

C Programming

Kasun De Zoysa

GNU Debugger (gdb)

It allows you to inspect what the program is doing at a certain point during execution.

Errors like segmentation faults may be easier to find with the help of gdb.

Normally, you would compile a program like:

```
gcc [flags] <source files> -o <output file>
```

Now you add a -g option to enable built-in debugging support (which gdb needs):

```
gcc [other flags] -g <source files> -o <output file>
```

Starting up gdb

Just try “gdb” or “gdb **prog1.x**.” You’ll get a prompt that looks like this:

```
(gdb)
```

If you didn’t specify a program to debug, you’ll have to load it in now:

```
(gdb) file prog1.x
```

Here, **prog1.x** is the program you want to load, and “file” is the command to load it.

Running the program

To run the program, just use:

```
(gdb) run
```

This runs the program.

- If it has no serious problems (i.e. the normal program didn't get a segmentation fault, etc.), the program should run fine here too.
- If the program *did* have issues, then you (should) get some useful information like the line number where it crashed, and parameters to the function that caused the error:

```
Program received signal SIGSEGV, Segmentation fault.  
0x0000000000400524 in sum_array_region (arr=0x7fffc902a270, r1=2, c1=5,  
r2=4, c2=6) at sum-array-region2.c:12
```

Setting the breakpoint

Breakpoints can be used to stop the program run in the middle, at a designated point. The simplest way is the command “`break`.” This sets a breakpoint at a specified file-line pair:

```
(gdb) break file1.c:6
```

This sets a breakpoint at `line 6`, of `file1.c`. Now, **if** the program ever reaches that location when running, the program will pause and prompt you for another command.

Tip

You can set as many breakpoints as you want, and the program should stop execution if it reaches any of them.

Now what?

- Once you've set a breakpoint, you can try using the `run` command again. This time, it should stop where you tell it to (unless a fatal error occurs before reaching that point).
- You can proceed onto the next breakpoint by typing "`continue`" (Typing `run` again would restart the program from the beginning, which isn't very useful.)

```
(gdb) continue
```

- You can single-step (execute *just* the next line of code) by typing "`step`." This gives you really fine-grained control over how the program proceeds. You can do this a *lot*...

```
(gdb) step
```

Printing the value

- So far you've learned how to interrupt program flow at fixed, specified points, and how to continue stepping line-by-line. However, sooner or later you're going to want to see things like *the values of variables*, etc. This *might* be useful in debugging. :)
- The `print` command prints the value of the variable specified, and `print/x` prints the value in hexadecimal:

```
(gdb) print my_var
```

```
(gdb) print/x my_var
```

Watching the variable

Whereas breakpoints interrupt the program at a particular line or function, watchpoints act on variables. They pause the program whenever a *watched* variable's value is modified. For example, the following `watch` command:

```
(gdb) watch my_var
```

Now, whenever `my_var`'s value is modified, the program will interrupt and print out the old and new values.

Other useful commands

- `backtrace` - produces a stack trace of the function calls that lead to a seg fault (should remind you of Java exceptions)
- `where` - same as `backtrace`; you can think of this version as working even when you're still in the middle of the program
- `finish` - runs until the current function is finished
- `delete` - deletes a specified breakpoint
- `info breakpoints` - shows information about all declared breakpoints

Look at sections 5 and 9 of the manual mentioned at the beginning of this tutorial to find other useful commands, or just try `help`.

Recursions

It is legal for one function to call another; it is also legal for a function to call itself.

It may not be obvious why that is a good thing, but it turns out to be one of the most magical things a program can do.

For example, look at the following C function:

```
void countDown(int n){  
    printf("%d \n",n);  
    if(n>1) countDown(n-1);  
    else printf("Blast!!\n");  
}
```

Function countDown

If n is less than or equal to one, it outputs the word, “Blastoff!” Otherwise, it calls a function named **blast**—itself—passing $n-1$ as an argument.

What happens if we call this function like this?
countDown(3)

3

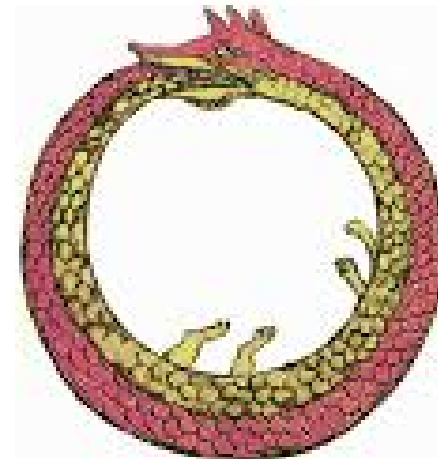
2

1

0

Blastoff!!

