x += y
3        print x
4        return x
5
6    def main():
7        n = 4
8        out = f(n,2)
9        print out
10
11    main()
```



# Function Calls and Recursive Implementation

```
1     def f(x,y):
2         x += y
3         print x
4         return x
5
6     def main():
7         n = 4
8         out = f(n,2)
9         print out
10
11     main()
```

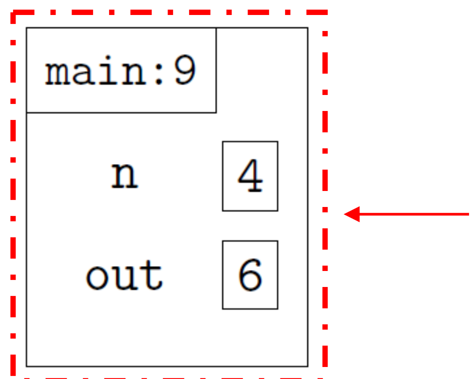



when line of 9 is executed

- print out

# Function Calls and Recursive Implementation

```
1     def f(x,y):
2         x += y
3         print x
4         return x
5
6     def main():
7         n = 4
8         out = f(n,2)
9         print out
10
11     main()
```




stack frame for main is deallocated, because main is complete

after executing line of 9

- main is complete; the program is finished

# Function Calls and Recursive Implementation

```
def f(x,y):
    x += y
    print x
    return x

def main():
    n = 4
    out = f(n,2)
    print out

 main()
```

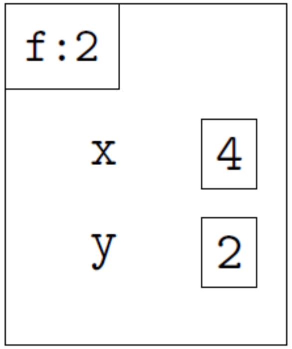
# Function Calls and Recursive Implementation (cont.)

- Characterize the state of a function by a set of information
  - an ***activation record*** or ***stack frame***
- Every time a function is called,
  - its ***activation record*** is created and placed on the run-time stack
  - an ***activation record*** exists for as long as a function owning it is executing
    - *private pool* of info. for that function
      - storing all info. necessary for function's operation and how to return to where it was called from
    - *short life span*
      - dynamically allocated at function entry
      - dynamically deallocated upon exiting

# Function Calls and Recursive Implementation (cont.)

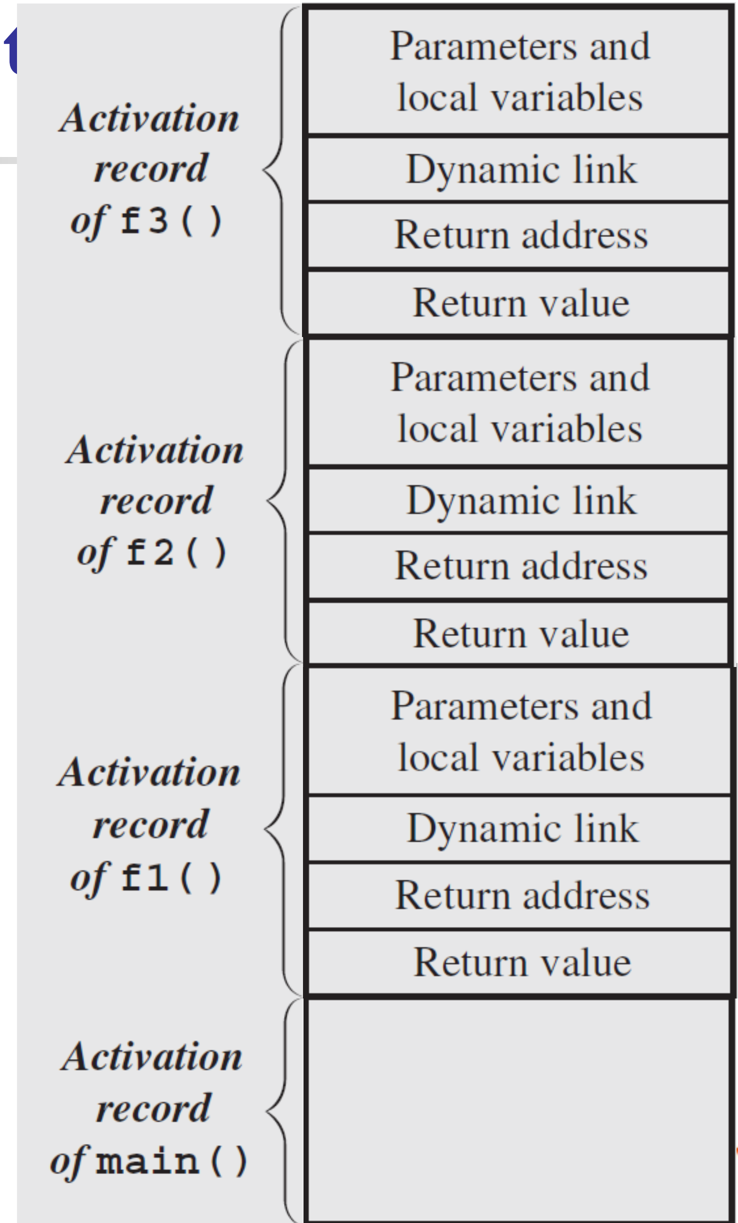
- The following information stored on the run-time stack:
  - values of the function's parameters, addresses of reference variables (including arrays)
  - copies of local variables
  - the return address of the calling function
  - a dynamic link to the calling function's activation record
  - the function's return value if it is not void

