CM10227/ CM50258: Lecture 2

Dr Rachid Hourizi and Dr. Michael Wright

October 9, 2017

Last Week

- The structure of this course
- The nature of programming
- First C Programs
  - Variables
  - Types
  - Pre-defined functions

This Week

- Begin writing our own functions
- Consider
  - Conditionals
  - Recursion
- And introduce the UNIX operating system

Resources

- General help on C
  - The C Book - `http://publications.gbdirect.co.uk/c_book/`
  - The Library has books on learning C
- More help with this course
  - Moodle `http://moodle.bath.ac.uk/course/view.php?id=30475`
  - E-mail - programming1@lists.bath.ac.uk
- Online C IDE
  - `https://www.codechef.com/ide`
  - Remember to select C as the language you are coding in...

- The places that you can get additional support if you are finding the pace of the course a little fast now include
  - A labs (Continued from week 1)
  - B labs (Starting this week: Fridays 17:15 to 19:15 in CB 5.13)
  - PAL sessions (Started this week: Mondays 14:15 to 15:05 1E 3.9)
  - Drop in Sessions (**Starting next week**: Wednesdays 11:15-13:05 EB0.7)

- Details about the Advanced Labs will also be posted on Moodle

Further Functions

Back to C

- Last week we looked at two pre-defined functions
  - main()
  - printf()
- This week, we will extend our discussion of both pre- and self-defined functions
  - Passing data to a functions (arguments and parameters)
  - Return data from a functions (return statements)
  - Managing types within functions
  - Scope

Reasons to Create New Functions

- Simplify code by grouping complex set of statements behind a single command
- Make program smaller by eliminating repetitive code

- Once we have written a function of our own, we can call it repeatedly.
- And can use one function to call another...

- Here is a function which takes no arguments, and outputs a newline character

```
void one_line(void)
{
    printf("\n");
}
```

- We can call (execute) our new line function the same way we call built-in functions

```
#include <stdio.h>

int main(void)
{
    one_line();

    return (0);
}

...
```

```
$ gcc -o example example.c
$ ./example

$
```

- If we wanted multiple newlines, we could call the same function repeatedly
- Note: functions can be composed of other functions

```
#include <stdio.h>

int main(void)
{
    one_line();
    one_line();
    one_line();
    return (0);
}

void one_line(void)
{
    printf("\n");
}
```

- Alternatively, write a function that prints three new lines:

```c
#include <stdio.h>

int main(void)
{
    three_lines();
    return (0);
}

void three_lines(void)
{
    one_line();
    one_line();
    one_line();
}

void one_line(void)
{
    printf("\n");
}
```

Flow of Execution (Adapted from "How To think Like a Computer Scientist)

- The order in which statements are executed can be described as the **"Flow of execution"**.
- The flow of execution begins with the first statement of the main function
- Statements are executed one at a time, in order, from top until bottom
- But function **calls** cause detours in the flow of execution.

Flow of Execution

- Note: Function **definitions** do not alter the flow of execution of the program,
- Statements inside the function definition are not executed until the function is **"called"** (used)
- But when a function is called, flow jumps to the first line of the called function.
- All statements of called function are then executed
- Then flow then returns to the line following the one from which function was called

Flow of Execution

- That sounds simple enough,
- until you remember that one function can call another.
- While in the middle of one function,
- the program might have to execute the statements in another function.
- But while executing that new function,
- the program might have to execute yet another function!