Tail Recursion



countDown

The execution of `countDown` begins with `n=3`, and prints 3 on the terminal. Since `n` is greater than 1, it calls itself...

The execution of `countDown` begins with `n=2`, and prints 2 on the terminal. Since `n` is greater than 1, then calls itself...

The execution of `countDown` begins with `n=1`, and prints 1 on the terminal. Now `n` is equal to 1. So it prints "Blast!" and then returns.



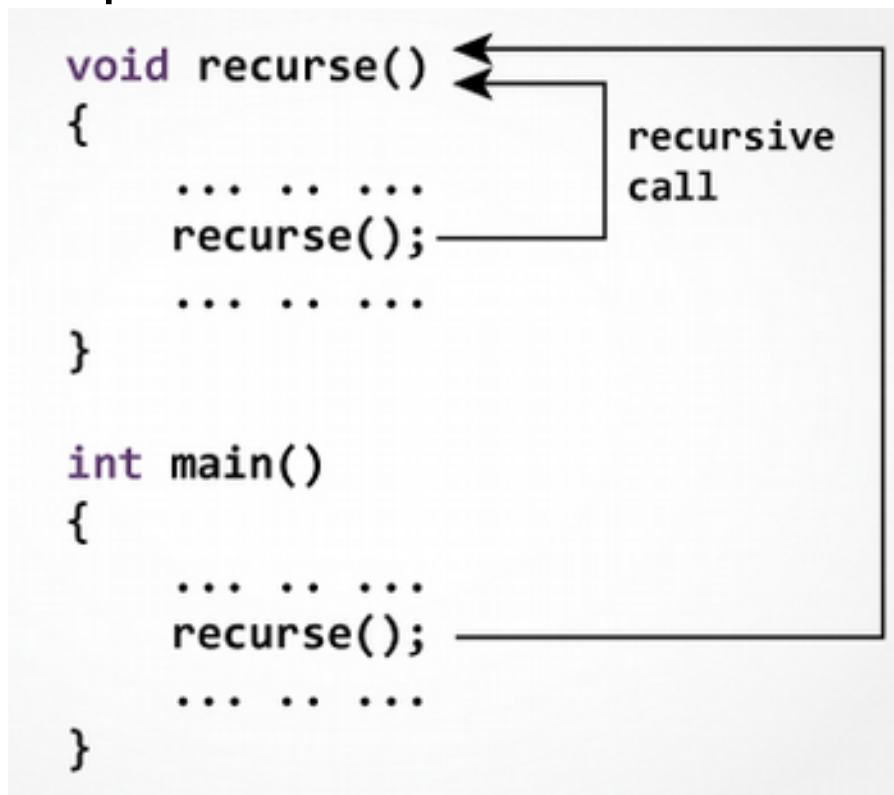
Recursive- Recursion

Now you know : like a snake that swallows its own tail, a function can call itself.

A function that calls itself is **recursive**; the process is called **recursion**.

Recursive Function

A function that calls itself is known as a recursive function. And, this technique is known as recursion.



The recursion continues until some condition is met to prevent it.

You can write a function to calculate factorial n.

