Fundamental Data Structures & Algorithms using the C language

David Corzo

2020 May 18

Contents

Chapter 1

Recursion

1.1 Introduction to recursion

- Anything done with recursion can be done with a loop.
- Memory sided way in which recursion goes on.
- Compare between recursive and iterative approaches of problem solving.
- Tail recursion.

It is recommended to try to understand recursion in an abstract way.

1.2 Basic concept of recursion

A function that calls itself until a base condition is met.

- The following program is an example of a recursive function call, the problem is that the function will call itself infinitely, we don't have anything to actually break out of the recursive call.

```c
void print(){
    printf("Hello!!\n");
    print(); // recursive call
}
```

- Suppose another example, we want to print a variable k inside the print function, can you guess what will it print as the value of k?

```c
void print(){
    int k = 1;
    printf("Hello!!,k = %d\n",k);
    k++;
    print(); // recursive call
}
/* Output:
Hello!!,k = 1
Hello!!,k = 1
Hello!!,k = 1
Hello!!,k = 1
...
*/
```

  - Even though we increment k before calling the function again recursively we still get 1, this is because upon calling the function again we are creating. k is allocated every time the function calls itself.

- We can solve this by making the k variable static, this means that you don't want to allocate a new space for k every time the function calls itself but rather the space allocated will be always the same.

  - Static variables are meant to reference only one allocated space during the entirety of the program and exist once during the entirety of the program.

```c
void print(){
    static int k = 1;
    printf("Hello!!,k = %d\n",k);
    k++;
    print(); // recursive call
}
/* Output:
Hello!!,k = 1
Hello!!,k = 2
Hello!!,k = 3
Hello!!,k = 4
...
*/
```

  - As we can see k is incrementing now because the k variable has been declared static.

- The recursive function above will never terminate, it doesn't have a case in which you want to break the recursion, we can use this with a simple if-else statement, and to this if-else statement we give the name as the *base case* which will break the loop.

```c
void print(){
    static int k = 1;
    printf("Hello!!,k = %d\n",k);
    k++;
    if (k <= 3){
        print(); // recursive call
    } else {
        return;
    }
}
/* Output:
Hello!!,k = 1
Hello!!,k = 2
Hello!!,k = 3

*/
```

  - Remember to look at this abstractly, if you inspect all the function calls at a low level you will find it more difficult to understand recursion.

- Usually we don't use recursion like this, k would be a parameter and the function would be called with variable parameters.

