Lists and Recursion

Lists in C++

Iterators

Recursion

Read chapter 21 of the textbook!

These slides will cover some of the contents, but the textbook is much more detailed.

Lists

- The web site http://www.cplusplus.com/reference/stl/ provides a reference for the C++ Standard Template Libraries
- The STL libraries are provided using `#include <cstdlib>`
- The `list` container template provides a doubly linked list structure, and its iterators
  - An iterator is used to step through the elements stored in a container
  - It works pretty much as the pointer `current` in our linked list class
  - However, iterators are external to the list object and the same list can have multiple iterators associated to it. In our linked list class, there was only one `current` pointer. This adds flexibility to the use of lists.

Iterators

- Iterators are objects created using member functions of the container class, for example:
  - `begin()` returns an iterator used to access the first item in the container
  - `end()` returns an iterator that is just past the last item in the container
    - And should *never* be used to attempt to access an item
  - The `*` operator is used to access the current item pointed to by the iterator
  - The `++` operator is used to move forward to the next item in the container
  - The `--` operator is used to move to the previous item in the collection
  - The `!=` and `==` operators can be used *with iterators on the same container* to see whether they are equal
- For more information, please visit http://www.cplusplus.com/reference/iterator/

Example – Adding to a list

```
int main()
{
// Creating a list of char
    list<char>* l1 = new list<char>();

// Creating a list of account
    list<account>* l3 = new list<account>();

// Creating instances of char
    char* c1 = new char('F');
    char* c3 = new char('H');

// Creating some instances of account
    account* a1 = new account(10);
    account* a3 = new account(30);

// Populating list by adding to head
    l1->push_front(*c1);
    l1->push_front(*c3);

// Populating list by adding to tail
    l3->push_back(*a1);
    l3->push_back(*a3);

    return EXIT_SUCCESS;
}
```

Example – Creating iterators

```
// To access first item
list<char>::iterator i1_start = l1->begin();

// Just past last item
list<char>::iterator i1_end = l1->end();

// Iterator to current
list<char>::iterator currentChar = i1_start;

// Same iterators, but for the list of accounts
list<account>::iterator i3_start = l3->begin();
list<account>::iterator i3_end = l3->end();
list<account>::iterator currentAcc = i3_start;
```

Example – Accessing items

```
// Stepping forward using iterator
    currentChar = i1_start;
    while (currentChar != i1_end){cout << * currentChar << endl; currentChar++;}

// Stepping backward with iterator
    currentChar = i1_end;
    while (currentChar != i1_start){currentChar--; cout << *currentChar << endl;}

// Now with the account data type: Printing the balances, back to front
    currentAcc = i3_end;
    while (currentAcc != i3_start){currentAcc--; cout << currentAcc->balance() << endl;}

// Printing and removing items
// Note pop_front() and pop_back() are void functions
    while (l1->size() != 0)
    {
        cout << l1->back() << endl;
        l1->pop_back();
    }

    while (l3->size() != 0){
        cout << l3->front() << endl; // front() returns the
        l3->pop_front();             // first element.
    }
```

Recursion

Read chapter 15 of the textbook!

These slides will cover some of the contents, but the textbook is much more detailed.

Recursion

- Recursion is a process of solving a problem by reducing it to smaller versions of itself. The main characteristic of a recursive method is that it calls itself.
- Moreover, a recursive method needs one or more termination conditions, also known as a base case(s), which if fails, triggers a recursive call.
- The base cases must guarantee that the recursion ends.
- For example:

```
int factorial (int n)
{
   if (n == 1) // base case
      return 1;
   else
      return n * factorial (n - 1); // recursive call
}
```

Recursion

- When this executes, for example in calculating `factorial(4)`, the following occurs:

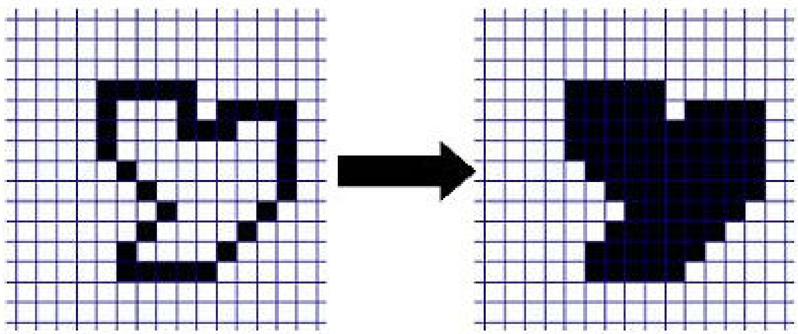
```
factorial(4)
   factorial(3)
      factorial(2)
         factorial(1)
         return 1
      return 2             (2 * 1)
   return 6                (3 * 2)
return 24                  (6 * 4)
```

Recursion

- To create a recursive algorithm:
  - Find one or more simple cases of the problem that can be solved directly, i.e. the base cases
  - Find a way to express the problem solution as a subproblem, i.e. a smaller version of the problem, which will produce a partial solution.
  - Find a way to combine the partial solutions
- Make sure that all the subproblems will eventually reach one of the base cases, as recursive solutions can perform *infinite recursion* if you are not careful.
- All recursive solutions can be re-written iteratively
- Some recursive implementations are much less time efficient than the equivalent iterative solution, for example `fibonacci()`.

Recursion

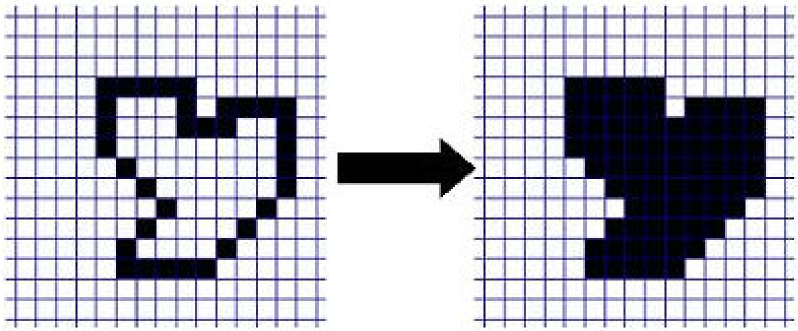
- `FloodFill`



1. If the point we want to color is already colored, don't do anything and stop, else color it.
2. If the point above it is not colored, call `FloodFill` from that point (recursive step).
3. If the point to the right is not colored, call `FloodFill` from that point (recursive step).
4. If the point to the left is not colored, call `FloodFill` from that point (recursive step).
5. If the point below it is not colored, call `FloodFill` from that point (recursive step).

Recursion

- FloodFill



```
class MyCanvas {
    bool board[100][100];
    void FloodFill(int x, int y, boolean c) {
        if (board[x][y] == c) return;
        board[x][y] = c;
        if (x < 99 && board[x+1][y] != c)
            FloodFill(x+1,y,c); // to the right

        if (x > 0 && board[x-1][y] != c)
            FloodFill(x-1,y,c); // to the left

        if (y<99 && board[x][y+1] != c)
            FloodFill(x,y+1,c); // down

        if (y>0 && board[x][y-1] != c)
            FloodFill(x,y-1,c); // up
    }
}
```

Types of recursive functions

- There are three basic structures to recursive solutions:
  1. Do something, recurse, and return nothing
  2. Do something, recurse, and return the same value on each call
  3. Do something, recurse, and return a new value on each call

First type of recursion

- Do something, recurse, and return nothing
  - Example: Printing out a string

```
// Prints out the content of a string, one character at a time

if there are no more characters to print
   return
else
   print out the first character
   print out the rest of the string // recursive call
```

First type of recursion

```
void str_print(string str, int index)
{
    if (index < str.length())
    {
        cout << str[index];
        str_print(str, index + 1);
    }
}
```