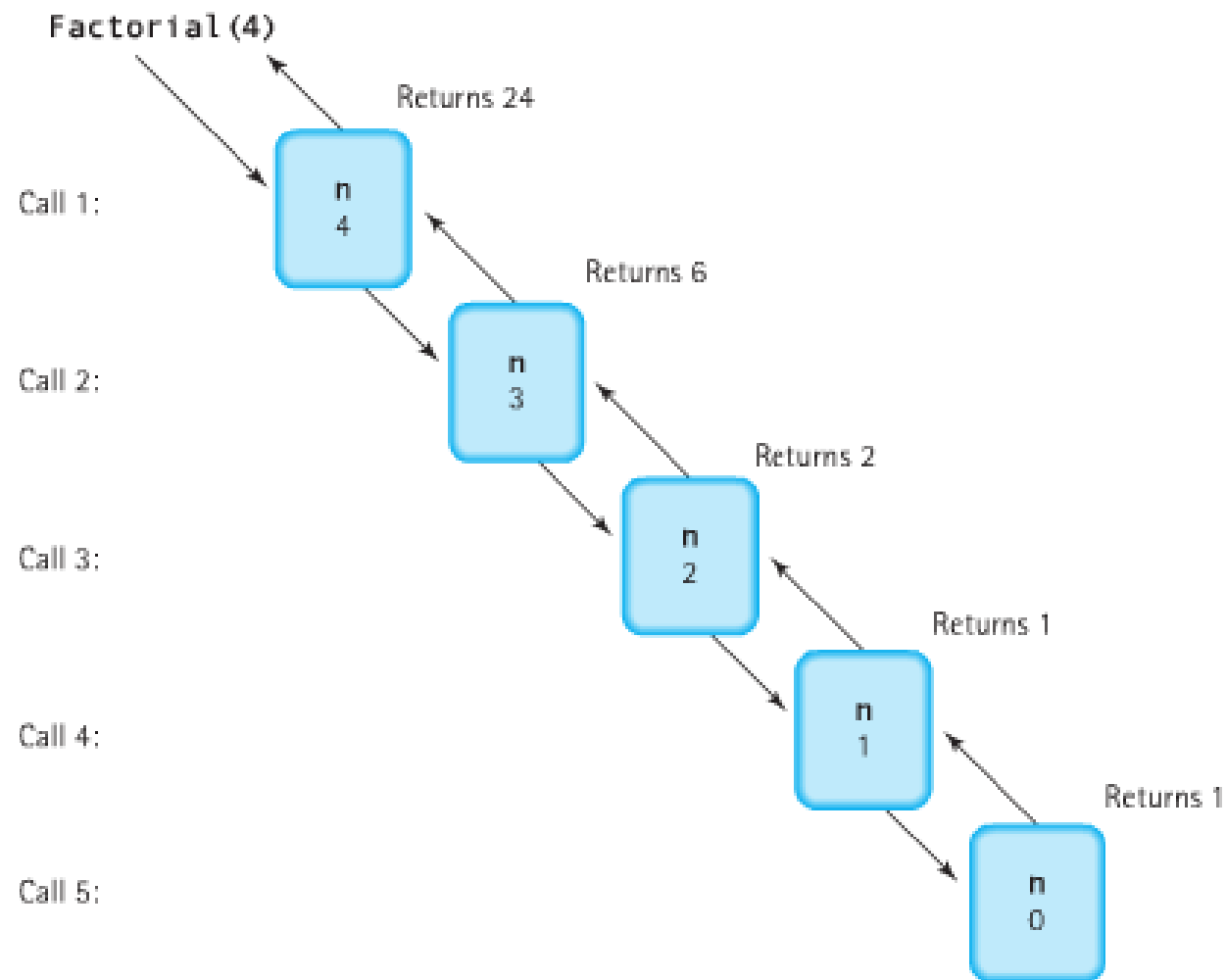
Factorial n = 1* 2 3 * n



0! = 1	
1! = 1	
2! = 2	
3! = 6	
4! = 24	
5! = 120	
6! = 720	
7! = 5040	
8! = 40320	
9! = 362880	
10! = 3628800	

