LECTURE 3:
RECURSION

COMP1002J: Introduction to Programming 2

Dr. Brett Becker (brett.becker@ucd.ie)
Beijing Dublin International College

A little about me

- Originally from New Jersey, USA
- Living in Dublin for 17 years
- BA Physics, Drew University (NJ, USA)
- BA Computer Science, Drew University (NJ, USA)
- MSc Computational Science, University College Dublin
- PhD Computer Science, University College Dublin
- MA Higher Education, Dublin Institute of Technology
- Certificate in University Teaching and Learning, University College Dublin

A little about me

- Teaching for 14 years
- Lecturer / Senior Lecturer, Griffith College Dublin
- Head of Faculty of Computing, College of Computing Technology, Dublin, Ireland
- Assistant Professor, School of Computer Science, University College Dublin & Beijing Dublin International College, 2015 - present

A little about me

- Research Interests
  - **Computer Science Education**
    - Novice Compilation Behaviour
    - Naturally Accumulating Programming Process Data
  - High Performance Computing
    - Parallel Computing
    - Heterogeneous Computing

A little about me

- Last week I was at the Association of Computing Machinery (ACM), Special Interest Group on Computer Science Education, Technical Symposium
  - There were almost 2,000 Computer Science educators there
  - Minneapolis, Minnesota, USA
  - http://sigcse2019.sigcse.org/
  - I presented three papers:

1. Prather, J.; Pettit, R.; **Becker, B.A.**; Denny, P.; Loksa, D.; Peters, A.; Albrecht, Z. and Masci, K. *First Things First: Providing Metacognitive Scaffolding for Interpreting Problem Prompts*. Proceedings of the 50th ACM Technical Symposium on Computer Science Education (SIGCSE 2019), Minneapolis, Minnesota, USA, February 2019. ACM. ***Best Paper, CS Education Research Track***

A little about me

2. **Becker, B.A.** and Quille, K. *50 Years of CS1 at SIGCSE: A Review of the Evolution of Introductory Programming Education Research*. Proceedings of the 50th ACM Technical Symposium on Computer Science Education (SIGCSE 2019), Minneapolis, Minnesota, USA, February 2019. ACM. ***SIGCSE Technical Symposium 50th Celebration Submissions***
3. **Becker, B.A.** and Fitzpatrick, T.* *What Do Syllabi Reveal About Our Expectations of Introductory Programming Students?* Proceedings of the 50th ACM Technical Symposium on Computer Science Education (SIGCSE 2019), Minneapolis, Minnesota, USA, February 2019. ACM.

* Thomas was a UCD undergraduate student when we wrote the paper. He is now a UCD PhD student.

A little about me

- I am the maintainer of the Irish Supercomputer List
  - www.IrishSupercomputerList.org

- For more on me and my research: www.brettbecker.com

Module Timetable

- **Lectures:**
  - Weeks 1-12, 14-15
  - **Mondays** @ 13:30-15:05, Room 102, Teaching Building 4
- **Labs**
  - Weeks 3*-15
  - **Wednesdays** @ 13:30-15:05, Room 102, Teaching Building 4

- Labs start this Wednesday!
- Please install MinGW and Notepad++
  - http://www.mingw.org/
  - https://notepad-plus-plus.org/

Moodle

- There are only 89 / 120 students enrolled on moodle. Please enroll now!
  - https://csmoodle.ucd.ie/moodle/user/index.php?id=772

Introduction - Recursion

- Most of the programs we have looked at so far are generally structured as functions or procedures that call one another in a disciplined, hierarchical manner.
- But it might be useful to have functions call themselves.
  - What? A function that calls itself?

```
int func(int x){
     int y;
     //do something
     func(y);
}
```

  - What is going on here?

Introduction - Recursion

- A **recursive** function is a function that calls itself *either directly or indirectly* (through another function).

```
int func(int x){
    int y;
    //do something
    func2(y);
}
```

- Above, `func2` might call `func`, so the above might be recursive, even if it doesn't look like it is!
- Recursion is a complex topic! Here we will look at some simple examples.

What is recursion?

- A recursive function calls itself indirectly or directly. Here we will just consider direct recursion.
- The function only knows how to solve the simplest case(s), or so-called *base-case(s)* of the problem at hand.
  - If the function is called with a base case it simply returns the result.
  - If the function is called with a more complex case then it divides the function into:
    - a piece it knows how to do
    - a piece it does not know how to do
- To make recursion feasible, the second piece (the piece it does not know how to do) must resemble the original problem, but be a slightly simpler or slightly smaller version of the original problem.

