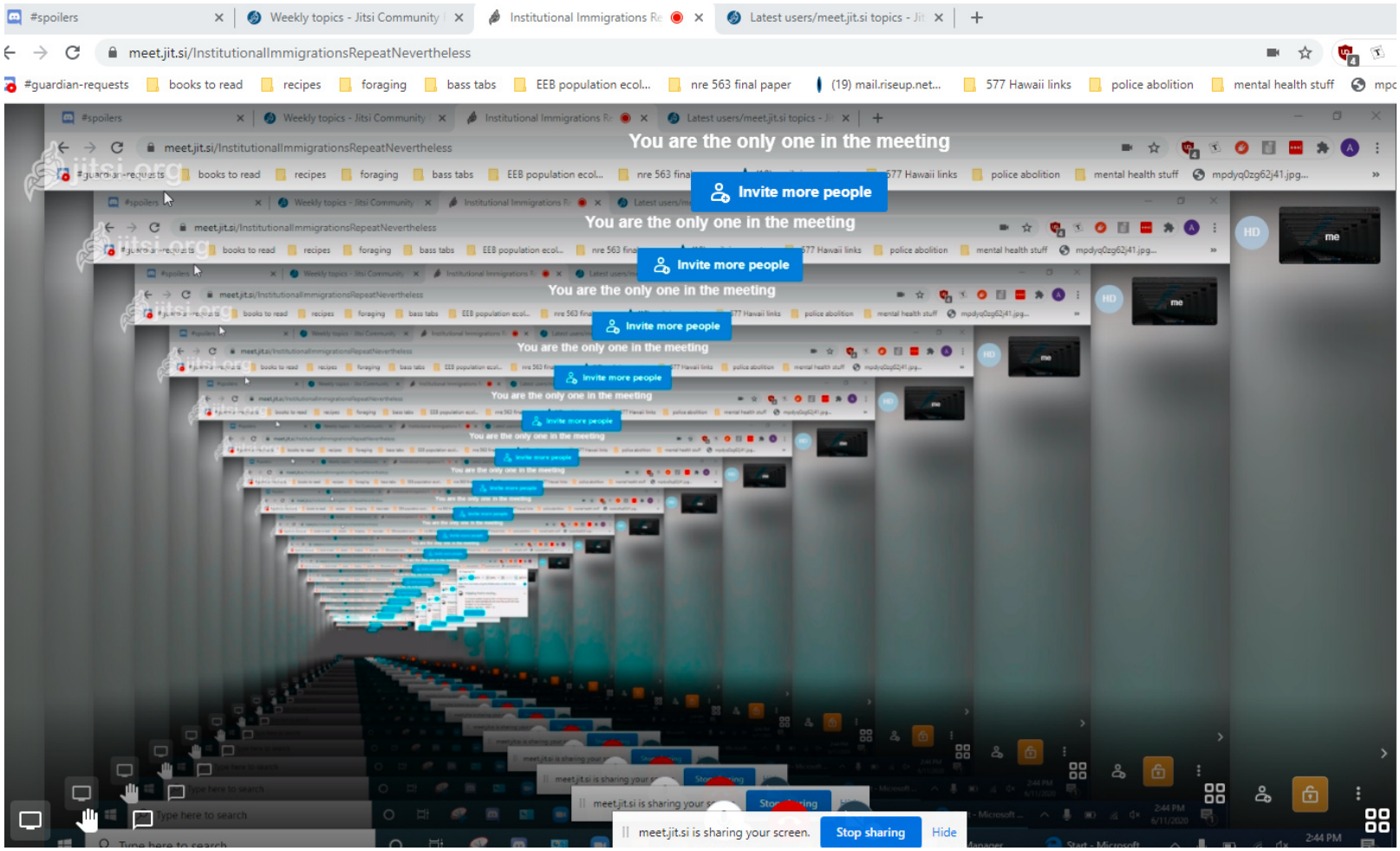
Functions and Recursion

Song Liu (song.liu@bristol.ac.uk)
GA 18, Fry Building,
Microsoft Teams (search "song liu").