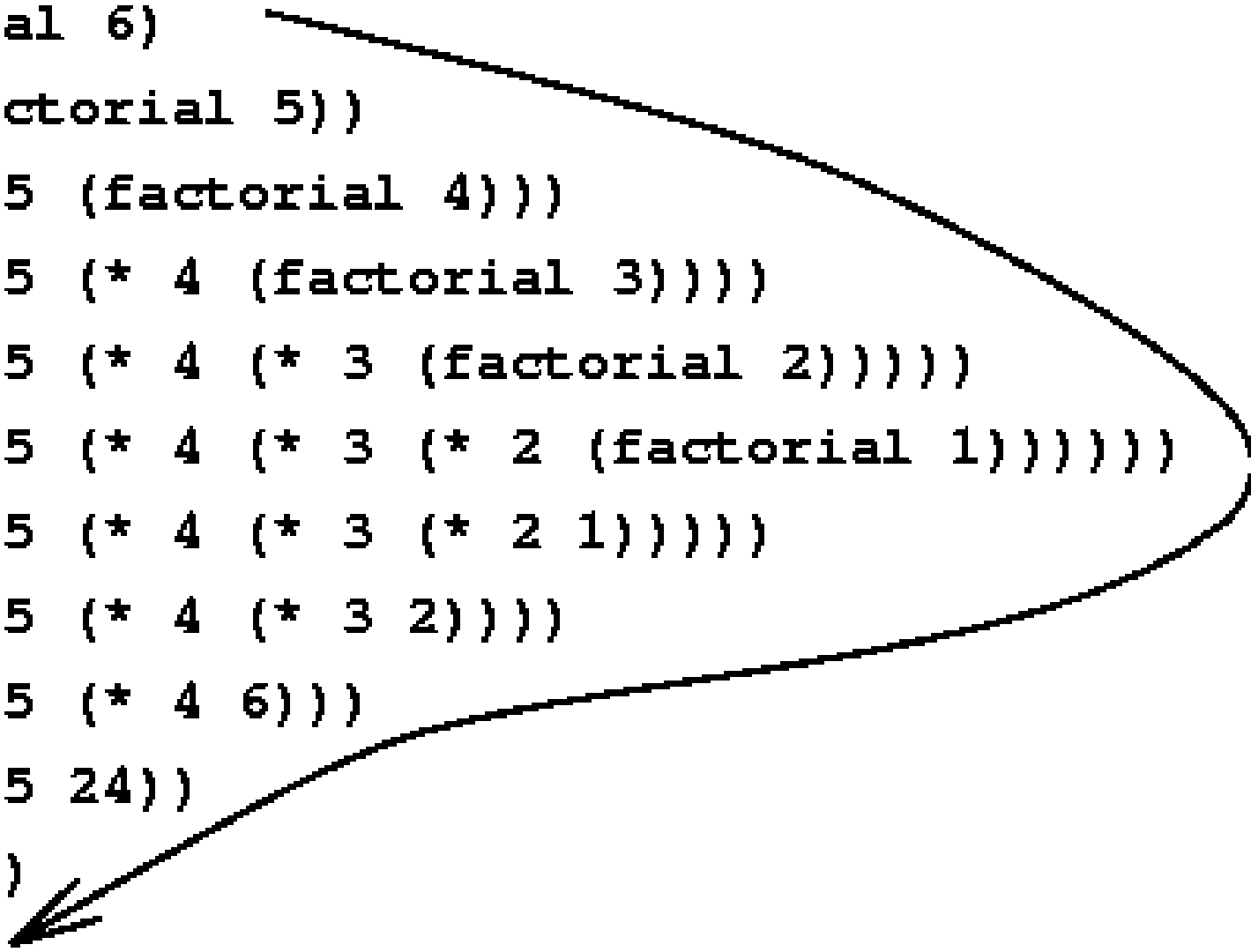
```
long int factorial(int n) {  
    if (n >= 1) return n*factorial(n-1);  
    else return 1;
```

factorial(4)



factorial(6)

```
(factorial 6)
(* 6 (factorial 5))
(* 6 (* 5 (factorial 4)))
(* 6 (* 5 (* 4 (factorial 3))))
(* 6 (* 5 (* 4 (* 3 (factorial 2)))))
(* 6 (* 5 (* 4 (* 3 (* 2 (factorial 1))))))
(* 6 (* 5 (* 4 (* 3 (* 2 1)))))
(* 6 (* 5 (* 4 (* 3 2))))
(* 6 (* 5 (* 4 6)))
(* 6 (* 5 24))
(* 6 120)
```



720

Infinite Recursion

If a recursion never reaches a base case, it goes on making recursive calls forever, and the program never terminates.

This is known as infinite recursion, and it is generally not a good idea.

Here is a minimal program with an infinite recursion:

Fibonacci

After factorial, the most common example of a recursively defined mathematical function is fibonacci, which has the following definition (see http://en.wikipedia.org/wiki/Fibonacci_number):