The Recursion Step

- Because this new problem looks like the original problem the function calls a copy of itself to work on the smaller problem.
    - This is called the *recursion step*
- The recursion step can result in many more recursive calls as the function keeps dividing each problem it is called with into two smaller problems.
- Each time the function calls itself with a slightly simpler version of the original problem.

Recursion

- Eventually, the smaller piece is so small, that it can be solved. We call this the *base case*.
- When the base case is reached, the function solves the base case and returns control to the calling function (the prior recursive call).
- This process is repeated. The calling function (which was a different copy of the same function) receives control. Then the next function ‘up’ receives control… this keeps happening ‘all the way up’ until the <u>original</u> call can return the final result.
- **It is imperative that we ensure that the recursion terminates!**
- **Thus, this sequence of smaller problems must eventually <u>converge</u> on the base case.**

Recursion

- Take the example of writing a function to calculate the factorial of a nonnegative integer *n*, written *n!*
- The factorial of a number is defined as follows
  - Fact(n) = n * (n-1) * …* 1
  - E.g. Fact(3) = 3 * 2 * 1 = 6
- The key to solving this recursively, is to realise:
  - Fact(3) = 3 * Fact(2)
  - Smaller problem than original problem, but similar to the original problem!

Recursion

- A lot of real-life problems are structured this way.

Non-recursive solution

- A non-recursive (iterative) version of this can be written as follows:

Factorial(n)
  fact = 1
  for i = 2 **to** n
      fact = fact * i
  endfor
  return fact
endalg

Fact_Iter.c
Fact_Function.c

Recursive Approach

- Can we take a recursive approach to this?

	5!	=	5 * 4 * 3 * 2 * 1
		=	5 * 4!
and	4!	=	4 * 3!
and	3!	=	3 * 2!
and	2!	=	2 * 1!
and	1!	=	1

Original problem

- From this we can see that n! = n * (n-1)!

Slightly smaller problem (but similar)

```
Factorial(num)
  if num == 1 then
        return 1
  else
        return num * Factorial(num – 1)
  endif
endalg
```

