Recursion

1320-Intermediate Programming

University of Texas at Arlington

Lecture Overview

- Lecture
  - Patterns -> Code
  - Same Problem, Different Implementation
- Before We Code
  - Header files
- Sample Programs

Patterns->Code

Let's take a look at the following sequence of numbers (known as the Fibonacci Sequence):

1 1 2 3 5 8 13 21 34 55…

As a computer scientist, the question is (as always):
How can we turn this into code?

Patterns->Code

General process to take:

1. Identify what is going on
   - There is a pattern of some sort
2. How do we represent this pattern? Can we represent it mathematically?
   - Yes
3. How can we turn this representation into code?
   - I need to decide-what would I want my program to **actually do**?
     - I want to enter the spot and return the value in that spot
   - How do I want to do it? What technique should I use?

1 1 2 3 5 8 13 21 34 55…

Patterns->Code

1. Identify what is going on
   - There is a pattern of some sort:

Notice that all numbers (after the first two) are produced by adding the two preceding numbers

Patterns->Code

2. How do we represent this pattern? Can we represent it mathematically?
   - Yes

1	**1**	**2**	**3**	**5…**
1st number	2nd number	3rd number	4th number	5th number

Let's let ***n*** represent the number's place in the sequence. For example:

- So ***n=3*** means we are talking about the third number (so **2**)
- The number **3** means we are talking about ***n=4***

Patterns->Code

- We can say for any number (except the first two):
  - The value is equal to the sum of the two previous numbers
  - Number (in spot **n**) = Number (in spot **n-1**)+ Number (in spot **n-2**)

Number = **Number** + **Number**

n **n-1** **n-2**

3. How do we turn this representation into code?

- What do we want our code to do?
  1) We could have it compute a Fibonacci number
     - We could give it the location, n, and have it compute the Fibonacci number: Given n=3, it computes 2

1	**1**	**2**	**3**	**5** …
n=1	n=2	n=3	n=4	n=5

  2) We could have it figure out where in the sequence a Fibonacci number is
     - We could give a Fibonacci number and have it calculate the location, n: Given 3, it computes 4 (n=4)

1	**1**	**2**	**3**	**5** …
n=1	n=2	n=3	n=4	n=5

1	1	2	3	5…
1st number	2nd number	3rd number	4th number	5th number

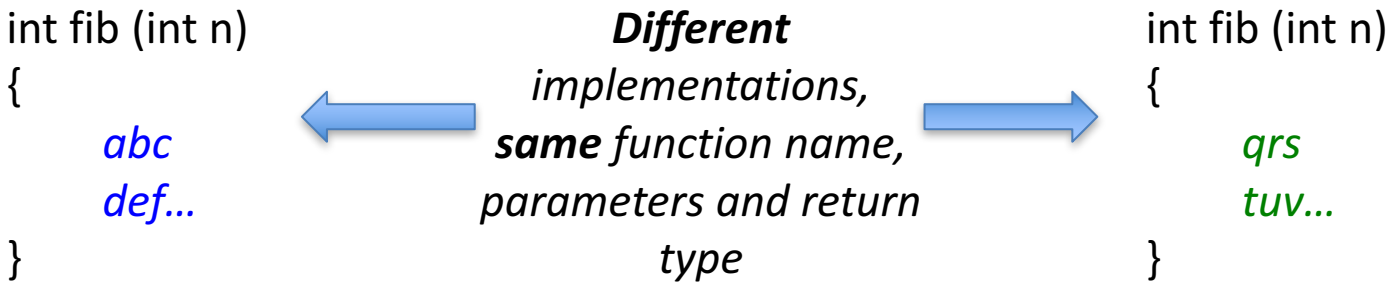
In our code today, we will do the first option (given a location, compute the Fibonacci number) by creating a function. We can give the value of n (the location of a number in the sequence) as a parameter and let the return value be the number in the Fibonacci sequence.

A function declaration might look like this: **int fib(int n);**

A line of code might be: **int answer=fib(3);**
The value of answer would be 2.

Implementation

- So now, we know what we want our code (put into a function) to do.
- The question is now HOW do we want to do it? HOW should we implement it?
  - What technique should we use? The function name, parameters and return type will stay the same, but the actual implementation can differ:



Implementation

- This is an important concept in computing- when we use a function, we don't necessarily care how it was implemented, we care if it does what we want it to do or not in an efficient manner
  - Note: we obviously want the best implementation possible (space and time constraints)
- When we use *printf* for example, we just use it (and not worry about the implementation)

Implementation

- Our declaration:
  - **int fib(int n);**
  - We don't have the details about implementation, but we just know if we give an int (the location), we get back an int (the value)

- Our definition is our actual implementation
  - We can have different implementations
  - The benefit of this is that even if we change the implementation, it does not affect the user-they can still call the function without worrying about the change in the code
    - If we find a better implementation, we can use it

Recursion

- I will show you two possible implementations of all the programs today
  - Recursion (a function calling itself)
  - Iteration (loops)

- You can think of recursion as a technique where a function continuously calls itself until some base case is reached
- A general rule is that if you can solve a problem iteratively, you can also solve it recursively

Recursion

- I will put each implementation into a separate function, but note that you could have one function and just change the implementation inside

- I will be drawing visual representations of recursion (trees and the stack) on the board- take notes if you are not familiar with recursion!

BEFORE WE CODE

Before We Code

- We will declare functions in a header file today(.h)
- We will define them in a separate .c file
  - Now, we won't have all our functions cluttering up the same file with the main function

SAMPLE PROGRAMS

Programs 1-3

- I will show the following individual programs:
  - Fibonacci
  - Factorial
  - Decimal to Binary
- I will show two possible implementations using:
  - Recursion
  - Iteration

Program 4

- Now I will declare all functions in a header file and put the definitions in a .c file
- Notes about the use of #include preprocessor directive (using <file> vs “file”) https://gcc.gnu.org/onlinedocs/cpp/Include-Syntax.html
- See next slide for more info

These are called header guards-these make sure contents are not included twice (in the case two files use the same header). Always include them.

myfirstheader.h

```
#ifndef FIRST_HEADER
#define FIRST_HEADER


int ages(int a, int b); /*function declaration*/

#endif
```

Function declarations here

myfirstheader.c

```
/*implementation*/
#include "myfirstheader.h"

int ages(int a, int b)
{
        return a+b;

}
```

Function definitions here

prac1.c

```
#include <stdio.h>
#include "myfirstheader.h"

int main()
{
        int a1=15;
        int a2=34;

        int total=ages(a1,a2);

        printf("Total: %d\n", total);

}
```

Actual program