Recursion

- Recursion is a computer programming technique that allows programmers to divide problems into smaller problems of the same type.
- You can think of it as a “divide and conquer strategy” for programming.
- Classic examples:
  - Raising to a Power function.
  - Finding Fibonacci sequence numbers.

What is recursion exactly?

- Recursion is a method for solving problems by defining functions which call themselves.
- If a function has a point in which it calls another instance of itself, it is considered recursive.

```
// A function which takes a double value x
// and raises it to the non-negative power y
double power(double x, int y){
   if(y == 0)
      return 1;
   else
      return x*power(x,y-1);
}
```

What tasks are appropriate for recursion?

- Any task which can be broken down into smaller subparts of the same problem.
  - The power example: x^y = x*x^(y-1)
  - GCD: The greatest common divisor can be found by performing gcd on mathematical derivatives of the original numbers.
- It is often easier to write these sorts of problems in a recursive manner than in an iterative (that is, without having a function call itself) manner.
  - Any problem that can be solved recursively can be solved iteratively!

What happens with recursion?

- Recall the stack we have talked about during class.
- Each time you call the recursive function, a new set of information (the call frame) is placed on the stack, and computation begins.
- When the function completes, the call frame and associated variables are removed, and the previous call frame continues where it left off.

Advantages to recursive programming

- Simpler code
- Frequently, it is more understandable – if you've programmed well!
- Often, it is a good way to lay the framework to make an iterative function.

Disadvantages to recursive programming

- Usually slower to execute than iterative methods
- Tend to require more space
- Stack overflow potential!

An example void function, recursively

- Let's print a number vertically:

```
void printVertical(int n){
   if(n < 10)
      cout << n << endl;
   else{
      printVertical(n/10);
      cout << n % 10 << endl;
   }
}
```

- How does this work?

An example of recursive non-void functions

- How can we return the factorial of a number?

```
// Function to compute n!
// n*n-1*...*2*1
int factorial(int n){
   if(n <= 1)
      return 1;
   else
      return n*factorial(n-1);
}
```

How recursion works, specifically

- We already know: every time a function is called, information about it is placed on a stack.
- Technically, the computer will let you continue to call functions this way until you run out of space.
- So, your function must have a case that is a stopping point – a “base case.”
- This stopping case will then (if necessary) be returned up the chain of recursive calls.
- If you don't have an appropriate stopping case, you get infinite recursion!

Designing a recursive function

- Three properties to satisfy
  - No infinite recursion – does your program eventually reach a base case?
  - Are your base cases correct?
  - Are recursive cases performed properly?
- If you can satisfy those properties, your function will behave properly.

Going from recursive to iterative

- In recursive functions, we generally work from an unknown answer to a known smaller answer, and then use that smaller answer to solve the larger problems.
- Sometimes this can translate directly to an iterative solution.
- Sometimes, you may want to think of it backwards – work from the known solution up!
- The main thought should be – how can this be done within a loop? What do I need to know in that loop?

Trying to make your own recursive function

- Let's stop here and let you try to write your own fibonacci sequence functions.
- Working with someone else, write a full function that is recursive.
- The function should take one integer parameter – the fibonacci number to look for (i.e. the 2nd fibonacci number, the 3rd, etc.) and return the integer that has the value of that fibonacci number.
- Once you've finished – think how you might make the function iterative.

Separate Compilation

- We have many times discussed the notion of encapsulation:
  - Separation of interface and implementation.
- How can we reflect this better in our coding?
  - C++ compilers have methods which allow you to put different parts of code in different files, then recompile them together.

How should you divide your code?

- Typically, you ought to place each class you create in a separate file set. You will have your header file (the interface), and your implementation file (function definitions).
- The header file will be a file ending with the suffix “.h” instead of the usual “.cc” or “.cpp.”
- This header should contain the interface to your code – this is your class definition, and the prototypes of any non-member function you want them to have access to.
  - Technically we would like to keep them from seeing any private members of the class, but that isn't possible.

How do I use this header file?