#spoilers
Weekly topics - Jitsi Community
Institutional Immigrations Re
Latest users/meet.jit.si topics - Jit
meet.jit.si/InstitutionalImmigrationsRepeatNevertheless
#guardian-requests
books to read
recipes
foraging
bass tabs
EEB population ecol...
nre 563 final paper
(19) mail.riseup.net...
577 Hawaii links
police abolition
mental health stuff
mpdyq0zg62j41.jpg...
You are the only one in the meeting
Invite more people
meet.jit.si is sharing your screen.
Stop sharing
Hide

What is a function?

- $f(x) = ax + b.$
- It receives an **input**.
- It produces an **output** following a certain rule.
- Function in programming is a generalization of this mathematical concept.

What is a Function in Programming?

- Functions are individual building blocks of your program that accomplish specific tasks.
    - Function helps you divide your code into smaller, more **manageable and readable** pieces.
    - Code in a function is only executed when its host function is "called".
- Some functions take *input arguments* from the caller.
- Some functions return *an output value* to the caller after all its code are executed.
- Some functions do not have input or output.

What is a Function in C?

In the example C code, we have already seen a function, `main`.

```
...
void main(){
    printf("Hello World! \n");
}
```

In this example, we **defined** a function `main` in which we called another function `printf` to display a string.

- All C program starts at the beginning of `main`.

How to Write a Function Definition?

1. A function definition starts by indicating the **return type** (`void` means no return value will be produced.).
2. Followed by the **function name**.
3. The **Input arguments** come after that, inside `(` and `)`.
4. The **body of the function** is enclosed by `{` and `}`.

```
...
return_type function_name(input argument deceleration){
    function body
}
```

Your Own Function

You can write your own function and call it from `main()`:

```
...
void sayhello(){
    printf("Hello World!\n");
}
void main(){
    sayhello(); //calling "sayhello" function.
}
```

- You can choose your own name for your function, but it should be **succinct, reflects what your function does**.
    - If possible, use a short phrase starting with a verb.
    - e.g. sayhello, sort, additem, deleteitem, etc.

Your Own Function 2

Below is an example calculating circumference of a circle:

```
...
double calculate_circumference(double radius){
    return 2.0*3.1415926*radius;
}
void main(){
    printf("%f\n", calculate_circumference(2.0))
}
```

- It takes one input argument `radius`, declared as type `double`.
- `double` type means high-precision float point number.
- It **returns** a float point number whose type is also `double`.
- It is called by `main`, who will collect its returned value.

`main` with Inputs and an Output

The `main` function can also take inputs and return output.

```c
#include <stdio.h>
int main(int nargs, char* args[]){
    printf("%s\n", args[0]);
    return 0;
}
```

- `main` takes two inputs: `nargs` and `args`. These are passed on to `main` **from** the OS.
- In this example, `main` returns an integer value. It returns 0 if succeeded, otherwise, returns a non-zero value.
- We will talk about this usage more later.

Function Body

- The function body may contain any statement.
- However, the convention is

```
... function_name(...){
    declaration of variables
    ...

    other statements
}
```

Example

```c
#include <stdio.h>
double calc_gravity(double dist){
    //declare variable m1, m2, G and gravity.
    double m1 = 1.0;
    double m2 = 2.0;
    double G = 6.674E-11;
    double gravity;

    //compute "gravity" using declared variables.
    gravity = G*m1*m2/dist/dist;
    return gravity;
}
int main(){
    printf("%E", calc_gravity(1.256));
    return 0;
}
```

Declarations

- Variable in C is a placeholder of some value.
- The value held by a variable can be changed later.
- In C programming language, all variables must be **declared**.
- The syntax of declaration is: `data_type variable_name`.
    - Declaration: `double gravity;`
    - Declaration **with initializations**: `double m1 = 1.0;`
    - Declaration of multiple variables of the same type: `double m1, m2;`.
    - Declaration of multiple variables of the same type with initializations: `double m1 = 1.0, m2 = 2.0;`.