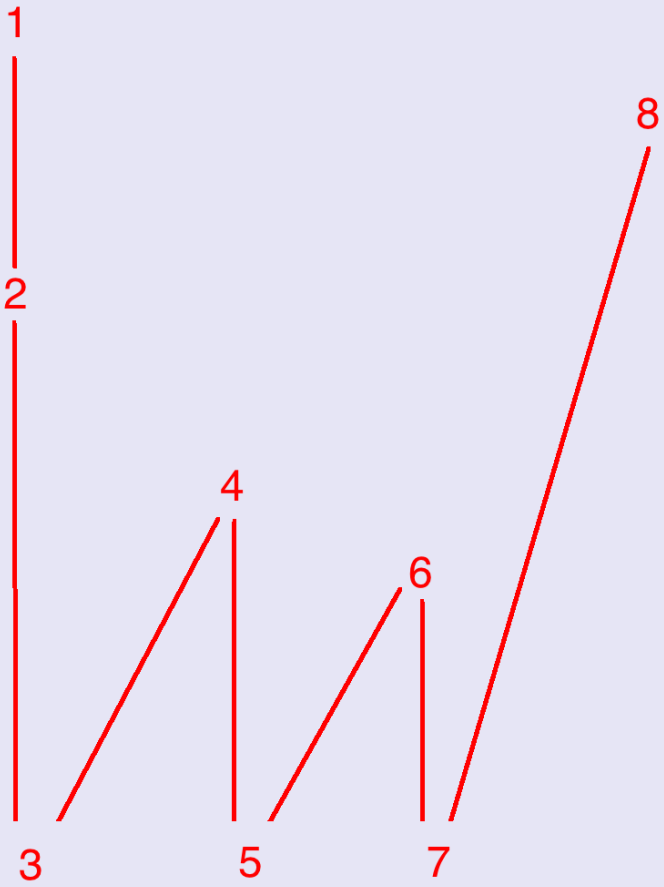
Flow of Execution

```c
#include <stdio.h>

int main(void)
{
    three_lines();
    return (0);
}

void three_lines(void)
{
    one_line();
    one_line();
    one_line();
}

void one_line(void)
{
    printf("\n");
}
```



Flow of Execution

- Fortunately, C is adept at keeping track of where it is,
- So each time a function completes,
- The program picks up where it left off in the function that called it.
- When it gets to the end of the program (i.e. the end of the main function)
- It terminates.

- When you read a program,
- dont read from top to bottom.
- Instead, follow the flow of execution.

- To this point, the functions that we have written

- Cause detours in the flow of execution
- But take no input as they do so
- And return no output
- Though they do print to the **standard out** (usually the screen)

- As we build more complex functions, we may need to

- Provide input to our functions
- And have them return values or data
- Rather than just printing to the screen

Arguments and Parameters

- In order to pass data **to** a function in C, we use a combination of parameters and arguments
- A parameter is what appears in the definition of the function.
- An argument is the instance passed to the function at runtime.

```c
#include <stdio.h>

int main(void)
{
    print_input(3);
    return 0;
}

void print_input(int i)
{
    printf("%d\n", i);
}
```

Returning Data from Functions

- Just as some of the functions that we write accept input
- Some (but not all) will also return output
- In other words you can write functions that yield results.
- If you want to pass on the result, you use the 'return' statement.

```c
#include <stdio.h>

int main(void)
{
    int num1 = 7;
    int num2 = 3;
    int result = remainder_of(num1,num2);
    print_input(result);
    return 0;
}

void print_input(int i)
{
    printf("%d\n",i);
}

int remainder_of(int i, int j)
{
    return (i%j);
}
```

Aside: Modulo

- Note the use of the modulo or % operator in the previous code example
- The modulo operator works on integers (and integer expressions)
- Yields the remainder when the first operand is divided by the second.

- if x%y is zero, then x is divisible by y - e.g. $4\%2 = 0$
- x % 10 yields the rightmost digit of x (in base 10) - e.g. $124 \% 10 = 4$
- x % 100 yields the last two digits - e.g. $124 \% 100 = 24$

Back to Functions

- Note also that there need not be any relationship between
  - passing an input argument and
  - returning a value
- Functions can,
  - pass no argument and return no value
  - pass no argument and return a value
  - pass an argument but return no value
  - pass an argument and return a value

