

cis15  
advanced programming techniques, using c++  
lecture # VI.1

topics:

- recursion
- searching

recursion

- recursion is defining something in terms of itself
- there are many examples in nature
- and in mathematics
- and in computer graphics, e.g., the Koch snowflake

power function

- *power* is defined recursively:  $x^y = \begin{cases} \text{if } y == 0, & x^y = 1 \\ \text{if } y == 1, & x^y = x \\ \text{otherwise,} & x^y = x * x^{y-1} \end{cases}$

here it is in C++

- ```
// r1.cpp
#include <iostream>
using namespace std;

int power( int x, int y ) {
    if ( y == 0 )
        return( 1 );
    else if ( y == 1 )
        return( x );
    else
        return( x * power( x, y-1 ) );
} // end of power()

int main() {
    cout << "2^3 = " << power( 2,3 ) << endl;
}

• Notice that power() calls itself!
```

- You can do this with any method *except* `main()`
- BUT beware of infinite loops!!!
- You have to know when and how to stop the recursion — what is the *stopping* condition

let's walk through `power( 2,4 )`

|     | call                      | x | y | return value                  |
|-----|---------------------------|---|---|-------------------------------|
| 1   | <code>power( 2,4 )</code> | 2 | 4 | <code>2 * power( 2,3 )</code> |
| • 2 | <code>power( 2,3 )</code> | 2 | 3 | <code>2 * power( 2,2 )</code> |
| 3   | <code>power( 2,2 )</code> | 2 | 2 | <code>2 * power( 2,1 )</code> |
| 4   | <code>power( 2,1 )</code> | 2 | 1 | 2                             |

- the first is the *original call*
- followed by three *recursive calls*

## stacks

- the computer uses a data structure called a *stack* to keep track of what is going on
- think of a *stack* like a stack of plates
- you can only take off the top one
- you can only add more plates to the top
- this corresponds to the two basic *stack operations*:
  - *push* — putting something onto the stack
  - *pop* — taking something off of the stack
- when each recursive call is made, `power()` is pushed onto the stack
- when each return is made, the corresponding `power()` is popped off of the stack

## another example: factorial

- *factorial* is defined recursively:
 
$$N! = \begin{cases} \text{if } N == 1, & N! = 1 \\ \text{otherwise,} & N! = N * (N - 1)! \end{cases}$$
- (for  $N > 0$ )

here it is in C++

```
• // r2.cpp
#include <iostream>
using namespace std;

int factorial ( int N ) {
    if ( N == 1 )
        return( 1 );
    else
        return( N * factorial( N-1 ));
} // end of factorial()

int main() {
    cout << "5! = " << factorial( 5 ) << endl;
}
```

recursive iteration

- You can also use recursion to iterate.
- Here's an example. Compare printI() (iterative) with printR() (recursive).

```
// r3.cpp
#include <iostream>
using namespace std;

class array {
private:
    int *data;
    int size;
public:
    array( int n ) : size( n ) { data = new int[n]; }
    void set( int x, int v ) { data[x] = v; }
    void printI();
    void printR( int x );
}; // end of class array
```

```
void array::printI() {
    for ( int i=0; i<size; i++ )
        cout << data[i] << " ";
    cout << endl;
} // end of printI()

void array::printR( int x ) {
    if ( x < size ) {
        cout << data[x] << " ";
        printR( x+1 );
    }
    else {
        cout << endl;
    }
} // end of printR()
```

```
int main() {
    array A( 5 );
    for ( int i=0; i<5; i++ )
```

```
        A.set( i,i*10 );
        cout << "output from iterative printI(): ";
        A.printI();
        cout << "output from recursive printR(): ";
        A.printR( 0 );
    } // end of main() method
```

and the output is:

```
output from iterative printI(): 0 10 20 30 40
output from recursive printR(): 0 10 20 30 40
```

### and the details...

- in the recursive version, each call is like one iteration inside the for loop in the iterative version

|   | call        | index | output | next call   |
|---|-------------|-------|--------|-------------|
| 1 | printR( 0 ) | 0     | 0      | printR( 1 ) |
| 2 | printR( 1 ) | 1     | 10     | printR( 2 ) |
| 3 | printR( 2 ) | 2     | 20     | printR( 3 ) |
| 4 | printR( 3 ) | 3     | 30     | printR( 4 ) |
| 5 | printR( 4 ) | 4     | 40     | printR( 5 ) |
| 6 | printR( 5 ) | 5     | endl   | (none)      |

- In the example above, each time `printR( i )` is called, the array is printed from the  $i$ -th element to the end of the array.
- In the `power(x,y)` example, each time the function is called, power is computed for each  $x^y$ , in terms of the previous  $x^{y-1}$ .
- In the `factorial(N)` example, each time the function is called, factorial is computed for each  $N$ , in terms of the previous  $N - 1$ .