A typical call would be:

```
string str = "SENG1120";
str_print(str, 0);
```

First type of recursion

```
str_print("SENG1120", 0);
                                                      S
  str_print("SENG1120", 1);
                                                      SE
    str_print("SENG1120", 2);
                                                      SEN
      str_print("SENG1120", 3);
                                                      SENG
        str_print("SENG1120", 4);
                                                      SENG1
          str_print("SENG1120", 5);
                                                      SENG11
            str_print("SENG1120", 6);
                                                      SENG112
              str_print("SENG1120", 7);
                                                      SENG1120
              return
            return
          return
        return
      return
    return
  return
return
```

Second type of recursion

- Recurse and return the same value on each call
  - Example: Finding the first occurrence of a target character

```
// Return the position of the first instance
// of the target character in a String, or -1
// if it doesn't exist.

if there are no more characters to examine
   return -1
else
   if the current character equals the target
      return the current position
   else
      return the result of searching the rest of the string
```

Second type of recursion

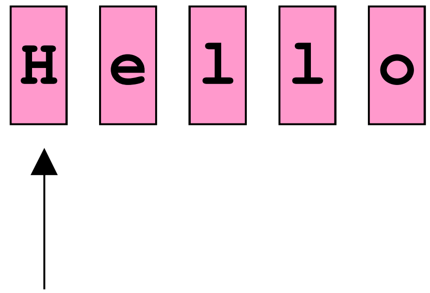
```
int find(string str, int index, char target)
{
    if (index > str.length())
        return -1;
    else
        if (str[index] == target)
            return index;
        else
            return find(str, index + 1, target);
}
```

- A typical call would be:

```
string str = "Hello";
cout << find(str, 0, 'l') << endl;
```

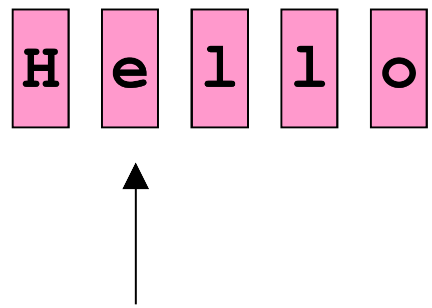
Second type of recursion

```
find("Hello", 0, 'l');
```



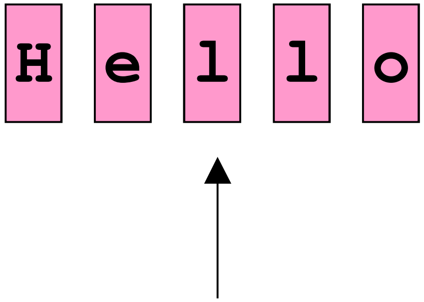
Second type of recursion

```
find("Hello", 0, 'l');
  find("Hello", 1, 'l');
```



Second type of recursion

```
find("Hello", 0, 'l');
  find("Hello", 1, 'l');
    find("Hello", 2, 'l');
    return 2
  return 2
return 2
```



Third type of recursion

- Recurse and return a different value on each call
  - Example: Reverse the order of the characters in a string

```
// Return a new string whose characters are
// in the reverse order of those of the
// parameter string.

if there are no more characters to examine
   return the empty string
else
   reverse the rest of the string after the current char
   return the concatenation of this result
      and the char at the current position
```

Third type of recursion

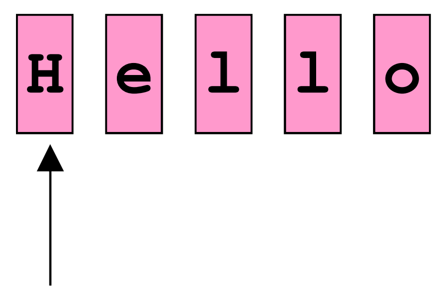
```
string reverse(string str, int index)
{
    if (index == str.length())
        return "";
    else
    {
        string rest = reverse(str, index + 1);
        return (rest + str[index]);
    }
}
```