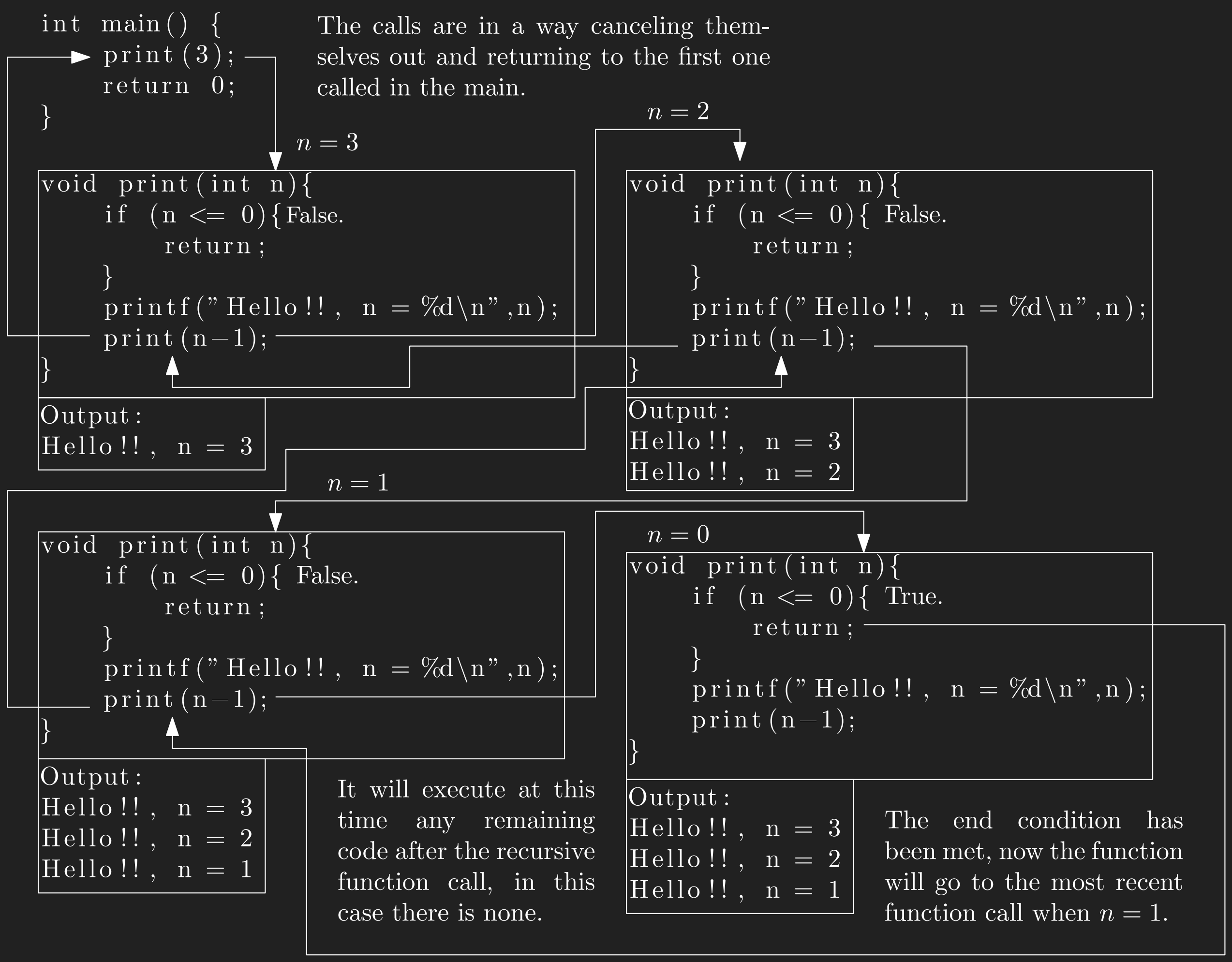
1.3 When and how to terminate — the base condition of recursion

- If we were to print a variable number of times we must accept a parameter and based on that execute the function.

```c
void print(int n){
    if (n <= 0){
        return;
    }
    printf("Hello!!\n");
    print(n-1); // recursive call
}
```

  - In order to meet the base condition and eventually terminate the recursive calls

1.4 Let us go into the depth of the call



```
int main() {
    print(3);
    return 0;
}
```

The calls are in a way canceling themselves out and returning to the first one called in the main.

```
void print(int n){
    if (n <= 0){False.
        return;
    }
    printf("Hello!!, n = %d\n",n);
    print(n-1);
}
```

```
Output:
Hello!!, n = 3
```

```
void print(int n){
    if (n <= 0){ False.
        return;
    }
    printf("Hello!!, n = %d\n",n);
    print(n-1);
}
```

```
Output:
Hello!!, n = 3
Hello!!, n = 2
```

```
void print(int n){
    if (n <= 0){ False.
        return;
    }
    printf("Hello!!, n = %d\n",n);
    print(n-1);
}
```

```
Output:
Hello!!, n = 3
Hello!!, n = 2
Hello!!, n = 1
```

```
void print(int n){
    if (n <= 0){ True.
        return;
    }
    printf("Hello!!, n = %d\n",n);
    print(n-1);
}
```

```
Output:
Hello!!, n = 3
Hello!!, n = 2
Hello!!, n = 1
```

It will execute at this time any remaining code after the recursive function call, in this case there is none.

The end condition has been met, now the function will go to the most recent function call when $n = 1$.

- If you were to place anything below the recursive call it will be executed after the calls have been made.