- The #include directive.
  - When you say `#include <iostream>`, the compiler understands that you mean you want it to place the code from the iostream file in this spot.
  - The <> brackets tell the compiler to look for this information in the location used for pre-defined headers.
  - You will use `#include "myClass.h"` to include your code. This tells the compiler to look in the local directory (or a pre-defined location for programmer-defined headers) for the file.

#ifndef

- Since the include directive simply places code into your program, you don't want it to place the same code over and over – this will lead to errors! However, you may need to indicate to the compiler that you are using your header in many different files.
- How do you make it so it only processes your definition once?
- You use a technique that tells the compiler not to process your header again if it has already been processed once.

#ifndef

- Enclose your header like this:

```
#ifndef _MYHEADER_H_
#define _MYHEADER_H_
<header file code>
#endif
```

- This can be thought of as telling the compiler
  - If myHeader.h has not been seen yet
  - Define myHeader.h
  - Include the code up until endif.
- Really, you're just putting _MYHEADER_H_ on a list that says “this keyword has been seen.” But it does what it looks like, in a way.

How do I define the functions?

- Write your implementation file for “myClass.h”. This should be named “myClass.cpp” and use `#include “myClass.h”` at the beginning.
- The implementation file will have all the function definitions from function declared in the header file, as well as prototypes and definitions for any functions that aren't part of the interface.

How do I compile all of this?

- Now to compile, instead of compiling one file with all the definitions directly, you will compile in stages.
- First you will need to compile each implementation file into an 'object file.'

```
CC myClass.cpp -c -o myClass.o
CC myClass2.cpp -c -o myClass2.o
...
```

- Then, compile your application file as an object.

```
CC myProgram.cpp -c -o myProgram.o
```

How do I compile all of this?

- Finally, compile your program by letting the compiler link all your object files and make your final program:

```
CC myProgram.o myClass.o myClass2.o -o myProgram
```

- How to think of compilation and object files:
  - What the compiler does is create a series of files which may be incomplete, and lets them be prepared to be linked with other object files which will fill in the missing spots.

Why bother?

- What are the advantages of this?
  - Makes your code separated according to encapsulation principles.
  - Makes your code easier to read – think of your header file as providing the instructions on how to use the class.
  - Makes reuse of classes you design MUCH easier.
    - No more cutting and pasting definitions into new files!

Doing all this compilation is annoying.

- There is a useful tool called make which allows you to define instructions on how to compile a program.
- Make reads a Makefile, and proceeds through it so it can run commands you tell it to. It will also check to see if your program is out of date, and recompile if needed.
- To run make, you need to create a Makefile.

Writing a makefile

- The file must be named Makefile.
- It follows a certain syntax:
  - At the top, place comments such as who you are and when the file was written. Use `#` like `//`.
  - Define any variables you might want. Just place on a single line `<var_name>=<string>`. You can access these variables in the make file using `${var_name}`
  - Dependencies: These are the heart of a Makefile. They use this format:

```
<dependecy_name>: <other_d1> <other_d2>...
   <command 1>
   <command 2>
   ...
```

Understanding dependencies

- The idea is somewhat simple: you define a dependency by “naming” a series of commands (<dependency_name>).
- Before the commands can be executed by make, it must make sure the other dependencies (<other_d1>,<other_d2>) are run first, or are already up to date.
- Then, it runs the commands you listed. Each command MUST be on a separate line with a SINGLE TAB IN FRONT.

Using dependencies

- Think of your Makefile as a mini-program which runs your compilations for you.
- The dependencies say which steps should be done first.
- Note: you should always have the first dependency listed be the dependency you want completed at the end. You can specify targets, but if you don't, this is the one that will be run.

Midterm Exam Review

- Topics to expect on the exam (not a complete list!):
  - C++ basics: data types and basic input/output, basic terminology, #include, namespaces
  - Flow control: loops, how they work, when to use them
  - Functions: how they work, using parameters, understanding scope, returning values
  - Parameters: Call-by-Value vs. Call-by-Reference, overloading functions
  - Arrays: Declaring arrays, multi-dimensional arrays, using arrays in functions
  - Structures: How to define them, what their purpose is, how to use them and access them
  - Dynamic memory: Heap vs. Stack, Pointers, allocating memory, creating dynamic arrays, delete, new, linked lists