- A typical call would be:

```
cout << reverse("Hello", 0) << endl;
```

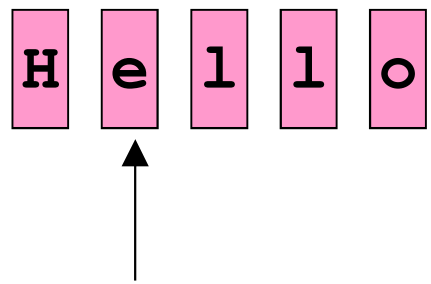
Third type of recursion

```
reverse("Hello", 0);
```



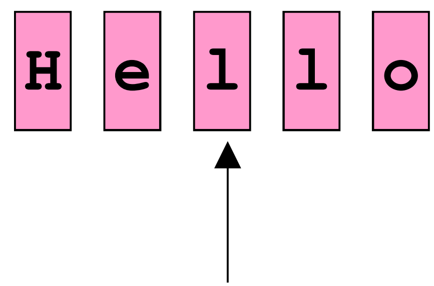
Third type of recursion

```
reverse("Hello", 0);
  reverse("Hello", 1);
```



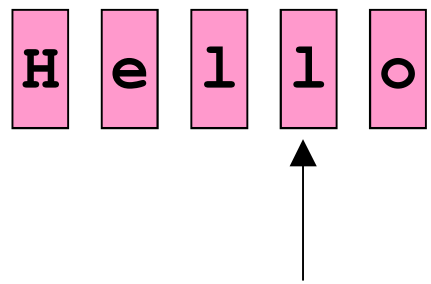
Third type of recursion

```
reverse("Hello", 0);
  reverse("Hello", 1);
    reverse("Hello", 2);
```



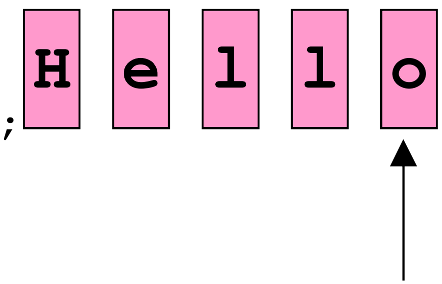
Third type of recursion

```
reverse("Hello", 0);
  reverse("Hello", 1);
    reverse("Hello", 2);
      reverse("Hello", 3);
```



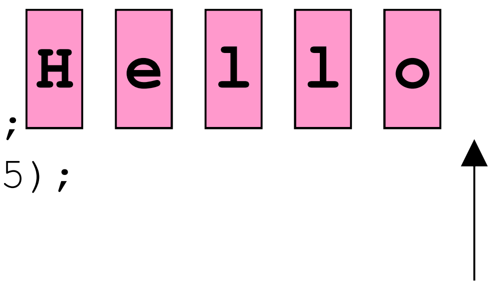
Third type of recursion

```
reverse("Hello", 0);
  reverse("Hello", 1);
    reverse("Hello", 2);
      reverse("Hello", 3);
        reverse("Hello", 4);
```



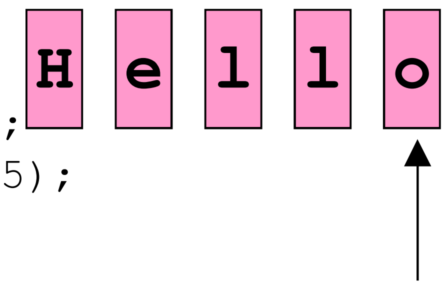
Third type of recursion

```
reverse("Hello", 0);
  reverse("Hello", 1);
    reverse("Hello", 2);
      reverse("Hello", 3);
        reverse("Hello", 4);
          reverse("Hello", 5);
          return ""
```



