

Recursion

Lecture 6

1

Recursion

- Recursive Function
 - A function that calls itself or is part of a cycle in the sequence of function calls
- The ability to invoke itself enables a recursive function to be repeated with different parameter values.

2

Recursion

- Recursion can be an alternative solution to iteration
 - Less efficient due to the overhead for the extra function calls
- In many instances, the use of recursion enables a very natural, simple solution to a problem that otherwise would be very difficult to solve

3

Recursion

- Recursion is an important and powerful tool in problem solving and programming

4

The Nature of Recursion

- Problems that lend themselves to a recursive solution have the following characteristics:
 - One or more **simple cases** of the problem have a straightforward solution
 - The other cases can be **redefined** in terms of problems that are closer to the simple case
 - By **applying this redefinition process** every time the recursive function is called, eventually the **problem is reduced entirely to simple cases**, which are relatively easy to solve

5

The Nature of Recursion

- General form of a recursive algorithm

if this is a **simple case**

solve it

else

redefine the problem using recursion

6

The Nature of Recursion

- A problem of size n can be split into
 - a **sub-problem of size 1**
 - Can be solved easily
 - a **sub-problem of size $n - 1$**
 - Can be split further into
 - ✓ a sub-problem of size 1
 - » Can be solved easily
 - ✓ a sub-problem of size $n - 2$
 - » Can be split further into ...
- At the end, we **solve easily n problem of size 1**

7

The Nature of Recursion

- Example: Multiply 3 by 6, assuming we know how to add and we know that $x * 1 = x$

➤ **Split the problem:**

1. Multiply 6 by 2
2. Add 6 to the result

➤ **Split 1 further:**

1. Multiply 6 by 2
 1. Multiply 6 by 1
 2. Add 6 to the result of problem 1.1
2. Add 6 to the result

8

The Nature of Recursion

■ Implementation

```
int multiply (int m, int n)
{
    int ans;

    if (n == 1)
        ans = m;
    else
        ans = m + multiply (m, n - 1);
    return (ans);
}
```

9

The Nature of Recursion

- To solve a problem recursively
 - First, trust the function to solve a **simpler version** of the problem
 - Then, **build the solution** to the whole problem on the result from the simpler version

10

The Nature of Recursion

- Recursion are useful for processing varying-length lists
 - Strings
 - Linked-lists
 - Etc.

11

The Nature of Recursion

- Example: Function to count the number of times a particular character x appears in a string
 - **Split the problem:**
 1. Check the rest of the string
 2. Update the counter if the first character is x
 - **Split 1 further:**
 1. Check the rest of the string
 1. Check the rest of the string
 2. Update the counter if the second character is x
 2. Update the counter if the first character is x

12

The Nature of Recursion

■ Implementation

```
int count (char ch, const char *str)
{
    int ans;

    if (str[0] == '\0')
        ans = 0;
    else
        if (ch == str[0])
            ans = 1 + count (ch, &str[1]);
        else
            ans = count (ch, &str[1]);
    return (ans);
}
```

13

Tracing a Recursive Function

- **Hand tracing** an algorithm's execution provides **valuable insight** into how that algorithm works
- **Hand tracing**
 - Draw an **activation frame** for function call
 - An activation frame shows the **parameter values** for each call and **summarizes** the execution of the call

14

Tracing a Recursive Function

- **For functions that return a value**
 - The returned value should be shown for each activation frame
 - Example: Multiplication
- **Void functions are simpler to trace**
 - Activation frames are used, but there are no return values
 - Example: Reverse n input words

15

Tracing a Recursive Function

- **Parameters and Local Variable Stacks**
 - **Local function values are kept in a stack**
 - A **stack** is a data structure in which the last data item added is the first data item processed
 - The system stack is an area of memory where parameters and local variables are
 - **allocated** when a function is called and
 - **deallocated** when the function returns

16

Tracing a Recursive Function

- Parameters and Local Variable Stacks
 - When a program is executing
 - Calling a function pushes its local values onto the top of stack
 - Returning from a function pops its local values from the top of the stack

17

Tracing a Recursive Function

- Trace Recursive Functions
 - To understand recursion and debug a function
 - But not to try to develop a recursive algorithm
- When developing a recursive function
 - Trace a specific case simply by trusting any recursive call to return a correct value based on the function purpose

18

Recursive Math Functions

- Many mathematical functions are defined recursively
 - Example: Factorial of n ($n!$) is
 - $0! = 1$
 - $n! = n \times (n - 1)!$, for $n > 0$

19

Recursive Math Functions

- Factorial - implementation is straightforward

```
int factorial (int n)
{
    int ans;

    if (n == 0)
        ans = 1;
    else
        ans = n * factorial (n - 1);

    return (ans);
}
```

20

Recursive Math Functions

- The **Fibonacci** numbers
 - Sequence of numbers that have many uses
- The Fibonacci sequence is
 - 1, 1, 2, 3, 5, 8, ...
 - The sequence is produced as follows
 - $\text{Fibonacci}_1 = 1$
 - $\text{Fibonacci}_2 = 1$
 - $\text{Fibonacci}_n = \text{Fibonacci}_{n-1} + \text{Fibonacci}_{n-2}$, for $n > 2$

21

Recursive Math Functions

- Fibonacci - implementation is straightforward

```
int fibonacci (int n)
{
    int ans;

    if (n == 1 || n == 2)
        ans = 1;
    else
        ans = fibonacci (n - 1) + fibonacci (n - 2);

    return (ans);
}
```

22