CSCI-1200 Data Structures — Fall 2013
Lecture 12 — Recursion

Review from Lecture 11 & sneak peek at Lab 7

- Limitations of singly-linked lists
- Doubly-linked lists:
  - Structure
  - Insert
  - Remove
- Our own version of the STL `list<T>` class, named `dslist`
- Implementing `list<T>::iterator`
- Importance of destructors & using Dr. Memory / Valgrind to find memory errors
- Decrementing the `end()` iterator

Today's Lecture

- Recursion vs. Iteration
- "Rules" for writing recursive functions
- Advanced Recursion — problems that cannot be easily solved using iteration (for or while loops):
  - Merge sort
  - Non-linear maze search

12.1 Recursive C++ Functions

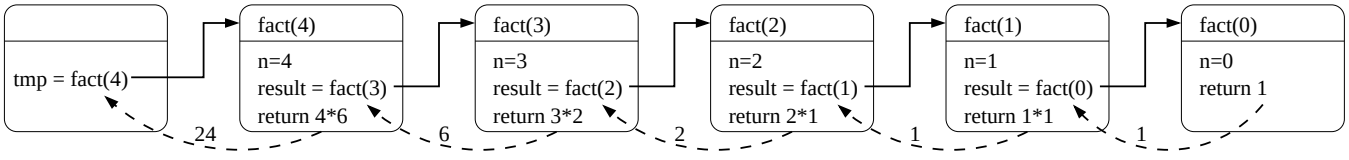
C++, like other modern programming languages, allows functions to call themselves. This gives a direct method of implementing recursive functions. Here are the recursive implementations of factorial and integer power:

```
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 );
  }
}
```

12.2 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*.

12.3 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.*

12.4 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`.

12.5 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)!

12.6 Recursion Example: Merge Sort

- Idea: 1) Split a vector in half, 2) Recursively sort each half, and 3) Merge the two sorted halves into a single sorted vector.
- Suppose we have a vector called `values` having two halves that are each already sorted. In particular, the values in subscript ranges `[low..mid]` (the lower interval) and `[mid+1..high]` (the upper interval) are each in increasing order.
- Which values are candidates to be the first in the final sorted vector? Which values are candidates to be the second?
- In a loop, the merging algorithm repeatedly chooses one value to copy to `scratch`. At each step, there are only two possibilities: the first uncopied value from the lower interval and the first uncopied value from the upper interval.
- The copying ends when one of the two intervals is exhausted. Then the remainder of the other interval is copied into the scratch vector. Finally, the entire scratch vector is copied back.

12.7 Exercise: Complete the Merge Sort Implementation

```
#include <iostream>
#include <vector>

template <class T> void mergesort(std::vector<T>& values);
template <class T> void mergesort(int low, int high, std::vector<T>& values, std::vector<T>& scratch);
template <class T> void merge(int low, int mid, int high, std::vector<T>& values, std::vector<T>& scratch);

int main() {
  std::vector<double> pts(7);
  pts[0] = -45.0;  pts[1] =   89.0;  pts[2] =    34.7;  pts[3] =   21.1;
  pts[4] =   5.0;  pts[5] =  -19.0;  pts[6] =  -100.3;
  mergesort(pts);
  for (unsigned int i=0; i<pts.size(); ++i)
    std::cout << i << ": " << pts[i] << std::endl;
}

// The driver function for mergesort.  It defines a scratch std::vector for temporary copies.
template <class T>
void mergesort(std::vector<T>& values) {
  std::vector<T> scratch(values.size());
  mergesort(0, int(values.size()-1), values, scratch);
}

// Here's the actual merge sort function.  It splits the std::vector in
// half, recursively sorts each half, and then merges the two sorted
// halves into a single sorted interval.
template <class T>
void mergesort(int low, int high, std::vector<T>& values, std::vector<T>& scratch) {
  std::cout << "mergesort:  low = " << low << ", high = " << high << std::endl;
  if (low >= high)  //  intervals of size 0 or 1 are already sorted!
    return;
  int  mid = (low + high) / 2;
  mergesort(low, mid, values, scratch);
  mergesort(mid+1, high, values, scratch);
  merge(low, mid, high, values, scratch);
}

// Non-recursive function to merge two sorted intervals (low..mid & mid+1..high)
// of a std::vector, using "scratch" as temporary copying space.
template <class T>
void merge(int low, int mid, int high, std::vector<T>& values, std::vector<T>& scratch) {
  std::cout << "merge:  low = " << low << ", mid = " << mid << ", high = " << high << std::endl;
  int i=low, j=mid+1, k=low;









}
```

12.8 Thinking About Merge Sort

- It exploits the power of recursion! We only need to think about
  - Base case (intervals of size 1)
  - Splitting the vector
  - Merging the results
- We can insert `cout` statements into the algorithm and use this to understand how this is is happening.
- Can we analyze this algorithm and determine the order notation for the number of operations it will perform? Count the number of pairwise comparisons that are required.