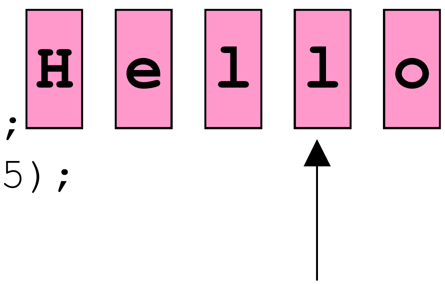
Third type of recursion

```
reverse("Hello", 0);
  reverse("Hello", 1);
    reverse("Hello", 2);
      reverse("Hello", 3);
        reverse("Hello", 4);
          reverse("Hello", 5);
          return ""
        return "o"
```



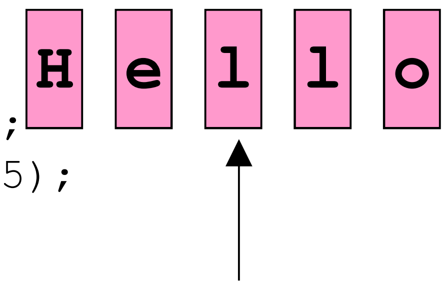
Third type of recursion

```
reverse("Hello", 0);
  reverse("Hello", 1);
    reverse("Hello", 2);
      reverse("Hello", 3);
        reverse("Hello", 4);
          reverse("Hello", 5);
          return ""
        return "o"
      return "ol"
```



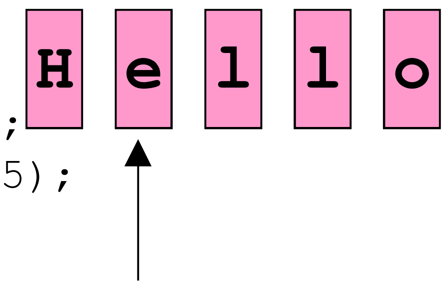
Third type of recursion

```
reverse("Hello", 0);
  reverse("Hello", 1);
    reverse("Hello", 2);
      reverse("Hello", 3);
        reverse("Hello", 4);
          reverse("Hello", 5);
          return ""
        return "o"
      return "ol"
    return "oll"
```



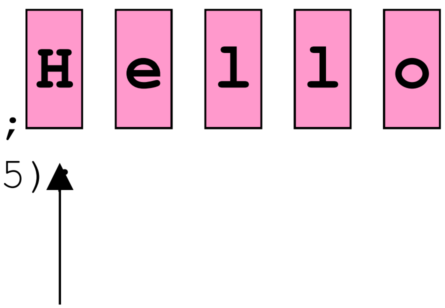
Third type of recursion

```
reverse("Hello", 0);
  reverse("Hello", 1);
    reverse("Hello", 2);
      reverse("Hello", 3);
        reverse("Hello", 4);
          reverse("Hello", 5);
          return ""
        return "o"
      return "ol"
    return "oll"
  return "olle"
```



Third type of recursion

```
reverse("Hello", 0);
  reverse("Hello", 1);
    reverse("Hello", 2);
      reverse("Hello", 3);
        reverse("Hello", 4);
          reverse("Hello", 5);
          return ""
        return "o"
      return "ol"
    return "oll"
  return "olle"
return "olleH"
```



Another example – Searching in an array

- Simple linear search

```
int iterLinearSearch(float array[], float target)
{
   int index = 0;
   while (index < array.length())
   {
      if (array[index] == target) {return index;}
      index++;
   }
   return -1;
}

// function receives an array of integers and a
// target value. Returns the position of the first
// occurrence of the target, or -1 if the target is
// not present.
```

Another example – Searching in an array

- Iterative binary search

```
int iterativeBinarySearch(float array[], float target)
{
   int middle, first, last;
   first = 0; last = array.length()-1;
   while (first <= last)
   {
      middle = (first + last) / 2;
      if (array[middle] == target) {return middle;}
      else if (target < array[middle]) {last = middle - 1;}
      else if (target > array[middle]) {first = middle + 1;}
   }
   return -1;
}

// function receives an array of integers and a
// target value. Returns the position of the first
// occurrence of the target, or -1 if the target is
// not present. The array must be ordered!
```