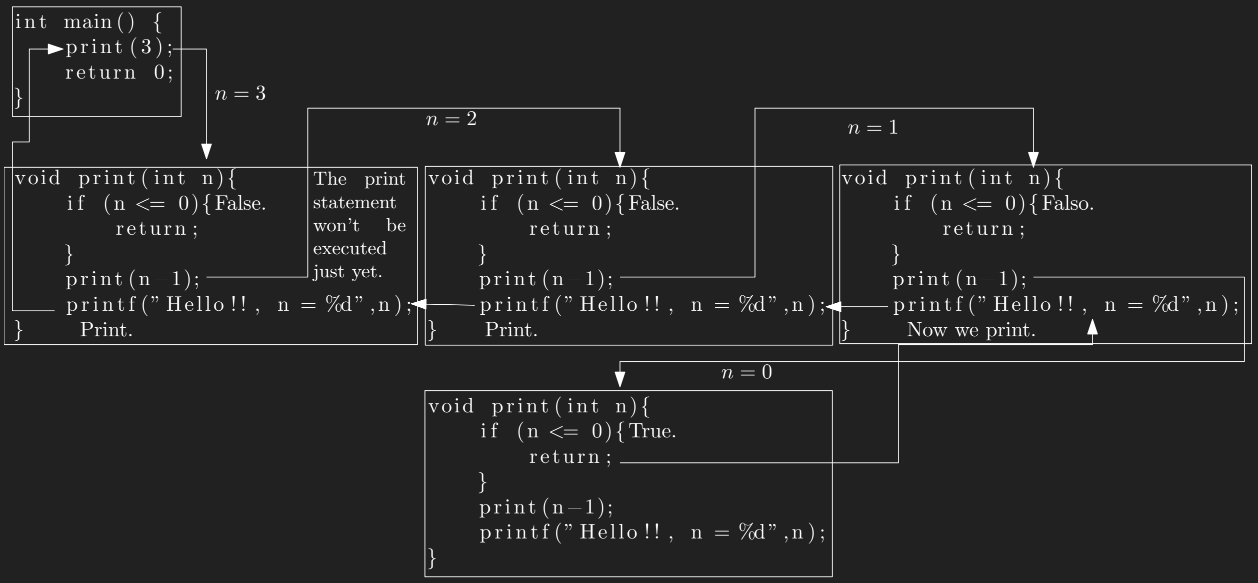
```c
    void print(int n){
    if (n <= 0){
        return;
    }
    printf("Hello!!\n");
    print(n-1); // recursive call
}
/* Output:
Hello!!, n = 3
Hello!!, n = 2
Hello!!, n = 1

*/
```

- If we were to make the print statement after the recursive call the order will be inverted.

```
void print(int n){
    if (n <= 0){
        return;
    }
    print(n-1); // recursive call
    printf("Hello!!\n");
}
/* Output:
Hello!!, n = 1
Hello!!, n = 2
Hello!!, n = 3

*/
```



```
int main() {
    print(3);
    return 0;
}
```

```
void print(int n){
    if (n <= 0){False.
        return;
    }
    print(n-1);
    printf("Hello!!, n = %d",n);
}   Print.
```

```
void print(int n){
    if (n <= 0){False.
        return;
    }
    print(n-1);
    printf("Hello!!, n = %d",n);
}   Print.
```

```
void print(int n){
    if (n <= 0){Falso.
        return;
    }
    print(n-1);
    printf("Hello!!, n = %d",n);
}   Now we print.
```

```
void print(int n){
    if (n <= 0){True.
        return;
    }
    print(n-1);
    printf("Hello!!, n = %d",n);
}
```

1.5 Recursion example — Juggler Sequence

https://en.wikipedia.org/wiki/Juggler_sequence

```
#include <stdio.h>
#include <stdlib.h>
```

```c
#include <math.h>

//  asume the congecture 10^6 nums end at one.
void juggler(int a){
    // base condition.
    if (a == 1){
        printf("%d\n",a);
        return;
    }
    printf("%d, ",a);
    juggler((a % 2) != 0 ? (int)pow(a,1.5) : (int)pow(a,0.5));
}
void juggler_rev(int a){
    // base condition.
    if (a == 1){
        printf("%d, ",a);
        return;
    }
    juggler_rev((a % 2) != 0 ? (int)pow(a,1.5) : (int)pow(a,0.5));
    printf("%d, ",a);
}
int main() {
    printf("Juggler(3): ");
    juggler(3);
    printf("Juggler(3) reverse: ");
    juggler_rev(3);
    return 0;
}
/* Output:
Juggler(3): 3, 5, 11, 36, 6, 2, 1
Juggler(3) reverse: 1, 2, 6, 36, 11, 5, 3,
*/
```

1.6 Recursion example — Finding Factorial

https://es.wikipedia.org/wiki/Factorial#:~:text=Podemos%20definir%20el%20factorial%20de,menores%20o%20iguales%20que%20n.

```c
#include <stdio.h>
#include <stdlib.h>

unsigned int factorial(unsigned int n){
    if (n == 0){
        return 1;
    }
    return n * factorial(n-1);
}

int main() {
    printf("%u",factorial(5));
    return 0;
}
```