- A return statement ends the execution of the function
- Control is then passed back to the code that called the function.
- Code that appears in a function after a return statement, or any other place, the flow of execution can never reach, is called dead.

```c
#include <stdio.h>

int main(void)
{
    print_input(remainder_of(2,5));
    return 0;
}

int remainder_of(int i, int j)
{
    return (i%j);
    /* dead code */
    int new_result = (j%i);
    print_input(new_result);
}

void print_input(int i)
{
    printf("%d\n",i);
}
```

- C forces us to define data types at compile time for
  - Arguments and parameters
  - Return values
- In this sense, C is pretty strict in its handling of types

Terminology : Typed Languages

- Statically typed language
  - A language in which types are fixed at compile time
  - Most statically typed languages enforce this by requiring you to declare all variables with their datatypes before using them
  - C and Java are statically typed languages
- Dynamically typed language
  - A language in which types are discovered at execution time - the opposite of statically typed
  - JavaScript and Python are dynamically typed, because they figure out what type a variable is when you first assign it a value

- Strongly typed language
  - A language in which types are always enforced
  - Java and Python are strongly typed. If you have an integer, you can't treat it like a float without explicitly converting it.
- Weakly typed language
  - A language in which types may be ignored; the opposite of strongly typed
  - Perl and VBScript are weakly typed
  - In VBScript, you can concatenate the string '12' and the integer 3 to get the string '123', then treat that as the integer 123, all without any explicit conversion.

- C (like Java) is
  - Statically typed: because it uses explicit datatype declarations
  - Strongly typed: because once a variable has a datatype, it actually matters.

Aside: Type Conversion

- It is possible in C to convert between types
- The first is implicit conversion, for example of integers to floats during division
- This is also called **type coersion**
  - float x = 9 / 1.0;
- The second is **casting**, which **explicitly** changes data of one type type to data of another
- for example, casting an integer to a float

```
#include <stdio.h>

main() {
    int sum = 17, count = 5;
    double mean;

    mean = (double) sum / count;
    printf("Value of mean : %f\n", mean );
}
```

Aside: Robustness

- You should also note that your code may at some point receive unexpected input
  - e.g. a parameter of the wrong type (e.g. double instead of int)
  - e.g. a parameter that is out of range (e.g. a negative number when a positive one is expected
- The user might not be aware of this
- Your code should exit gracefully when wrong input is provided
- In other words, your code has to be made **robust**
- We will look at ways to make your code robust as we progress through the course

Back to Functions

- Function variables and parameters are local
- Variables created *inside* a function can only be used *inside*.
- When example_function terminates in the following code, the value of the variable example_variable is destroyed.
- The following code generates an error at compile time

```c
#include <stdio.h>

int main(void)
{
    example_function();
    printf("%d\n", example_variable);
    return 0;
}

void example_function()
{
    int example_variable = 4;
    printf("%d\n", example_variable);
}
```

- The same rules that apply to variables also apply to parameters
- For example, outside the function example_function2, there is no such thing as example_param
- If you try to use it, C will complain

```c
#include <stdio.h>

int main(void)
{
    example_function2();
    printf("%d\n", example_param);
    return 0;
}

void example_function2(int example_param)
{
    int example_variable = 4;
    printf("%d\n", example_variable);
}
```

- We can, however, create variables whose scope is not limited (i.e. variables that can be used throughout your code)
- Those variables are described as global variables
- Global variables are (rather unsurprisingly) defined *outside* individual functions

```c
#include <stdio.h>

int example_global_variable = 3;

int main(void)
{
    example_function();
    printf("%d\n", example_global_variable);
    return 0;
}

void example_function()
{
    printf("%d\n", example_global_variable);
}
```

- Bringing those two ideas together:

- Global variables can be accessed from everywhere
- Other variables and parameters can only be accessed within the function where they are defined
  - and can only be accessed after being defined
- The area in which a variable can be referenced is called the **scope** of the variable
- In C, braces define the scope of a variable.