Another example – Searching in an array

- Recursive binary search

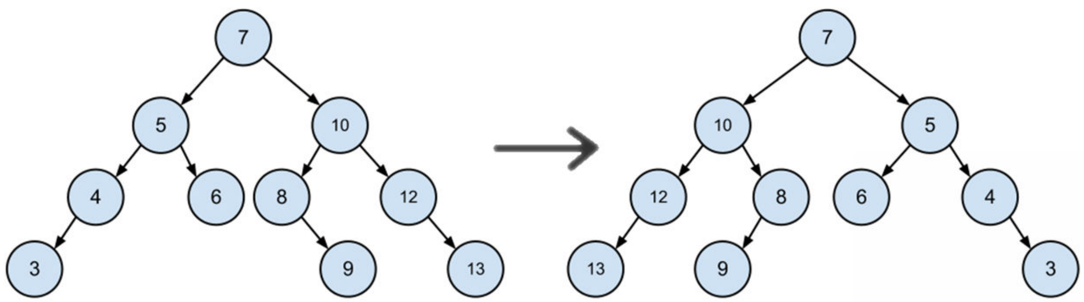
```
int recursiveBinarySearch(float array[], int first, int last, float target)
{
   int middle = (first + last) / 2;

   if (first <= last && array[middle] != target)
   {
      if (target < array[middle])
         middle = recursiveBinarySearch(array, first, middle-1, target);
      else if (target > array[middle])
         middle = recursiveBinarySearch(array, middle+1, last, target);
   }
   if (target == array[middle]) {return middle;} else {return -1;}
}

// function receives an array of integers and a
// target value. Returns the position of the first
// occurrence of the target, or -1 if the target is
// not present. The array must be ordered!
```

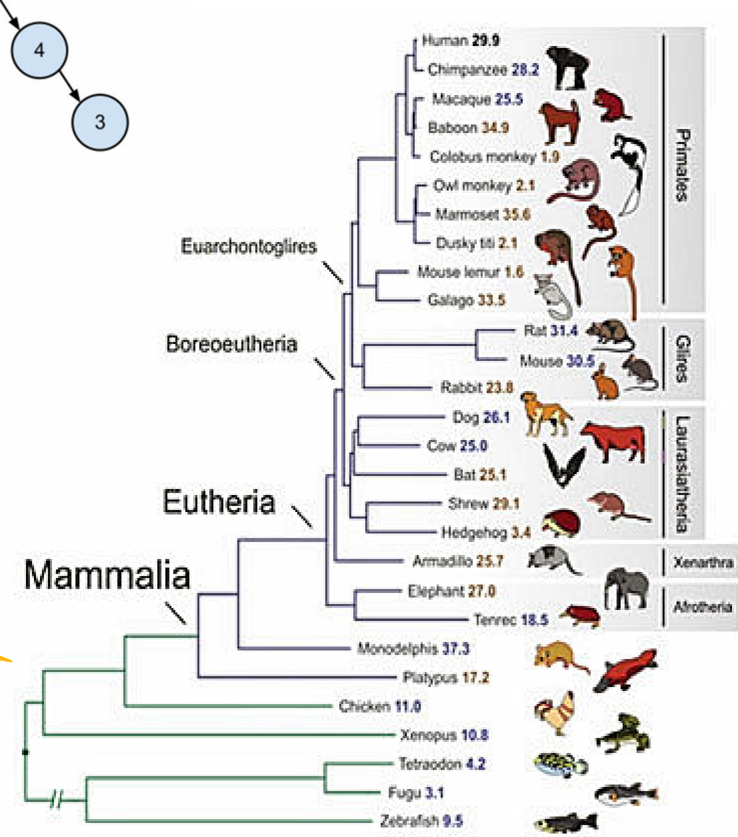
Another example – Inversion of trees

- Inversion of trees



Binary search trees

Phylogenetic trees



See you next week!