```
def f(x,y):
    x += y
    print x
    return x
```



# Function Calls and Recursive Implementation (cont

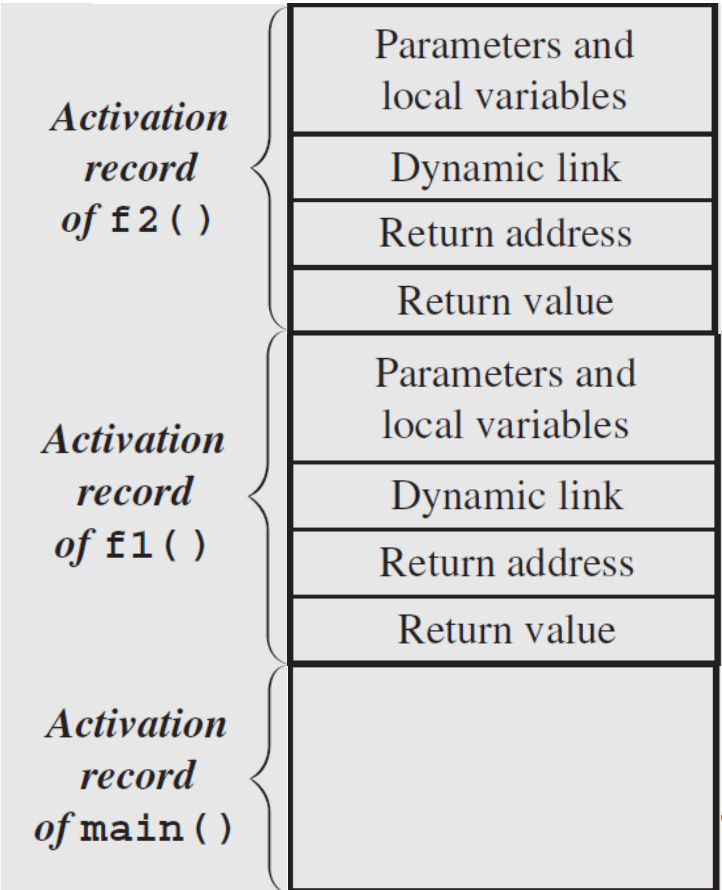
- A snapshot of run-time stack:
  - always contain the **current state** of the function
- e.g., main() → f1() → f2() → f3()



# Function Calls and Recursive Implementation (cont.)

- A snapshot of run-time stack:
  - always contain the **current state** of the function
- e.g., main() → f1() → f2() → f3()

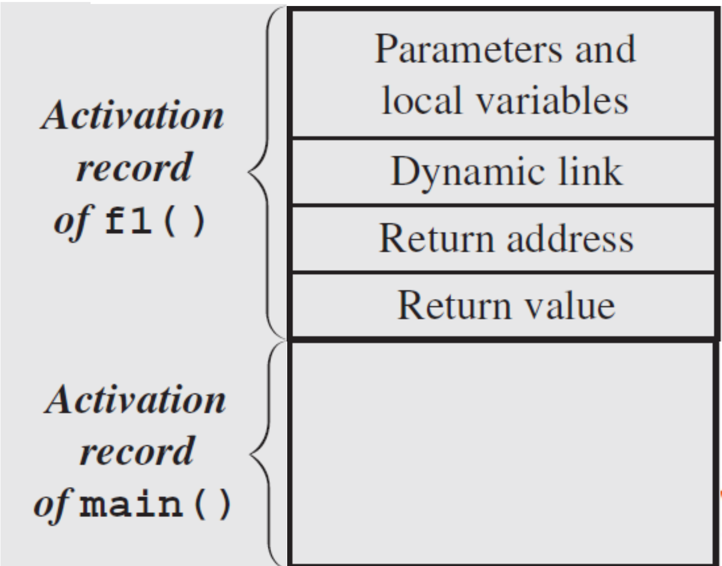
- Once f3() completes,
  - its record is **popped**
  - f2() can **resume**



# Function Calls and Recursive Implementation (cont.)

- A snapshot of run-time stack:
  - always contain the **current state** of the function
- e.g., main() → f1() → f2() → f3()

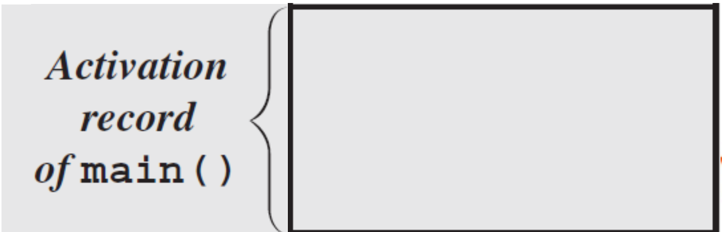
- Once f2() completes,
  - its record is **popped**
  - f1() can **resume**



# Function Calls and Recursive Implementation (cont.)

- A snapshot of run-time stack:
  - always contain the **current state** of the function
- e.g., main() → f1() → f2() → f3()

- Once f1() completes,
  - its record is **popped**
  - main() can **resume**



