Objectives

In these lectures, we'll consider:

- Recursive Definitions
- Function Calls and Recursive Implementation
- Anatomy of a Recursive Call
- Tail Recursion
- Nontail Recursion
- Indirect Recursion
- Excessive Recursion

1

1

Recursive Definitions

- Many useful programming constructs are defined in terms of themselves!
- These definitions are called ***recursive definitions*** and are often used to define infinite sets
- Examples of this include the Fibonacci numbers, nested parentheses, and binary strings
- Recursion is useful when an exhaustive enumeration of a set base on rules is impossible, so some others means to define it is needed
- The formal basis for these definitions must be given such that no violations of the rules occurs

2

2

Recursive Definitions

- There are two parts to a recursive definition
  - The ***anchor*** or ***ground case*** (also called the ***base case***) which establishes the basis for all the other elements of the set
  - The ***inductive clause*** which establishes rules for the creation of new elements in the set
- Using this, we can define the set of positive integer values $\mathbf{Z}^+$ as follows:
  1. $1 \in \mathbf{Z}^+$ (anchor)
  2. if $n \in \mathbf{Z}^+$, then $(n + 1) \in \mathbf{Z}^+$ (inductive clause)
  3. there are no other objects in the set $\mathbf{Z}^+$ (exclusivity rule)
- Other examples include factorials, powers of k, power sets

3

3

Recursive Definitions

- We can use recursive definitions in two ways:
  - To define new elements in the set in question
  - To demonstrate that a particular item belongs in a set
- Generally, the second use is demonstrated by repeated application of the inductive clause until the problem is reduced to the base case
- This is often the case when we want to define functions and sequences of numbers
- However this can have undesirable consequences
- For example, to determine 3! (3 factorial) using a recursive definition (n! = n * (n-1)!), we have to work back to 0!

4

4

Recursive Definitions

- This results from 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
- This is cumbersome and computationally inefficient – a simple loop is much more efficient here
- It would be helpful to find a formula that is equivalent to the recursive one without referring to previous values
- For factorials, we can use $n! = \prod_{i=1}^{n} i$
- For more complex examples, however, this is frequently non-trivial and often quite difficult to achieve

5

5

Recursive Definitions

- These discussions and examples have been on a theoretical basis
- From the standpoint of computer science, recursion occurs frequently in language definitions as well as programming
- Fortunately, the translation from specification to code is fairly straightforward; consider our factorial example coded in C++:

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

6

6

Recursive Definitions (continued)

- Although the code is simple, the underlying ideas supporting its operation are quite involved
- Fortunately, most modern programming languages incorporate mechanisms to support the use of recursion, making it (mostly) transparent to the user
- Typically, recursion is supported through use of the **runtime stack** – a data structure that handles function calls
- A runtime stack has no problem with calling a function that happens to match a previously called function. Each call has its own program counter and its own copies of local variables – they can even share the same executable code segment!
- So to get a clearer understanding of recursion, we will look at how function calls are processed

Function Calls and Recursive Implementation

- What kind of information must we keep track of when a function is called?
- If the function has parameters, they need to be initialized to their corresponding argument values – call-by-value will copy these values into local-scope storage (if using pointers, then passing pointers as call-by-value works like call-by-reference)
- In addition, we need to know where to resume the calling function once the called function is complete, by use of a stored PC (Program Counter) containing the address of the next instruction to be executed – every function keeps its own local copy of the PC
- Since functions can be called from other functions, we need to keep track of local variables for scope purposes
- Because we may not know in advance how many calls will occur, a stack is a more efficient way to save information (as long as we have enough memory allocated for the process to hold the stack)

Function Calls and Recursive Implementation

- We can characterize the state of a function by a set of information, called an ***activation record*** or ***stack frame***
- This is stored on the runtime stack, and contains the following information:
  - Values of the function's parameters, addresses of reference variables (including arrays)
  - Copies of local variables
  - The return address of the calling function (the PC value)
  - A dynamic link to the calling function's activation record
  - The function's return value if it is not declared as **`void`**
- Every time a function is called, its activation record is created and placed on top of the runtime stack

9

9

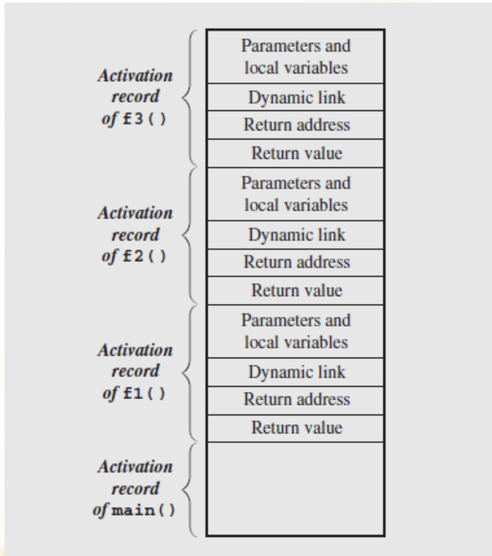
Function Calls and Recursive Implementation

- So the runtime stack always contains the current state of the function
- As an illustration, consider a function **`f1()`** called from **`main()`**
- It in turn calls function **`f2()`**, which calls function **`f3()`**
- Once **`f3()`** completes, its activation record is popped off the stack, and function **`f2()`** can resume and access information in its record
- If **`f3()`** instead calls another function, the new function has its activation record pushed onto the stack as **`f3()`** is suspended

10

10

Function Calls and Recursion



Function Calls and Recursive Implementation

- The use of activation records on the runtime stack allows recursion to be implemented and handled correctly
- Essentially, when a function calls itself recursively, it simply pushes a new activation record of itself on the stack, just as it would for calling a different function. In other words, the function doesn't necessarily have to know it's recursing!
- This suspends the calling instance of the function, and allows the new activation record to carry on the process
- Thus, a recursive call creates a series of activation records for different instances of the **same** function

Anatomy of a Recursive Call

- To gain further insight into the behavior of recursion, let's dissect a recursive function and analyze its behavior
- The function we will look at is defined in the text, and can be used to raise a number *x* to a non-negative integer power *n*:

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

- We can also represent this function using C++ code, shown on the next slide

13

Anatomy of a Recursive Call

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

- Using the definition, the calculation of x^4 would be calculated as follows: $x^4 = x \cdot x^3 = x \cdot (x \cdot x^2) = x \cdot (x \cdot (x \cdot x^1)) = x \cdot (x \cdot (x \cdot (x \cdot x^0))) = x \cdot (x \cdot (x \cdot (x \cdot 1))) = x \cdot (x \cdot (x \cdot (x))) = x \cdot (x \cdot (x \cdot x)) = x \cdot (x \cdot x \cdot x) = x \cdot x \cdot x \cdot x$
- Notice how repeated application of the inductive step leads to the base case

14