- *Since all of our code sits within the global scope*
- *We say that the scope of the function is nested in the global scope*

- The functions we have looked at so far either print to the standard out or return the result of a simple calculation
- We may not, of course, want to return the same value every time that we call (execute) a function
- Before considering branches in an example program that will cause **different** returns, depending on the arguments passed, we will need to consider a new data type - booleans

Booleans

- A boolean can have one of two values
  - true
  - false
- You can combine booleans using **and (&)**, **or (|)**, **not (!)**
  - t=true
  - f=false
  - t & t $\rightarrow$ true
  - f & t $\rightarrow$ false

- In programming languages such as Java and C#, a boolean is a built in type
- In C a boolean is not a built in type
- (However <stdbool.h >does provide this functionality)
- In C boolean values (i.e. TRUE and FALSE) are represented as 1 and 0

Comparisons

- We can compare values or expressions that evaluate to numbers in the following way
  - $3 < 5 \rightarrow$ true (1)
  - $3.0 < 5 \rightarrow$ true (1)
  - $3!{=}5 \rightarrow$ true (1)
  - $3 == 5 \rightarrow$ false (0)
  - $3 < 5 <{=}7 \rightarrow$ true (1)
  - $3 < 5 > 2 \rightarrow$ true (1)

Assignment and Testing

- Please note the difference between assignment and testing for equality
  - Use a single equals sign ($=$) for assignment
  - Use a double equals sign ($==$) to test if two things have equal values

```c
#include <stdio.h>

int main(void)
{
    int x = 5;

    printf("%d\n", (5 == x));

    printf("%d\n", (x == 4));

    return 0;
}
```

Conditional Statements

- Armed with boolean statements, we can go on to create **conditional statements** in our code
- Conditional statements allow us to check certain conditions and change the behaviour of the program accordingly
- The simplest conditional statement is the if statement (also described as the IF-THEN statement)

```
if (testscore >= 90)
{
    char grade = 'A';
}
```

- The example on the previous slide uses conditional control statements (e.g. an if statement) to create conditional branching,
- So called because because this type of control structure causes the flow of execution to branch off in different directions.

if statements:

- The code between the brackets () is called the condition
  - If it is 1 (or true), the code between the braces $\{\}$ is executed
- If not (i.e. 0 or false), nothing happens (the indented code is skipped)
- The condition can be any expression that evaluates to a boolean
  - boolean expressions
  - comparisons
  - functions with booleans as a return value

Constructing an if statement

- if statements, (somewhat) like function definitions are compound statements
- Their syntax is as follows:
  - if (condition) {
    - First Statement
    - Second Statement
    - Etc.
  - }
- All the statements between braces are treated as a unit
- Either all are executed or none

- A second form of the if statement is IF/ELSE
- The else block allows us to provide code that will execute if and only if the condition is NOT met

```
if (testscore >= 90)
{
    grade = 'A';
}
else
{
    grade = 'B';
}
```

- We can chain if/else statements

```
if (testscore >= 90)
{
    grade = 'A';
}
else if (testscore >= 80)
{
    grade = 'B';
}
else
{
    grade = 'C';
}
```

- We can also nest if/else statements within each other
- Note that nesting if/else statements has a different effect to chaining

```
int age = 29;
if (age < 16)
{
  printf("Child");
}
else
{
  if (age < 65)
  {
    printf("Adult");
  }
  else
  {
    printf("Senior");
  }
}
```

- If you create a function that requires a return statement
- then you have to guarantee that every possible path through the program hits such a return

```
if (age > 16)
{
    return "Can␣drive";
}
else if (age < 16)
{
    return "Can't␣drive";
}
```

- PROBLEM: What happens if age = 16?