# Function Calls and Recursive Implementation (cont.)

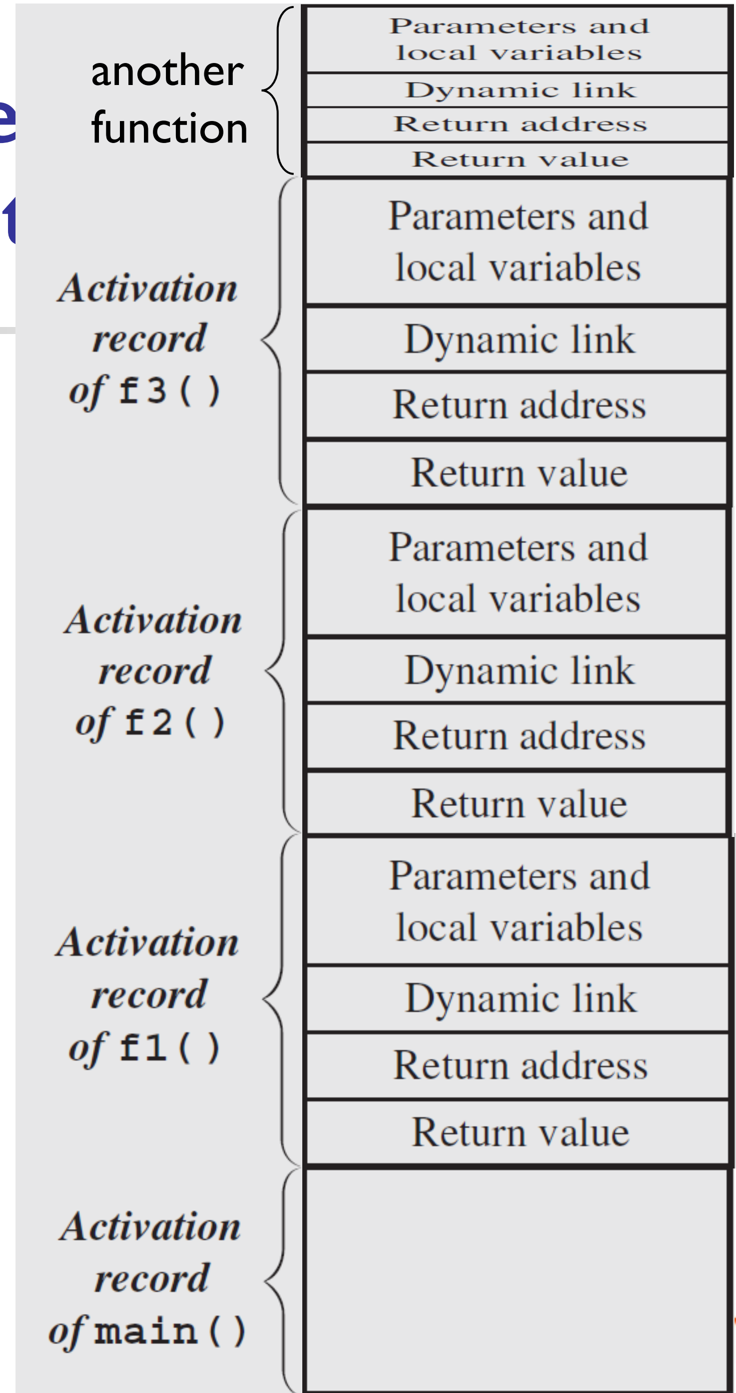
- A snapshot of run-time stack:
  - always contain the **current state** of the function
- e.g., main() → f1() → f2() → f3()

- Once main() completes,
  - its record is **popped**
  - program is finished

# Function Calls and Re Implementation (cont

- A snapshot of run-time stack:
  - always contain the **current state** of the function
- e.g., main() → f1() → f2() → f3()

- Once f3() completes,
  - its record is **popped**
  - f2() can **resume**
- If f3() calls another function,
  - the new function has its activation record **pushed** onto the stack
  - f3() is **suspended**



# Function Calls and Recursive Implementation (cont.)

- The use of ***activation records*** on the run-time stack
  - allow **recursion** to be implemented and handled correctly
- When *a function calls itself recursively*,
  - **push** a *new* activation record of *itself* on the stack
  - **suspend** the calling instance of the function
  - **allow** the new activation to carry on the process

- A recursive call
  - create a series of activation records for different instances of the **same** function