

# CSCI-1200 Computer Science II — Fall 2006

## Lecture 12 — Recursion I

### Announcements: Test 2 Information

- Test 2 will be held **Tuesday, October 17th, 2006 from 2-3:50pm in West Hall Auditorium**. No make-ups will be given except for emergency situations, and even then a written excuse from the Dean of Students office will be required.
- Coverage: Lectures 1-11, Labs 1-7, HW 1-5.
- Closed-book and closed-notes *except for 1 sheet of notes on 8.5x11 inch paper (front & back) that may be handwritten or printed*. Computers, cell-phones, palm pilots, calculators, PDAs, music players, etc. are not permitted and must be turned off.
- All students must bring their Rensselaer photo ID card.
- Practice problems are available on the course website. Solutions will be posted on Monday.

### Review from Lecture 11

- Overloading operators, example: complex numbers
- Friend functions & classes
- Operators as member functions, non-member functions, or friend functions
- Stream operators

### Today's Lecture

- Introduction to recursion: factorials and exponentiation
- How recursion works
- Iteration vs. recursion
- Rules for writing recursive functions
- Examples that we will work on together:
  - Printing a vector in reverse order
  - Binary search

We will look at more sophisticated problems in Lecture 10.

### 12.1 Recursive Definitions of Factorial 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:

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

## 12.2 Recursive C++ Functions

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

- Here's the implementation of factorial:

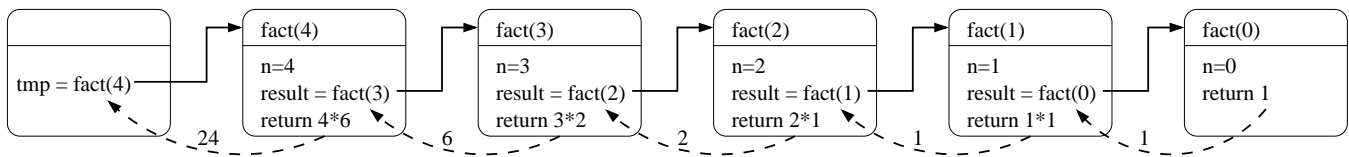
```
int fact(int n) {
    if (n == 0) {
        return 1;
    } else {
        int result = fact(n-1);
        return n * result;
    }
}
```

- And here's the implementation of exponentiation:

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

## 12.3 The Mechanism of Recursive Function Calls

- For each recursive call (or any function call), a program creates an *activation record* to keep track of:
  - **Completely separate instances** of the parameters and local variables for the newly-called function.
  - The location in the calling function code to return to when the newly-called function is complete. (Who asked for this function to be called? Who wants the answer?)
  - Which activation record to return to when the function is done. For recursive functions this can be confusing since there are multiple activation records waiting for an answer from the same function.
- 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. Inside of each box we list the parameters and local variables and make notes about the computation.



- This chain of activation records is stored in a special part of program memory called *the stack*.

## 12.4 Iteration vs. Recursion

- Each of the above functions could also have been written using a `for` or `while` loop, i.e. *iteratively*.
- For example, here is an iterative version of factorial:

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

- Often writing recursive functions is more natural than writing iterative functions, especially for a first draft of an problem implementation.

- You should learn how to recognize whether an implementation is recursive or iterative, and practice rewriting one version as the other. Note: We'll see that not all recursive functions can be rewritten in iterative form!
- Note: The order notation for the number of operations for the recursive and iterative versions of an algorithm is usually the same. However in C, C++, Java, and some other languages, *iterative functions are generally faster than their corresponding recursive functions*. This is due to the overhead of the function call mechanism. Compiler optimizations will sometimes (but not always!) reduce the performance hit by automatically eliminating the recursive function calls. This is called *tail call optimization*.

## 12.5 Exercise

1. Draw a picture to illustrate the activation records for the function call

```
cout << intpow(4, 4) << endl;
```

2. Write an iterative version of `intpow`.

## 12.6 Rules for Writing Recursive Functions

Here is an outline of five steps that are useful in writing and debugging recursive functions. Note: You don't have to do them in exactly this order...

1. Handle the base case(s).
2. Define the problem solution in terms of smaller instances of the problem. Use *wishful thinking*, i.e., if someone else solves the problem of `fact(4)` I can extend that solution to solve `fact(5)`. 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. (What are you going to do with the result of the recursive call?)
5. Assume the recursive calls work correctly, but make sure they are progressing toward the base case(s)!

## 12.7 Example: Printing the Contents of a Vector

- Here is a function to print the contents of a vector. Actually, it's two functions: a *driver function*, and a true recursive function. It is common to have a driver function that just initializes the first recursive function call.

```
void print_vec(vector<int>& v) {
    print_vec(v, 0);
}

void print_vec(vector<int>& v, unsigned int i) {
    if (i < v.size()) {
        cout << i << ": " << v[i] << endl;
        print_vec(v, i+1);
    }
}
```

- **Exercise:** What will this print when called in the following code?

```
int main() {  
    vector<int> a;  
    a.push_back(3); a.push_back(5); a.push_back(11); a.push_back(17);  
    print_vec(a);  
}
```

- **Exercise:** How can you change the second `print_vec` function as little as possible so that this code prints the contents of the vector in reverse order?

## 12.8 Binary Search

- Suppose you have a `vector<T> v` (where `T` is a placeholder for a specific type), sorted so that:

`v[0] <= v[1] <= v[2] <= ...`

- Now suppose that you want to find if a particular value `x` is in the vector somewhere. How can you do this without looking at every value in the vector?
- The solution is an algorithm called *binary search*. Let's write the recursive version of this algorithm. We're going to write it for general vectors using the template syntax. This function will work on vectors of *any* type, as long as the basic operators such as `<` and `==` are defined for that type. Here's the prototype for the driver function:

```
template <class T> bool binsearch(const vector<T>& v, const T& x);
```

## 12.9 Looking Ahead to Lecture 13

- The problems we looked at today can be easily solved using non-recursive techniques. This will not be true when we look at these two problems next time:
  - Sorting a vector using the mergesort technique.
  - Solving a “word search” problem in a grid of letters allowing non-straight paths.