Announcements

• Tests will be returned today. We will discuss test content, test preparations, and grades.

• HW 4 is due tonight. I will stay after class to help with problems.

• Lab next week will be on recursion.

Today’s Class

• Introduction to recursion: factorials and exponentiation

• How recursion works

• Iteration vs. recursion

• Rules for writing recursive functions

• Another example: printing an array in order and in reverse order.

• More advanced examples:
  – Counting the number of subsets of size $k$
  – Towers of Hanoi

Reading: All of Chapter 2 of Carrano and Prichard.
Recursive Definitions of Factorials and Integer Exponentiation

- The factorial is defined for non-negative integers as
  \[ n! = \begin{cases} 
  n \cdot (n - 1)! & n > 0 \\
  1 & n = 0 
  \end{cases} \]

- Computing integer powers is defined as:
  \[ n^p = \begin{cases} 
  n \cdot n^{p-1} & p > 0 \\
  1 & p = 0 
  \end{cases} \]

- These are both examples of recursive definitions.
Recursive C++ Functions

C++, like most other modern programming languages, allows functions to call themselves. This gives a direct method of implementing recursive functions.

- Here’s the implementation of factorial:

```c++
int fact( int n )
{
    if ( n == 0 )
        return 1;
    else
    {
        int result = fact( n-1 );
        return n * result;
    }
}
```

- Here’s the implementation of exponentiation:

```c++
int intpow( int n, int p )
{
    if ( p == 0 )
        return 1;
    else
    {
        return n * intpow( n, p-1 );
    }
}
```

The Mechanism of Recursive Function Calls

- When it makes a recursive call (or any function call), a program creates an activation record to keep track of
  - The function’s own completely separate instances of parameters and local variables.
  - The location in the calling function code to return to when the function is complete.
– Which activation record to return to when the function is done.

• This is illustrated in the following diagram of the call fact(4). Each box is an activation record, the solid lines indicate the function calls, and the dashed lines indicate the returns.

- We will draw activation records to illustrate the behavior of other recursive functions as well.
Iteration vs. Recursion

• Each of the above functions could also have been written using a for loop, i.e. iteratively.

• For example, here is an iterative version of factorial:

```c
int ifact( int n )
{
    int result = 1;
    for ( int i=1; i<=n; ++i )
        result = result * i;
    return result;
}
```

• Iterative functions are generally faster than their corresponding recursive functions. Compiler optimizations sometimes (but not always!) can take care of this by automatically eliminating the recursion.

• Sometimes writing recursive functions is more natural than writing iterative functions, however. We will see some examples of this.
Rules for Writing Recursive Functions

Here is an outline of five steps I find useful in writing and debugging recursive functions:

1. Handle the base case(s) first, at the start of the function.

2. Define the problem solution in terms of smaller instances of the problem. This defines the necessary recursive calls. It is also the hardest part!

3. Figure out what work needs to be done before making the recursive call(s).

4. Figure out what work needs to be done after the recursive call(s) complete(s) to finish the computation.

5. Assume the recursive calls work correctly, but make sure they are progressing toward the base case(s)!
Example: Printing the Contents of An Array

The following example is important for thinking about the mechanisms of recursion.

- Printing them in order:

```cpp
void print_array( float arr[], int i, int n )
{
    if ( i < n ){
        cout << i << " : " << arr[i] << endl;
        print_array( arr, i+1, n );
    }
}
```

- Printing them in reverse order:

```cpp
void rprint_array( float arr[], int i, int n )
{
    if ( i < n ){
        rprint_array( arr, i+1, n );
        cout << i << " : " << arr[i] << endl;
    }
}
```

- Of course these functions could also be written using pointers...
Counting the number of subsets

- Problem: in an ordinary deck of 52 playing cards, how many different 5 card “hands” are there?
- This is an instance of the following problem: given a set containing \( n \) elements, how many different \( k \) element subsets are possible?
- Let \( C(n, k) \) denote this number. It is a function of both \( n \) and \( k \).
- In class we will think about
  - the possible base cases, and
  - the recursive definition of the function.
- Then we will write the actual function, check it using our 5 rules above, and test it.
- Why is the resulting function so slow?
Towers Of Hanoi

Legend has it that in Hanoi there is a group of monks moving a set of 64 disks from pole A to pole B using pole C as an extra pole. When they complete this task, the world will end. Are you scared? Fortunately, there are restrictions: initially the disks were stacked in order of increasing size, only one disk can be moved at a time, and no disk can be placed on top of a smaller disk at any time.

- What is the sequence of disk moves that these monks must take? How long will it take to complete them?
- The answer to the first question is almost impossible to arrive at without recursion:
  1. Move (recursively) the top 63 disks from pole A to pole C, using B as an extra pole.
  2. Move the bottom disk to pole B.
  3. Move the top 63 disks from pole C to pole B, using A as an extra pole.
- Here’s C++ code to implement this:

```cpp
void solveTowers( int count, // the number of disks on the "source" pole
    char source,
    char destination,
    char extra )
{
    if ( count == 1 )
        cout << "Move top disk from pole " << source
            << " to pole " << destination << endl;
    else
    {
        solveTowers( count-1, source, extra, destination );
        solveTowers( 1, source, destination, extra );
        solveTowers( count-1, extra, destination, source );
    }
}
```
- Here’s the answer the second question: it will take them $2^{64} - 1$ moves. If they can do move one disk per second, they will require $5.85 \times 10^{11}$ years to finish the task.
Summary

• Recursive functions make calls to themselves.

• Solutions to many problems can be defined recursively.

• Recursion works in implementation because each recursive call creates a separate instance of the recursive function, including parameters and local variables.