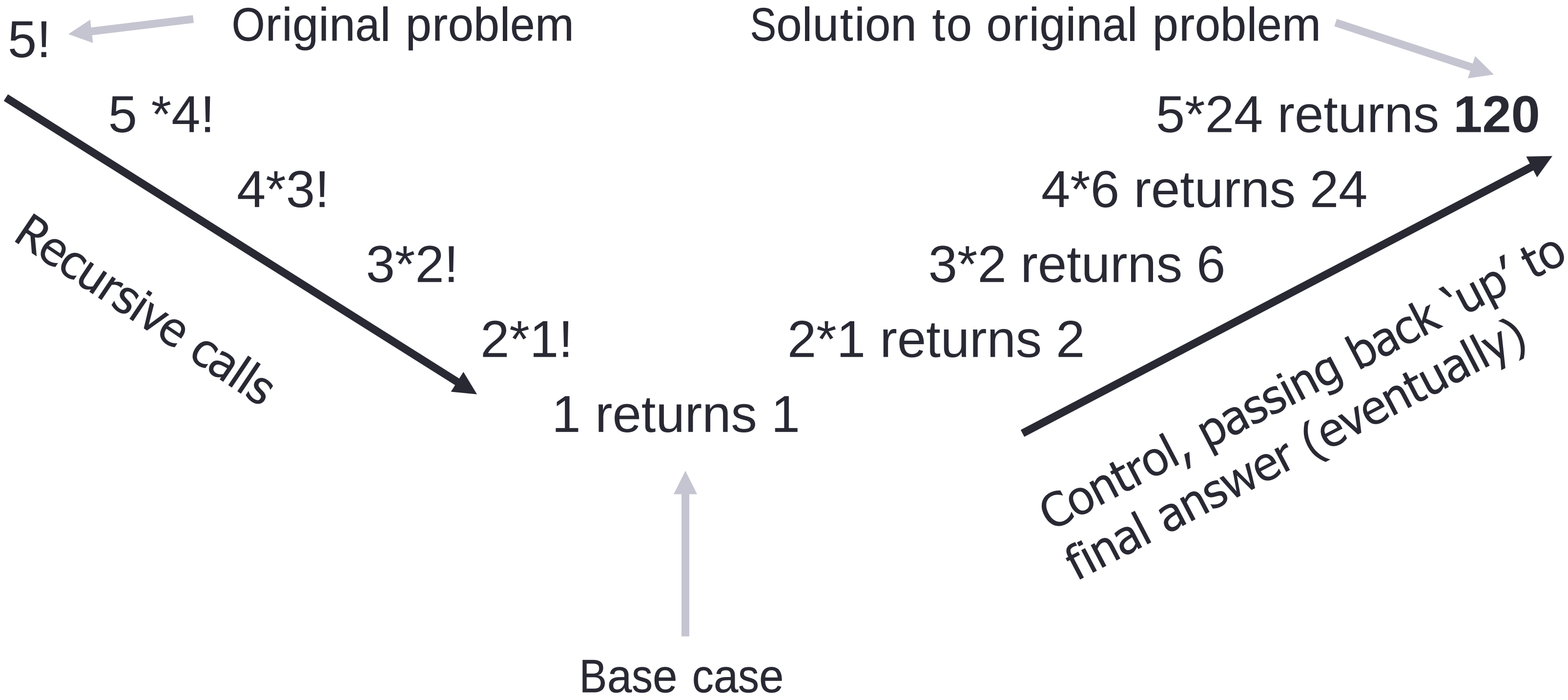
Base case

Recursive step (Recursive call)

Fact_Recur.c

Discussion

- What is happening is the following: (work from top left down, then from the bottom to the top right)



Problems

- The main problem which tends to occur when using recursion has to do with the base case.
  - Either the base case is accidentally omitted, or the recursion step is written so that it does not converge on the base case.
  - Both will cause infinite recursion, analogous to an infinite loop, which will eventually exhaust memory.
- The difference between infinite recursion and an infinite loop is that infinite recursion will quickly lead to a stack overflow because the recursion consumes memory quickly.
  - All of the ‘intermediate results’ and ‘intermediate function calls’ on the previous slide are what consumes the memory

Recursion v Iteration

- **Any function that can be written recursively can also be written iteratively. NOT every iterative function can be written recursively.**
- So, why would a programmer choose one method over another?
  - We'll answer this question slowly…
- Both iteration and recursion are based on a control structure.
  - Both iteration and recursion utilize repetition.
    - iteration <u>explicitly</u> uses a repetition structure;
    - <u>recursion achieves repetition through repeated function calls.</u>

- <u>Interesting question: what is the difference between iteration and recursion?</u>

Recursion v Iteration

- Iteration and recursion each involve a termination test.
  - iteration terminates when the loop-continuation condition fails
  - recursion terminates when a base case is reached.
- Iteration with counter-controlled repetition and recursion each gradually approach termination.
  - iteration keeps modifying a counter until the counter assumes a value that makes the loop condition fail
  - **recursion keeps producing simpler versions of the original problem until the base case is reached.**

Recursion v Iteration

- Let's look at the example of the factorial function and identify in both the iterative and recursive function the element mentioned above.
- Iterative Factorial Algorithm Recursive Factorial Algorithm

```
Iter-Fact(n)
    fact = 1
    for i = 1 to n
        fact = fact * i
    endfor
    return fact
endalg
```

```
Rec-Fact(n)
    if n == 1 then
        return 1
    else
        return n * Rec-Fact(n-1)
    endif
endalg
```

Counter controlled iterative loop

Base case

Recursive step

So Which Way is Best?

- So which way is best?
  - No simple answer!
- Recursion has many negative aspects. It repeatedly invokes function calls, and therefore incurs increasing overhead.
- This can be expensive in both processor time and memory space.
- Each recursive call causes another copy of the function (actually only the variables in the function) to be re-created. This can consume considerable amounts of memory. Essentially recursion can leave a lot of copies of variables around.
- Iteration normally occurs within a function so the overhead of repeated function calls and extra memory assignment is omitted.

Which is Best?

- However, it is often the case that a recursive approach more naturally mirrors the problem
- This can result in a program that is easier to understand and debug.
- Which of the factorial functions given above do you think is the neater solution to the problem?
- Often it is a matter of programmer taste.
- Another reason to choose a recursive solution is that an iterative solution may not be apparent.

Using Recursion

- The fibonacci series of numbers is,

  0, 1, 1, 2, 3, 5, 8, 13, 21, ...
- This series has the property that each subsequent Fibonacci number is the sum of the previous two Fibonacci numbers.