1.7 Recursion example — Binary Search

```c
#include <stdio.h>
#include <stdlib.h>

int bin_search(int arr[], int lb, int ub, int target){
    int middle = (int)((lb + ub) / 2);
    if (lb > ub) {
        return -1;
    }
    int m = (int)((lb + ub) / 2);
    if (arr[m] == target){
        return m;
    } else if (arr[m] > target){
        bin_search(arr,lb,m-1,target);
    } else {
        bin_search(arr,m+1,ub,target);
    }
}

int main() {
    int arr[] = {10,20,30,40,50,60,70,80,90};
    const int target = 80;
    int k = bin_search(arr,0,sizeof(arr)/sizeof(int)-1,target);
    k != -1? printf("Found at %d => %d",k,target): printf("Not found.");
    return 0;
}
/* Output:
Found at 7 => 80
*/
```

1.8 Recursion example — Decimal to Binary

```c
#include <stdio.h>
#include <stdlib.h>

void decToBin(unsigned n){
    if (n == 0) {
        printf("0");
        return;
    } else if (n == 1){
        printf("1");
        return;
    }
    int r = (int)(n % 2);
    n = n / 2;
    decToBin(n);
    printf("%d",r);
}
int main() {
    decToBin(67);
    printf("\n");
    decToBin(90);
    printf("\n");
    decToBin(1);
    printf("\n");
    decToBin(0);
    printf("\n");
    decToBin(5);
    printf("\n");
    return 0;
}
/* Output:
1000011
1011010
1
0
101

*/
```

1.9 Calling a function — Operating system creates a stack

- The operating system creates a stack and pushes all the function calls. Then as the functions are executed they get popped.
- The compiler will create this stack.

1.10 When there is no need for a stack

- Sometimes we don't need to preserve the variables and data structures declared before a function.
- Depending on the compiler, if the function call is the last thing done, the variables are not pushed to the call stack.

1.11 Tail recursion

- The stack saves and preserves the context, meaning variables and data structures for their usage, this is what makes recursion possible.
- Without the stack and the preservation of the current context recursion would not be possible.
- When the recursion function is the last thing performed in the function, we call it *tail recursion*; when there are more instructions below the recursive call we call it *non-tail recursion*, tail is more efficient because the context must not be pushed to the stack, the non-tail recursion is more inefficient for the extra time pushing and popping.

1.12 Recursion versus iteration

This is a debate and every programmer must know this.

1.12.1 When both are equivalent

- In the case of tail recursion, you can do this using a for loop, we know that tail recursion doesn't have a stack in memory, so this is equivalent for tail recursion.

Be aware of apparent tail recursion that looks like tail recursion but is not.

```
long factorial(unsigned n){
    if (n == 0){
        return 1L;
    }
    return n * factorial(n-1);
    // after the recursive call is done
    // we multiply n, this implies a stack being created in memory.
}
```

1.12.2 When a loop is better

- Use a loop when you can not find an equivalent of recursive logic to implement tail recursion.
- Tail recursion takes as much complexity as a for loop, thus when we can not find an equivalent tail-recursive operation, we must go with a for loop.

1.12.3 When recursion is better

Take the example of a decimal to binary converter, without the stack this problem cannot be solved, we can use the stack provided by the compiler, the alternative implementation to the recursion is iteration, in the iteration we need to implement the stack ourselves, keep track of all the variables, this implies more code, less readability and bigger code. Better we use the stack already implemented for us and use recursion. In conclusion when you need a stack and the stack is indispensable to the purpose, use the call stack and do the job recursively, if you don't need a stack, don't use a stack.

1.12.4 Synthesis in a programmer's way

```
1 if you can convert the recursion to tail recursion then
2     Iteration and recursion are equivalent;
3     # the factorial implementation the call stack is required for the recursive
        implementation, but you can do the same without a stack, for factorial the
        iteration is better.
4 if we can not convert the loop to tail recursion && it doesn't require a stack then
5     iteration is better;
6 if when we need a stack explicitly to solve a problem then
7     recursion is better;
8     # This is preferred rather to implementing our own stack.
```

Algorithm 1: When to use recursion or iteration