Data Types in C

- Some data types are:
  - `int` or `long` : integers
  - `float` or `double` : float point numbers
  - `char` : characters
- Each declaration tells the compiler: "reserve __ bytes of memory for this variable when function is running!".
- On modern PCs (and most smartphones):
  - `int` and `float` occupies 4 bytes of memory.
  - `long` and `double` occupies 8 bytes of memory.
  - `char` occupies 1 byte of memory.

Data Types in C

- Obviously, `long` and `double` are **more expressive** than `int` and `float`, but uses more memory.
- `int` has a range of -2147483648 to 2147483647.
- `long` has a range roughly plus or minus 9 quintillion
- `double` has about 15 decimal significant digits of precision, and has a range of about $\pm 10^{\pm 308}$
- If your memory space is precious, in applications such as computer graphics or data science, you can use `float`.
- We will see how the memory spaces are allocated for variables declared in functions later.

Declarations

Once a variable is declared, **the variable is assigned a memory location by the compiler.** If the variable is uninitialized, **the variable can contain whatever (rubbish) value** that is already in that memory location!

- Undefined value leads to all sorts of unpredictable behaviors and is a source of error!
- It is allowed by the compiler so you will not see an error or warning during the compilation!

```c
#include <stdio.h>
void main(){
    // It will print out some garbage value.
    int a;
    printf("%d\n", a);
}
```

Declarations

- If possible, initialize the variable when it is declared.

```
...
double calc_gravity(double dist){
    //declare variable m1, m2, G and gravity.
    double m1 = 1.0, m2 = 2.0;
    double G = 6.674e-11;
    //initialize gravity as soon as it is declared.
    double gravity = G*m1*m2/dist/dist;
    return gravity;
}
...
```

- If you do not know how to initialize the variable, assign an "default value" (e.g. 0), so that when you see the value, you know this variable has not been assigned any useful value.

Expressions and Assignments

- Computing operations are done using **expressions**:
  - `G*m1*m2/dist/dist`
  - Each expression has a value. The value of `G*m1*m2/dist/dist` is its computation outcome.
  - There are some expressions whose values are less obvious (we will see later).
- `m1 = 1.0, m2 = 2.0`, `gravity = G*m1*m2/dist/dist` are all **assignment expressions**.
  - It assigns the value of expression on the RHS to the variable on the LHS.
  - **The equality sign does not represent equality**.

Calling a Function

- You can call a function and obtain the returned value **after its definition**.
- Calls are made using the function name with all input values **in the same order they are declared**!

```c
#include <stdio.h>
double calc_gravity(double m1, double m2, double dist){
    //declare variable G and gravity.
    double G = 6.674e-11;
    return G*m1*m2/dist/dist;
}
int main(){
    // which one of the following numbers
    // corresponds to dist?
    printf("%E\n", calc_gravity(1.0, 2.0, 1.256));
    return 0;
}
```

Calling a Function Before It's Defined

- If you want to call a function before its definition, declare it before you call it!
- Declaration of a function is simply the definition without everything inside the brackets (and the brackets):
  `return_type function_name(input_arcuments);`

```c
#include <stdio.h>
// declaration of foo, tells the compiler
// what are foo's output and input types.
int foo(int a, int b);
void main(){
    //This works!
    printf("%d\n", foo(1,2));
}
int foo(int a, int b){return a+b;}
```

Calling a Function: Recursion

- A function can call itself!
  - This is called **recursive call**.