Recursion

- Functions, besides calling other functions, can also call themselves.
- This turns out to be rather useful.
- The process of a function calling itself is called **recursion**
- functions which perform recursion are said to be **recursive**

- This code will count down from n (try it in the lab)
- It will, however, attempt to keep going indefinitely (and eventually crash)

```c
#include <stdio.h>

int main(void) {
    example_recursive_function(5);
    return 0;
}

void example_recursive_function(int counter)
{
    printf("%d\n",counter);
    example_recursive_function(counter-1);
}
```

Recursion Problem Specification

- Recursive problems (and correctly constructed recursive functions) have at least one simple case, **the base case**, which can be solved without recursion
- All other cases can be reduced to a case closer to the base case by means of recursion

- Eventually all cases can be reduced to the base case
- If recursion never reaches a base case it will continue making recursive calls forever.
- This is called infinite recursion.
- A finite amount of memory is consumed on each recursive call, and eventually the program will terminate with an error

- **Recursion: Structure**
  - IF the base case is reached
    - ★ SOLVE
  - ELSE
    - ★ reduce the problem using recursion
- **Definition: Problem Size**
  - The number of reduction steps a problem is away from the base case is the **problem size**

- The following code counts down to zero and then stops (assuming positive input)

```c
#include <stdio.h>

int main(void) {
    example_recursive_function(5);
    return 0;
}

void example_recursive_function(int counter)
{
   if(counter == 0)
   {
       return;
   }
   else
   {
       printf("%d\n",counter);
       example_recursive_function(counter-1);
   }

}
```

- the if-statement (if counter == 0) contains the base case criteria
- If we reach this case, we can stop the recursive calling of the function or print or return the result
- The else-statement prints the current counter
- And then calls the example_recursive_function function with the lower counter value

- This output from this code is as follows:

```
$ gcc -o example example.c

$ ./example
5
4
3
2
1
```