12.9 Example: Word Search

- Take a look at the following grid of characters.

```
heanfuyaadfj
crarneradfad
chenenssartr
kdfthileerdr
chadufjavcze
dfhoepradlfc
neicpemrtlkf
paermerohtrr
diofetaycrhg
daldruetryrt
```

- The usual problem associated with a grid like this is to find words going forward, backward, up, down, or along a diagonal. Can you find "`computer`"?
- A sketch of the solution is as follows:
  - The grid of letters is represented as `vector<string> grid;` Each string represents a row. We can treat this as a *two-dimensional array*.
  - A word to be sought, such as "`computer`" is read as a string.
  - A pair of nested for loops searches the grid for occurrences of the first letter in the string. Call such a location (r, c)
  - At each such location, the occurrences of the second letter are sought in the 8 locations surrounding (r, c).
  - At each location where the second letter is found, a search is initiated in the direction indicated. For example, if the second letter is at $(r, c-1)$, the search for the remaining letters proceeds up the grid.
- The implementation takes a bit of work, but is not too bad.

12.10 Example: Nonlinear Word Search

- Today we'll work on a different, but somewhat harder problem: What happens when we no longer require the locations to be along the same row, column or diagonal of the grid, but instead allow the locations to snake through the grid? The only requirements are that
  1. the locations of adjacent letters are connected along the same row, column or diagonal, and
  2. a location can not be used more than once in each word
- Can you find `rensselaer`? It is there. How about `temperature`? Close, but nope!
- The implementation of this is very similar to the implementation described above until after the first letter of a word is found.
- We will look at the code during lecture, and then consider how to write the recursive function.

12.11 Exercise: Complete the implementation

```
#include <algorithm>
#include <fstream>
#include <iostream>
#include <string>
#include <vector>

// Simple class to record the grid location.
class loc {
public:
  loc(int r=0, int c=0) : row(r), col(c) {}
  int row, col;
};

bool operator== (const loc& lhs, const loc& rhs) {
  return lhs.row == rhs.row && lhs.col == rhs.col;
}

//  Prototype for the main search function
bool search_from_loc(loc position, const std::vector<std::string>& board,
                     const std::string& word, std::vector<loc>& path);

// Read in the letter grid, the words to search and print the results
int main(int argc, char* argv[]) {
  if (argc != 2) {
    std::cerr << "Usage: " << argv[0] << " grid-file\n";
    return 1;
  }
  std::ifstream istr(argv[1]);
  if (!istr) {
    std::cerr << "Couldn't open " << argv[1] << '\n';
    return 1;
  }

  std::vector<std::string> board;
  std::string word;
  std::vector<loc> path;           //  The sequence of locations...
  std::string line;

  //  Input of grid from a file.  Stops when character '-' is reached.
  while ((istr  >> line) && line[0] != '-')
    board.push_back(line);

  while (istr >> word) {
    bool found = false;
    std::vector<loc> path;  //  Path of locations in finding the word

    //  Check all grid locations.  For any that have the first
    //  letter of the word, call the function search_from_loc
    //  to check if the rest of the word is there.

    for (unsigned int r=0; r<board.size() && !found; ++r) {
      for (unsigned int c=0; c<board[r].size() && !found; ++c) {
        if (board[r][c] == word[0] &&
            search_from_loc(loc(r,c), board, word, path))
          found = true;
      }
    }

    //  Output results
    std::cout << "\n** " << word << " ** ";
    if (found) {
      std::cout << "was found.  The path is \n";
      for(unsigned int i=0; i<path.size(); ++i)
        std::cout << "  " << word[i] << ":  (" << path[i].row << "," << path[i].col << ")\n";
```

```
    } else {
      std::cout << " was not found\n";
    }
  }
  return 0;
}

// helper function to check if a position has already been used for this word
bool on_path(loc position, std::vector<loc> const& path) {
  for (unsigned int i=0; i<path.size(); ++i)
    if (position == path[i]) return true;
  return false;
}

bool search_from_loc(loc position, //  current position
                     const std::vector<std::string>& board,
                     const std::string& word,
                     std::vector<loc>& path)  //  path up to the current pos
{









}
```

12.12 Summary of Nonlinear Word Search Recursion

- Recursion starts at each location where the first letter is found
- Each recursive call attempts to find the next letter by searching around the current position. When it is found, a recursive call is made.
- The current path is maintained at all steps of the recursion.
- The "base case" occurs when the path is full **or** all positions around the current position have been tried.

12.13 Exercise: Analyzing our Nonlinear Word Search Algorithm

What is the order notation for the number of operations?

Final Note

We've said that recursion is sometimes the *most natural way* to begin thinking about designing and implementing many algorithms. It's ok if this feels downright uncomfortable right now. Practice, practice, practice!