Recursion

Instructor: Dr. Sunho Lim (Ph.D., Assistant Professor)

Lecture 06

sunho.lim@ttu.edu

Adapted partially from Data Structures and Algorithms in C++, Adam Drozdek, 4th Edition, Cengage Learning; and Algorithms and Data Structures, Douglas Wilhelm Harder, Mmath

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.)

- The recursive definition of the factorial function:

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

  - So 3! = 3 · 2! = 3 · 2 · 1! = 3 · 2 · 1 · 0! = 3 · 2 · 1 · 1 = 6
- Find a formula that is equivalent to the recursive one without referring to previous values

$$n! = \prod_{i=1}^{n} i$$

  - for factorials, we can use
  - 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
  - use the **runtime stack**

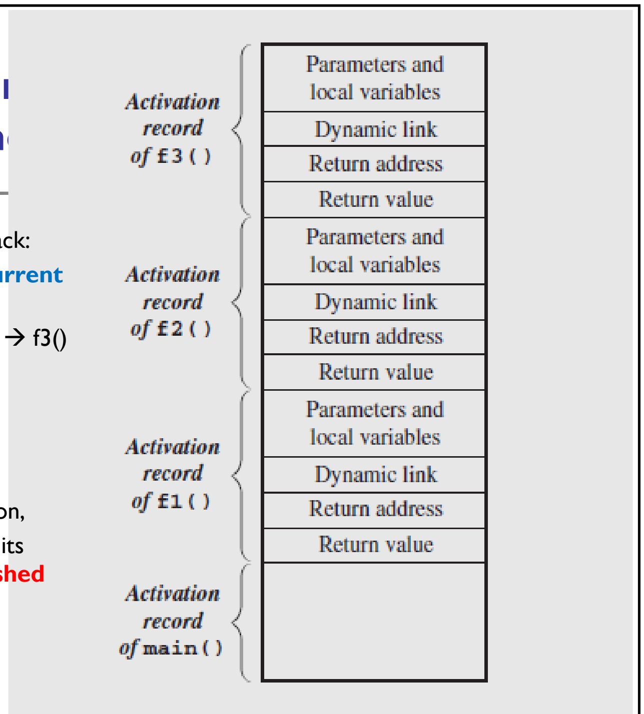
Function Calls and Recursive Implementation

- 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 (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 runtime stack
- The following information stored on the runtime 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

Functio
Implem



- A snapshot of runtime 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 runtime 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

Anatomy of a Recursive Call

- Analyze the recursive function and its behavior of recursion
  - e.g., a number x to a non-negative integer power n:

$$x^n = \begin{cases} 1 & \text{if } n = 0 \\ x \cdot x^{n-1} & \text{if } n > 0 \end{cases}$$

```
/* 102 */  double power (double x, unsigned int n) {
/* 103 */       if (n == 0)
/* 104 */          return 1.0;
                //else
/* 105 */       return x * power(x,n-1);
           }
```

- e.g., the calculation of x^4,
  - $x^4 = x \cdot x^3 = x \cdot (x \cdot x^2) = x \cdot (x \cdot (x \cdot x^1)) = x \cdot (x \cdot (x \cdot (x \cdot x^0))) = x \cdot (x \cdot (x \cdot (x \cdot 1))) = x \cdot (x \cdot (x \cdot (x))) = x \cdot (x \cdot (x \cdot x)) = x \cdot (x \cdot x \cdot x) = x \cdot x \cdot x \cdot x$
  - how repeated application of the **inductive step** leads to the **anchor**

Anatomy of a Recursive Call (cont.)

- Produce the result of x^0, which is 1,
  - return this value to the previous call
- That call, which had been suspended,
  - resume to calculate $x \cdot 1$, producing x
- Each succeeding return then takes the previous result
  - use it in turn to produce the final result

The sequence of recursive calls and returns,

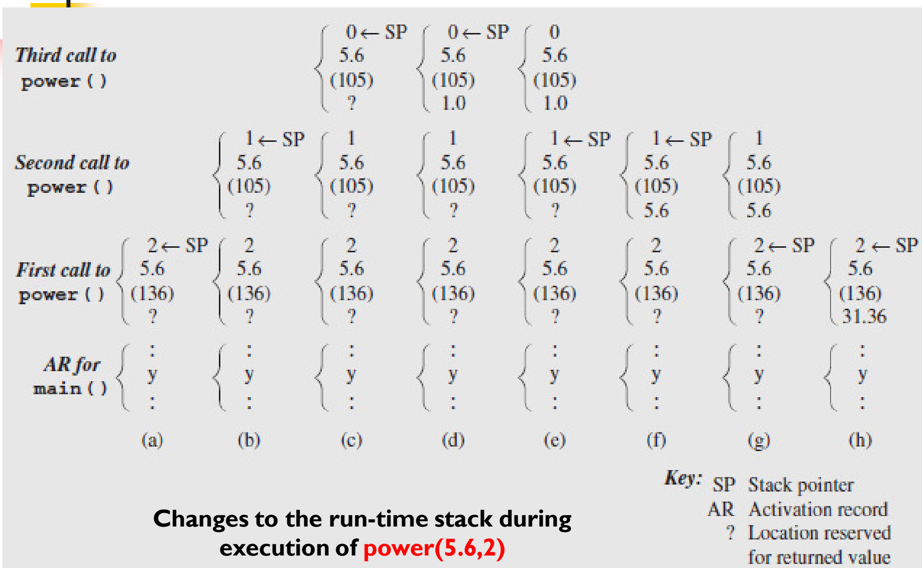
call 1	$x^4 = x \cdot x^3$	$= x \cdot x \cdot x \cdot x$
call 2	$x^3 = x \cdot x^2$	$= x \cdot x \cdot x$
call 3	$x^2 = x \cdot x^1$	$= x \cdot x$
call 4	$x^1 = x \cdot x^0$	$= x \cdot 1$
call 5	$x^0 = 1$	

Anatomy of a Recursive Call (cont.)

- The sequence of calls is kept track of on the **runtime stack**,
  - store the **return address** of the function call
  - used to remember where to resume execution after the function has completed
  - e.g., `power()` is called by the following statement:
  - ```
    /* 136 */       y = power(5.6,2);
                              .
                              .
    ```

```
/* 102 */  double power (double x, unsigned int n) {
/* 103 */       if (n == 0)
/* 104 */          return 1.0;
                //else
/* 105 */       return x * power(x,n-1);
           }
```


Anatomy of a Recursive Call (cont.)



Changes to the run-time stack during execution of power(5.6,2)
```

Anatomy of a Recursive Call (cont.)

- Possible to implement the `power()` function in a **non-recursive manner**??
  - **power(5.6,2)**

```
double nonRecPower(double x, unsigned int n) {
   double result = 1;
   for (; n > 0; n--)
      result *= x;
   return result;
}
```

- comparing this to the recursive version,
  - the recursive code is more intuitive, closer to the specification, and simpler to code