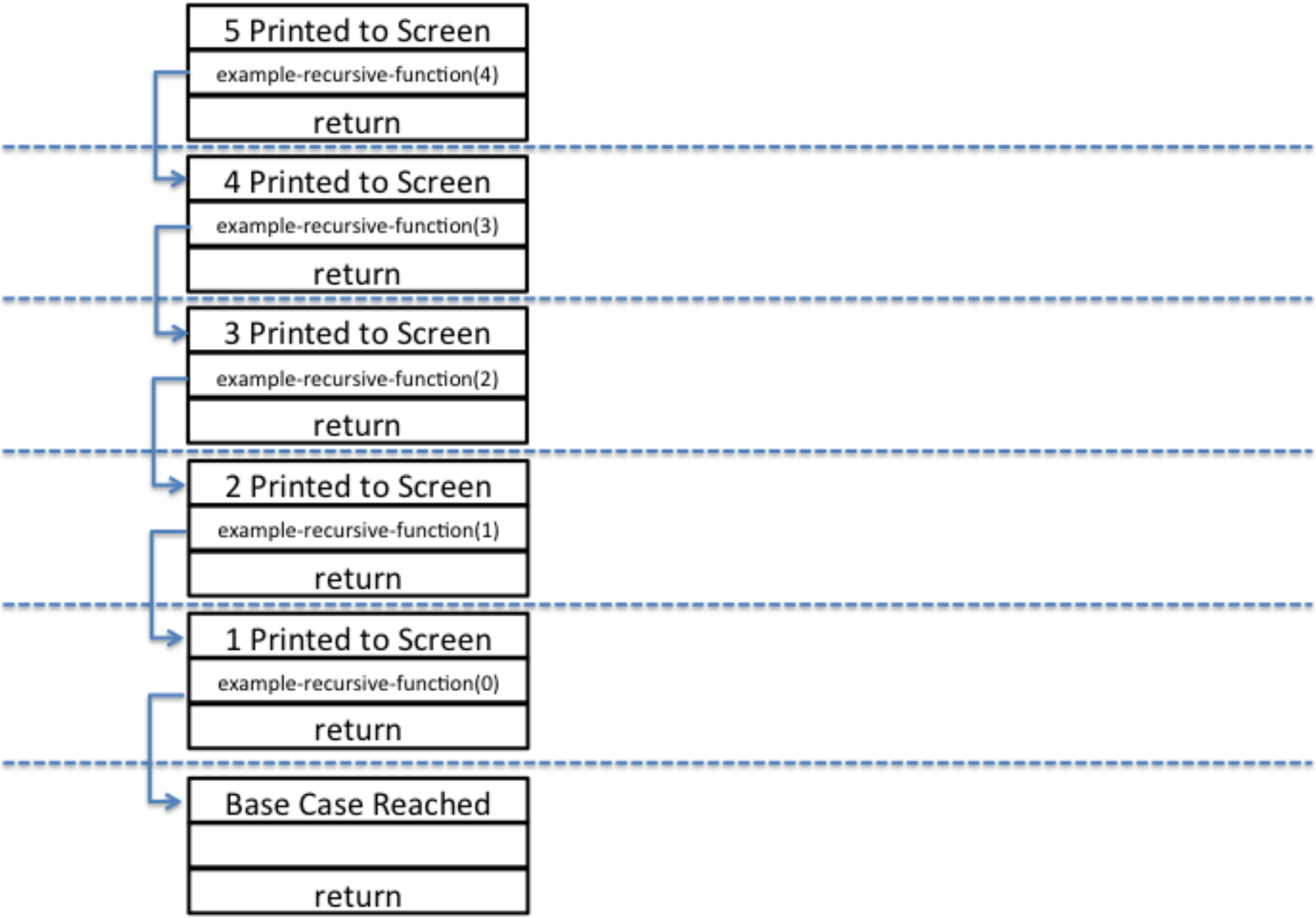
Stacks

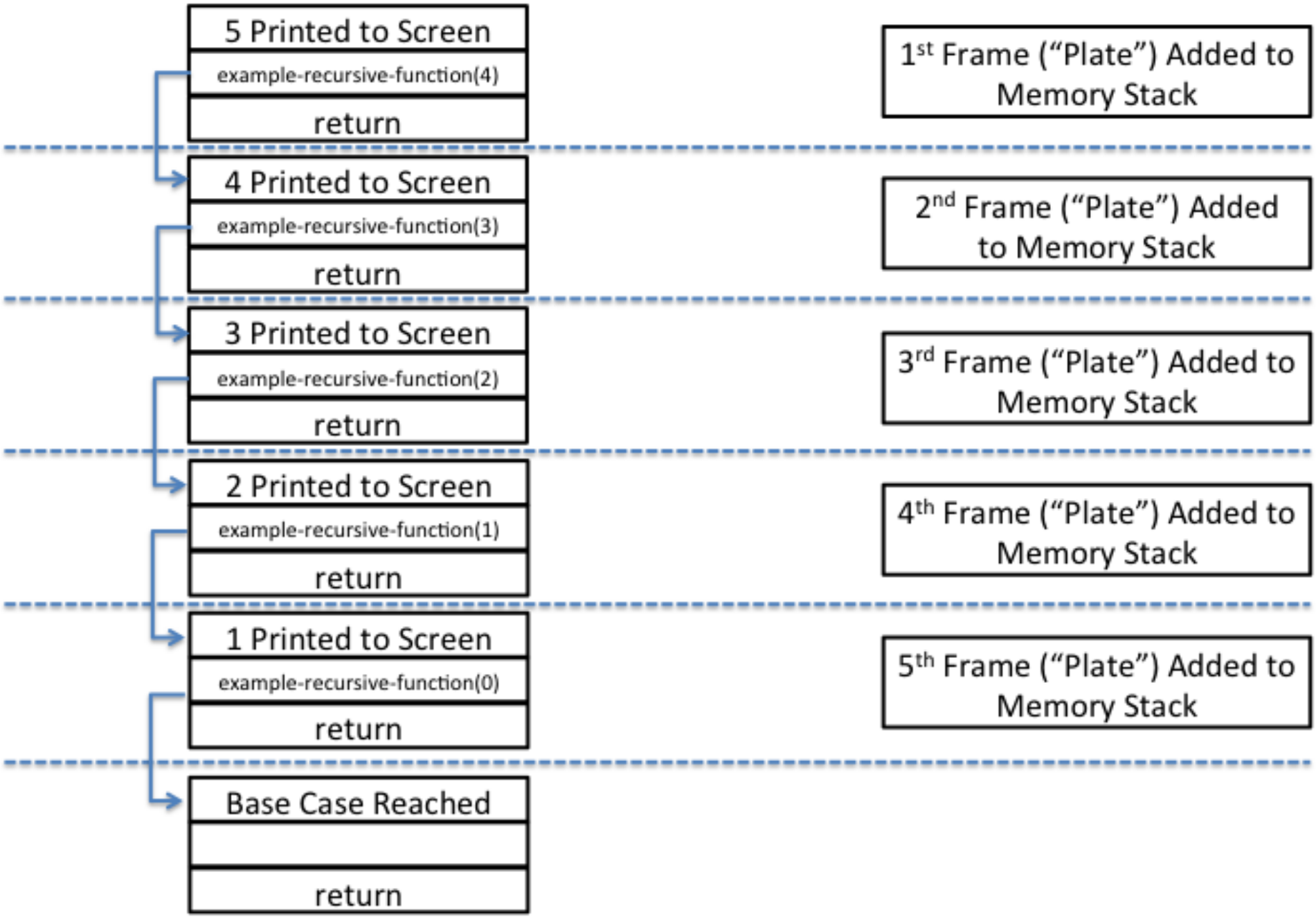
- You should note that taking a recursive approach to problems that require a great number of steps to reach the base case (have a large problem size) places a heavy burden on computer memory

