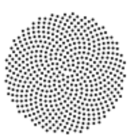
RECURSION

Today

- This lecture looks at
  - The basics of recursion.
  - Some examples of recursive functions.
- The textbook doesn't cover recursion in any detail (the only material is on pages 96 and 97 in my copy)..

Recursion

- Recursion is defining something in terms of itself
- There are many examples in nature:
  - Seeds in a sunflower

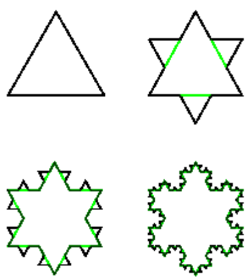


- . . . in mathematics:
  - Factorial
  - Induction
- . . . and in computer graphics:
  - Koch snowflake

Koch snowflake

- Starting with a line, then:
  1. Divide each line into three segments of equal length.
  2. Draw an equilateral triangle that has the middle segment from step 1 as its base and points outward.
  3. Remove the line segment that is the base of the triangle from step 2.
- Repeat as often as you like.

- Here are the first four iterations of the Koch snowflake.



- The more iterations, the more snowflaky it looks.

Power function

- *Power* is defined recursively:

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

- There are two parts to the definition:
  - The *base case*, what we do when y is zero.
  - The *recursive case*, what we do when y is not zero.
- This is the common pattern for all recursive definitions.

Here it is in C++

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

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

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


- Notice that `power()` calls itself!
- This seems to be magic, but we'll see how it is done in a moment.
- You can make recursive calls 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.

Walking through `power( 2,4 )`

- Initial call is `power(2, 4)`

	call	x	y	return value
1	power(2,4)	2	4	2 * power(2,3)
2	power(2,3)	2	3	2 * power(2,2)
3	power(2,2)	2	2	2 * power(2,1)
4	power(2,1)	2	1	2 * power(2, 0)
4	power(2,0)	2	0	1

- The first is the *original call*
- Followed by four *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;
}
```

- Walk through `factorial(4)`

Another example

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

void countDown (int n) {
  if ( n <= 0 )
    cout << "Blastoff!" << endl;
  else {
    cout << "Time to launch is " << n << " seconds" << endl;
    countDown(n - 1);
  }
} // end of countDown()


int main() {
  countDown(5);
}
```


- What is the output of this program?

- Now, let's switch the statements in the recursive case around.

```
//r4.cpp
#include <iostream>
using namespace std;

void countDown (int n) {
  if ( n <= 0 )
    cout << "Blastoff!" << endl;
  else {
    countDown(n - 1);
    cout << "Time to launch is " << n << " seconds" << endl;
  }
} // end of countDown()


int main() {
  countDown(5);
}
```

- What is the output of this program?

- Again `countDown` has the general structure:

```
// base case part

if (<base-case condition>)
   return <base-case-value>

// general case

else
   return <recursively computed expression>
```

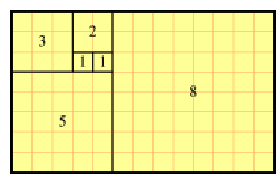
- This is common to all recursive functions — the only difference you'll see is that some functions have two base cases.

Fibonacci

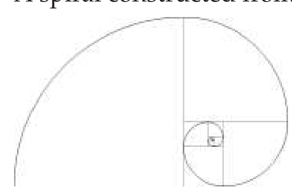
```
// in r5.cpp

int fibonacci (int n) {
  if (n == 0){
    return 0;
  }
  else
    if (n == 1){
      return 1;
    }
    else {
      return(fibonacci(n - 1) + fibonacci(n -2));
    }
} // end of fibonacci()
```


- – A tiling where tile sides are successive members of the Fibonacci sequence.



  – A spiral constructed from the above tiling.



Recursive and iteration

- You can use recursion to iterate.
- (Iteration = repetition, what you would normally do with a loop).
- The following slide has an example.
- Compare `printI()` (iterative) with `printR()` (recursive).


```
// recursion-iteration.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)*

- With recursion, each time the function is invoked, one step is taken towards the resolution of the task the function is meant to complete.
- Before each step is executed, the state of the task being completed is somewhere in the middle of being completed.
- After each step, the state of the task is one step closer to completion.
- 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 $N-1$.

Summary

- This lecture has looked at
  - The basic idea of recursion
  - A bunch of different examples of recursion
- You will find that the idea of recursion gets easier to cope with as you get more familiar with it.
- That means, like all programming ideas, it gets easier with use.