  13 = 5 + 8 21 = 8+13
- The ratio between successive fibonacci numbers converges to the value 1.618…
- This value also occurs in nature and is known as the *golden ratio* or *golden mean*.
- The fibonacci series may be defined recursively as follows:
  - fib(0) = 0, fib(1) = 1
  - fib(n) = fib(n-1) + fib(n-2)

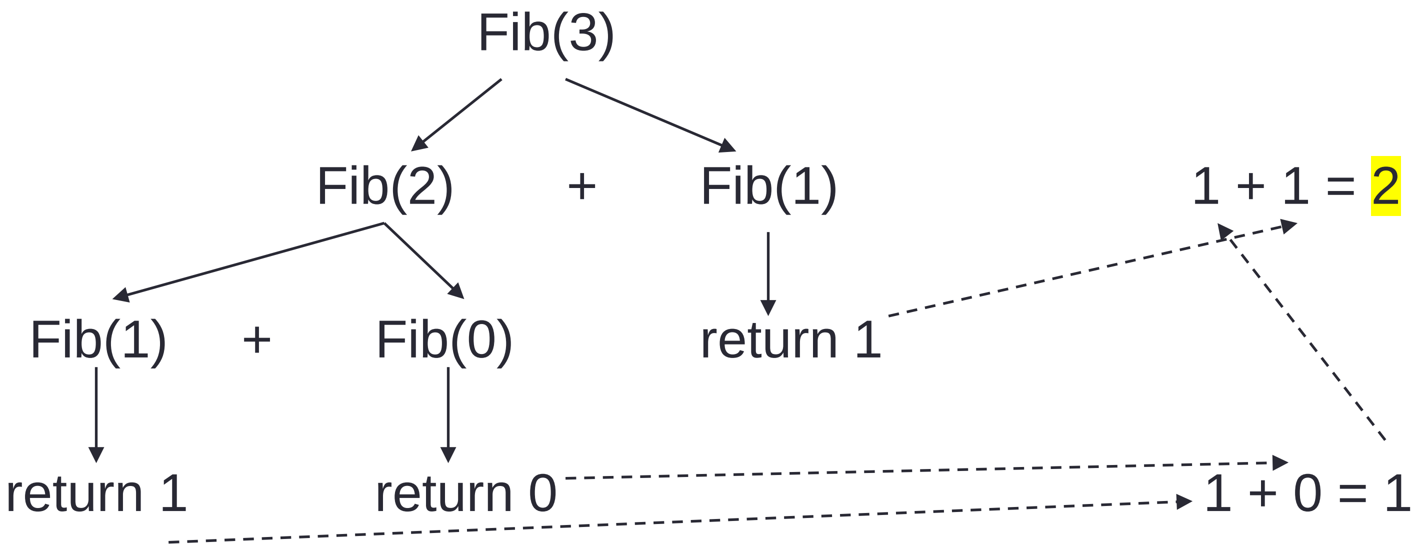
Fibonacci

```
Fib(n)
        if n = 0 or n = 1 then
            return n
        else
            return Fib(n-1) + Fib(n-2);
        endif
endalg
```

- When the algorithm is called it immediately checks for the base case. If the base case occurs then *n* is returned.
- If not, *two* recursive calls are made to the function, each a simpler version of the original problem.

Fibonacci

So, for Fib(3) we would get,



So, Fib(3) = 2. ☺

This required 5 function calls, and 5 variables.

Discussion

- A word of caution about recursive programs like the one we use here to generate Fibonacci numbers:
  - Each level of recursion in the fibonacci function has a doubling effect on the number of calls.
  - Calculating the 20th Fibonacci number would require on the order of about a million calls.
  - Calculating the 30th Fibonacci number would require around a billion calls!
- This is referred to as *exponential complexity*.
  - Sometimes this can be avoided by using *tail-recursion*. However we will not explore this topic.
- **How would you write this iteratively?**

Summary

- A recursive function is a function that calls itself either directly or indirectly through another function
- A recursive function divides a problem into:
  - a piece it knows how to do
  - a piece it doesn't know how to do
- The first is known as the base case, i.e. a simple version of the problem
- The second piece must resemble the original problem, but be a slightly simpler, or smaller, version of the original
- Any recursive function can be written iteratively
  - NOT every iterative function can be written recursively!
- Beware of exponential complexity!