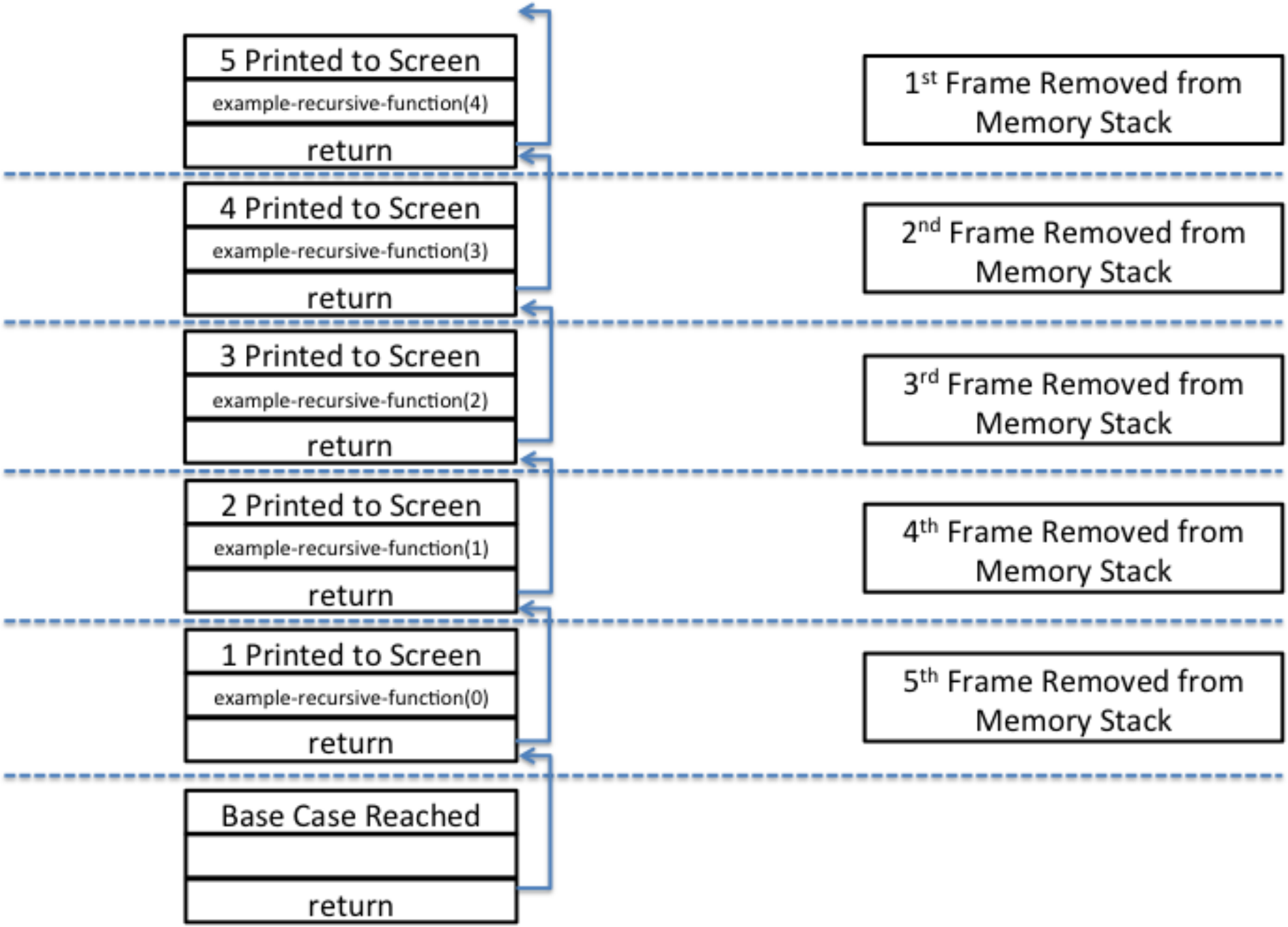
Stacks

- C uses a "stack" to store the data held in variables as code executes
- A stack can be seen as a pile of plates.
- Clean (new) plates go on top
- And plates are always taken from to top.
- So a stack works: LIFO:
  - Last In First Out
- These "Plates" are referred to as **frames**

- Every time you call a function or a procedure,
- memory is allocated on the memory stack to store the parameters and local variables
- As the following example shows, those parameters and variables are stored "above" those that went before
- This means that they will be accessed before those that went before


5 Printed to Screen
example-recursive-function(4)
return
4 Printed to Screen
example-recursive-function(3)
return
3 Printed to Screen
example-recursive-function(2)
return
2 Printed to Screen
example-recursive-function(1)
return
1 Printed to Screen
example-recursive-function(0)
return
Base Case Reached
return


5 Printed to Screen
example-recursive-function(4)
return
4 Printed to Screen
example-recursive-function(3)
return
3 Printed to Screen
example-recursive-function(2)
return
2 Printed to Screen
example-recursive-function(1)
return
1 Printed to Screen
example-recursive-function(0)
return
Base Case Reached
return
1st Frame ("Plate") Added to Memory Stack
2nd Frame ("Plate") Added to Memory Stack
3rd Frame ("Plate") Added to Memory Stack
4th Frame ("Plate") Added to Memory Stack
5th Frame ("Plate") Added to Memory Stack


5 Printed to Screen
example-recursive-function(4)
return
4 Printed to Screen
example-recursive-function(3)
return
3 Printed to Screen
example-recursive-function(2)
return
2 Printed to Screen
example-recursive-function(1)
return
1 Printed to Screen
example-recursive-function(0)
return
Base Case Reached
return
1st Frame Removed from Memory Stack
2nd Frame Removed from Memory Stack
3rd Frame Removed from Memory Stack
4th Frame Removed from Memory Stack
5th Frame Removed from Memory Stack

Fibonacci Sequence

- The first lab sheet asked you to write code that calculates the first n numbers in the Fibonacci sequence
- We will now consider a recursive programming approach to that problem

Fibonacci Sequence

- Problem Definition: number n in the Fibonacci sequence is the sum of numbers (n-1) and (n-2) in that same sequence
- The first three numbers in the sequence are 0, 1 and 1

Fibonacci Sequence : Pseudocode

- In pairs (or threes) write this algorithm in pseudocode

- HINT: The first 7 numbers in the Fibonacci sequence are
- 0 1 1 2 3 5 8

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

}
```

- In pairs (or threes) try to write down the rise and fall of the memory stack in this code...