CSCI-1200 Data Structures — Spring 2016
Lecture 9 — Recursion

Announcements: Test 1 Information

- Test 1 will be held **Monday, Feb. 29, 2016 from 6-7:50pm**.
- Students have been randomly assigned to a test room and seating zone. Check the homework submission website.
- No make-ups will be given except for pre-approved absence or illness, and a written excuse from the Dean of Students or the Student Experience office or the RPI Health Center will be required.
- Contact the professor by email by **TODAY Friday Feb. 26 to arrange for extra time accommodations**.
- Coverage: Lectures 1-9, Labs 1-5, and Homeworks 1-4.
- Practice problems from previous exams will be posted on the course website. Solutions to the problems will be posted a day or two before the exam. The best way to prepare is to completely work through and write out your solution to each problem, *before* looking at the answers.
- Bring to the exam room:

  Your Rensselaer photo ID card.

  Pencil(s) & eraser (pens are ok, but not recommended). The exam *will* involve handwriting code on paper (and other short answer problem solving). Neat legible handwriting is appreciated. We will be somewhat forgiving to minor syntax errors – it will be graded by humans not computers :)

  [OPTIONAL] You may bring 1 sheet of notes on 8.5x11 inch paper (front & back) that may be handwritten or printed. Learn how to print double-sided. You may not staple or tape or glue together 2 or more single-sided sheets of paper.
- Do not bring your own scratch paper. We will provide scratch paper.
- Computers, cell-phones, smart watches, calculators, music players, etc. are not permitted. Please do not bring your laptop, books, backpack, etc. to the exam room – leave everything in your dorm room. *Unless you are coming directly from another class or sports/club meeting.*

Review from Lecture 8

- Dynamically allocated memory in classes
- Copy constructors, assignment operators, and destructors
- Templated classes, Implementation of the DS `Vec` class, mimicking the STL `vector` class

Today

- An introduction to recursion

9.1 Recursive Definitions of Factorials and Integer Exponentiation

- The factorial function 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*.

9.2 Recursive C++ Functions

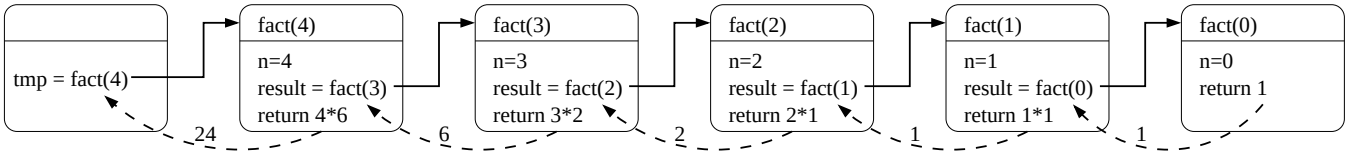
C++, like other modern programming languages, allows functions to call themselves. This gives a direct method of implementing recursive functions. Here are the recursive implementations of factorial and integer power:

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

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

9.3 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*.

9.4 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 a 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 *easily* 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.*
- Some languages such as Haskel and other *functional programming languages* are optimized for recursion.

9.5 Exercises

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`.

9.6 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)!

9.7 Recursion 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(std::vector<int>& v, unsigned int i) {
  if (i < v.size()) {
    cout << i << ": " << v[i] << endl;
    print_vec(v, i+1);
  }
}
void print_vec(std::vector<int>& v) {
  print_vec(v, 0);
}
```

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

```
int main() {
  std::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?

9.8 Binary Search

- Suppose you have a `std::vector<T> v` (for a placeholder type T), 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 a recursive algorithm called *binary search*, based on the idea of checking the middle item of the search interval within the vector and then looking either in the lower half or the upper half of the vector, depending on the result of the comparison.

```
template <class T>
bool binsearch(const std::vector<T> &v, int low, int high, const T &x) {
 if (high == low) return x == v[low];
  int mid = (low+high) / 2;
  if (x <= v[mid])
    return binsearch(v, low, mid, x);
  else
    return binsearch(v, mid+1, high, x);
}
template <class T>
bool binsearch(const std::vector<T> &v, const T &x) {
  return binsearch(v, 0, v.size()-1, x);
}
```

9.9 Exercises

1. Write a non-recursive version of binary search.
2. If we replaced the if-else structure inside the recursive binsearch function (above) with

```
  if ( x < v[mid] )
    return binsearch( v, low, mid-1, x );
  else
    return binsearch( v, mid, high, x );
```

   would the function still work correctly?