### searching

- Often, when you have data stored in an array, you need to locate an element within that array.
- This is called searching.
- Typically, you search for a *key* value (simply the value you are looking for) and return its *index* (the location of the value in the array)
- As with sorting, there are many searching algorithms.
- We'll study the following:
  - linear search
    - \* standard linear search, on sorted or unsorted data
    - \* modified linear search, on sorted data only
  - binary search
    - \* iterative binary search, on sorted data only
    - \* recursive binary search, on sorted data only

### linear search on UNSORTED DATA

- Linear search simply looks through all the elements in the array, one at a time, and stops when it finds the key value.
- This is inefficient, but if the array you are searching is not sorted, then it may be the only practical method.

```
// s1.cpp
#include <iostream>
using namespace std;

class array {
private:
    int *data;
    int size;
public:
    array( int n ) : size( n ) { data = new int[n]; }
    void set( int x, int v ) { data[x] = v; }
    int getSize() { return size; }
```

```

int getItem( int x ) { return data[x]; }
void print( int x );
int linearSearch( int key );
}; // end of class array

void array::print( int x ) {
    if ( x < size ) {
        cout << data[x] << " ";
        print( x+1 );
    }
    else {
        cout << endl;
    }
} // end of array::print()

int array::linearSearch( int key ) {
    for ( int i=0; i<getSize(); i++ ) {
        if ( key == getItem( i ) ) {
            return( i );
        }
    }
}

```

```

    } // end for i
    return( -1 );
} // end of array::linearSearch()

int main() {
    int x;
    array A( 5 );
    for ( int i=0; i<5; i++ )
        A.set( i,i*10 );
    cout << "here is the array: ";
    A.print( 0 );
    cout << "looking for item 30...";
    x = A.linearSearch( 30 );
    if ( x == -1 )
        cout << "not found\n";
    else
        cout << "found at location: " << x << endl;
    cout << "looking for item 65...";
    x = A.linearSearch( 65 );
    if ( x == -1 )

```

```

    cout << "not found\n";
    else
        cout << "found at location: " << x << endl;
} // end of main() method

```

and the output is:

```

here is the array: 0 10 20 30 40
looking for item 30...found at location: 3
looking for item 65...not found

```

### linear search on SORTED data

- If the array you are searching IS SORTED, then you can modify the linear search to stop searching if you have looked past the place where the key would be stored if it were in the array.
- This only helps shorten the run time if the key is not in the array...
- Below is an example that works when the array is sorted in ascending order (from smallest to largest).  
Note that it also counts and prints out the number of times the for loop is called, so that you can compare the number of iterations for this version as opposed to the previous linearSearch() function.

```

int array::linearSearchSorted( int key ) {
    int count = 0;
    for ( int i=0; i<getSize(); i++ ) {
        count++;
        if ( key == getItem( i ) ) {
            cout << "count=" << count << endl;
            return( i );
        }
    }
}

```

```

    }
    else if ( key < getItem( i )) {
        cout << "count=" << count << endl;
        return( -1 );
    }
} // end for i
    cout << "count=" << count << endl;
return( -1 );
} // end of array::linearSearchSorted()

```

and the output is:

```

here is the array: 0 10 20 30 40
looking for item 30 using linearSearch()...count=4
found at location: 3
looking for item 15 using linearSearch()...count=5
not found
looking for item 30 using linearSearchSorted()...count=4
found at location: 3
looking for item 15 using linearSearchSorted()...count=3
not found

```

## binary search

- Binary search is much more efficient than linear search, BUT ONLY WORKS ON A SORTED ARRAY.
- It takes the strategy of continually dividing the search space into two halves, hence the name *binary*.
- Say you are searching something very large, like the phone book. If you are looking for one name (e.g., "Gilligan"), it is extremely slow and inefficient to start with the A's and look at each name one at a time, stopping only when you find "Gilligan". This is what linear search does.
- Binary search acts much like you'd act if you were looking up "Gilligan" in the phone book.
  - You'd open the book somewhere in the middle, then determine if "Gilligan" appears before or after the page you have opened to.
  - If "Gilligan" appears after the page you've selected, then you'd open the book to a later page.
  - If "Gilligan" appears before the page you've selected, then you'd open the book to an earlier page.

- You'd repeat this process until you found the entry you are looking for.
- For example:

```

int array::binarySearch( int key ) {
    int count = 0;
    int lo = 0, hi = getSize()-1, mid;
    while ( lo <= hi ) {
        count++;
        mid = ( lo + hi ) / 2;
        if ( key == getItem( mid )) {
            cout << "count=" << count << endl;
            return( mid );
        }
        else if ( key < getItem( mid )) {
            hi = mid - 1;
        }
        else {
            lo = mid + 1;
        }
    }
} // end while

```

```

    cout << "count=" << count << endl;
    return( -1 );
} // end of array::binarySearch()

```

and the output is:

```

here is the array: 0 10 20 30 40
looking for item 30 using linearSearch()...count=4
found at location: 3
looking for item 15 using linearSearch()...count=5
not found
looking for item 30 using linearSearchSorted()...count=4
found at location: 3
looking for item 15 using linearSearchSorted()...count=3
not found
looking for item 30 using binarySearch()...count=2
found at location: 3
looking for item 15 using binarySearch()...count=3
not found

```

### recursive binary search

- Binary search lends itself very well to a recursive implementation.
- Here's the previous binary search re-written as a recursive function:

```
int array::recursiveBinarySearch( int key, int lo, int hi ) {
    if ( lo <= hi ) {
        int mid = ( lo + hi ) / 2;
        if ( key == getItem( mid ) ) {
            return( mid );
        }
        else if ( key < getItem( mid ) ) {
            return( recursiveBinarySearch( key, lo, mid-1 ) );
        }
        else {
            return( recursiveBinarySearch( key, mid+1, hi ) );
        }
    }
    else {
```

```
        return( -1 );
    }
} // end of array::recursiveBinarySearch()
```