$$\text{fibonacci}(0) = 0$$

$$\text{fibonacci}(1) = 1$$

$$\text{fibonacci}(n) = \text{fibonacci}(n - 1) + \text{fibonacci}(n - 2)$$

Fibonacci

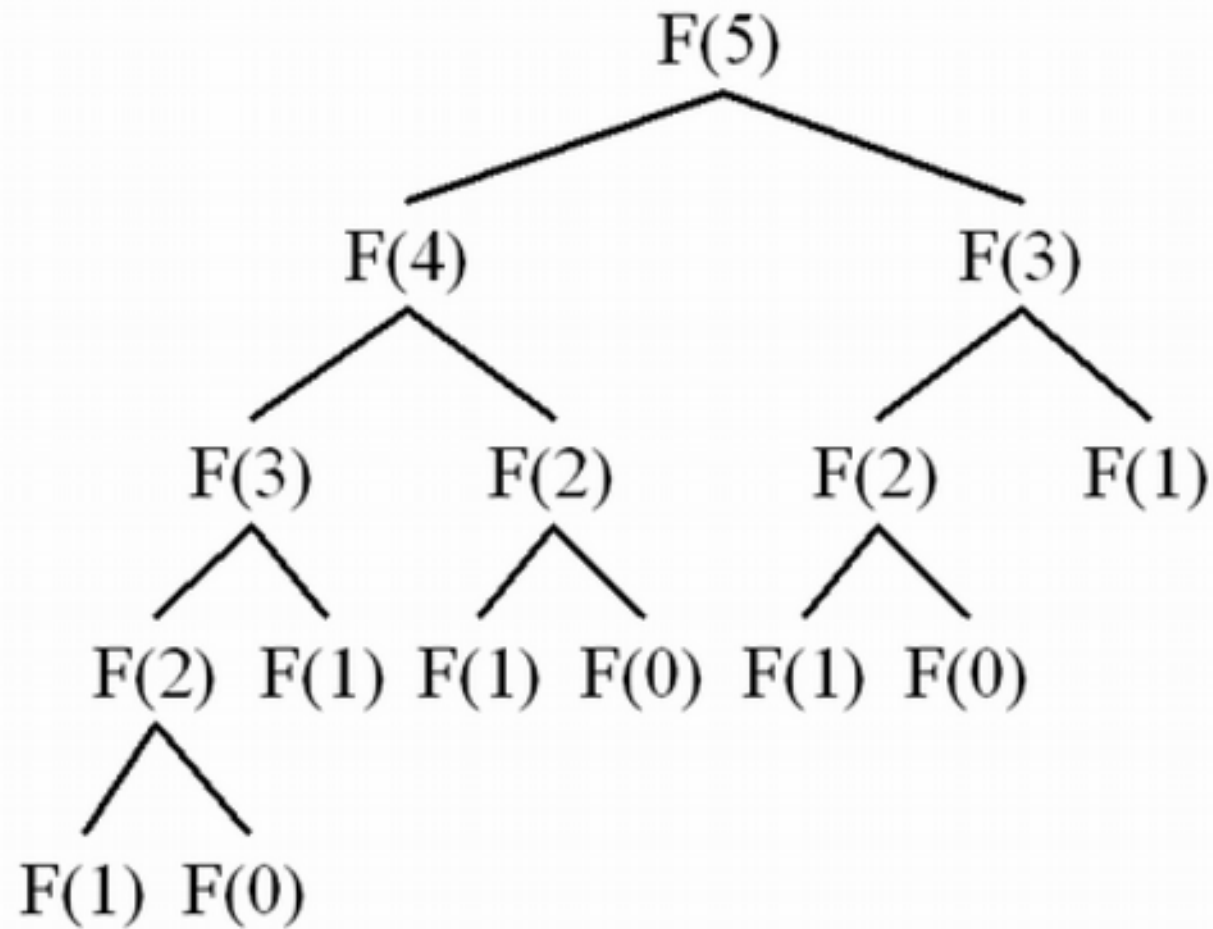
Translated into C, it looks like this:

```
#include <stdio.h>
int fibonacci(int);

int main(){
    int n=10;
    printf("Fibonacci %d => %d \n",n,fibonacci(n));
    return 0;
}

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

fibonacci(5)



Advantages and Disadvantages of Recursion

Recursion makes program elegant and cleaner.

All algorithms can be defined recursively which makes it easier to visualize and prove.

If the speed of the program is vital then, you should avoid using recursion.

Recursions use more memory and are generally slow.

Instead, you can use loop.

Problem?

The greatest common divisor (GCD) of a and b is the largest number that divides both of them with no remainder.

One way to find the GCD of two numbers is Euclid's algorithm, which is based on the observation that if r is the remainder when a is divided by b , then $\text{gcd}(a, b) = \text{gcd}(b, r)$.

As a base case, we can use $\text{gcd}(a, 0) = a$.

Write a recursive functions to calculate GCD of any given two integers.