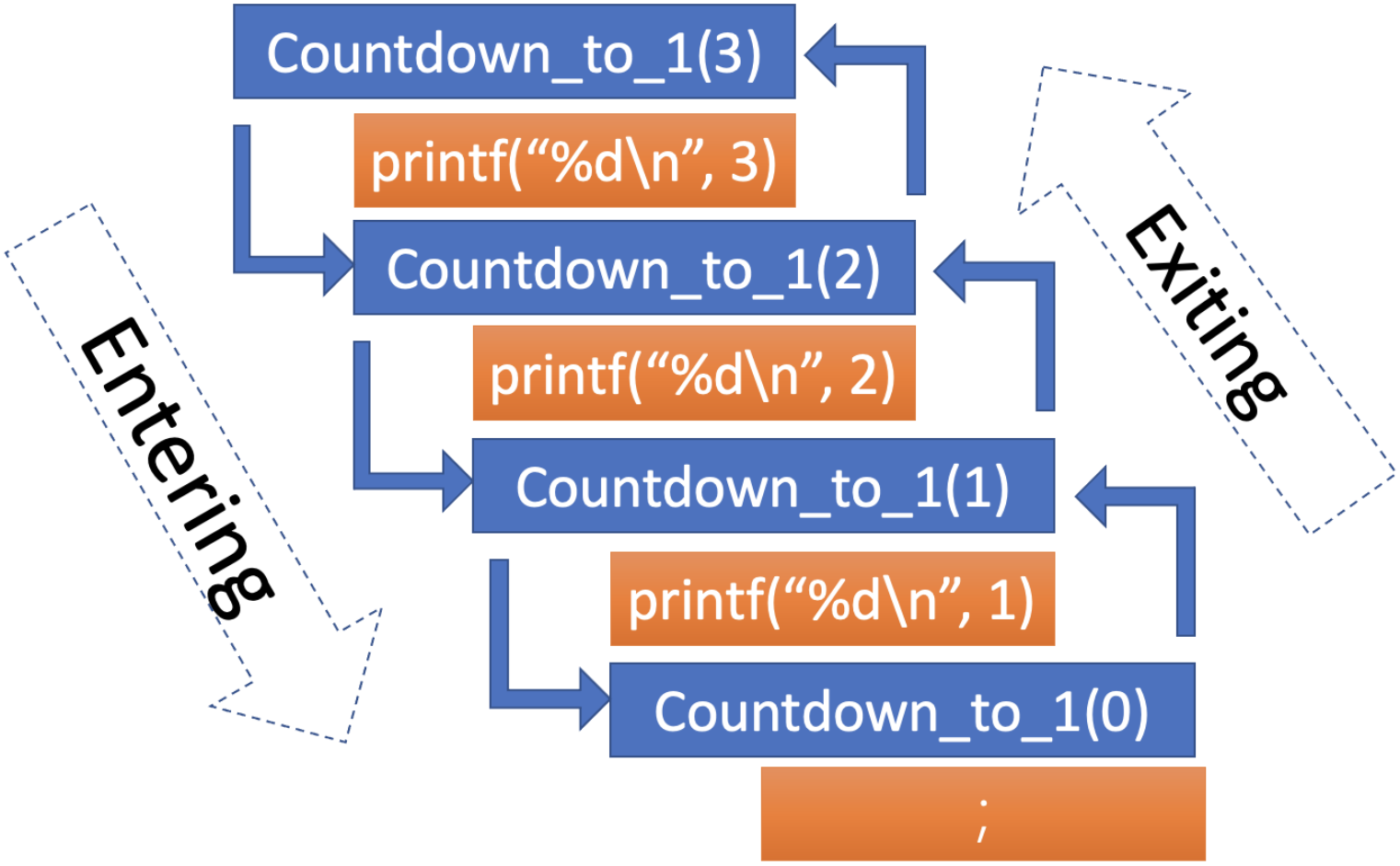
```c
#include <stdio.h>
void countdown_to_1(int n){
    if (n<1) {;}
    else{
        printf("%d\n",n);
        countdown_to_1(n-1);
    }
}
void main(){
    //It prints 10, 9, 8 ... 1
    countdown_to_1(10);
}
```

Calling a Function: Recursion

- If-Else is a flow control statement.
- `if (expr) {statements1} else {statements2}`
  - If `expr` is true, `statements1` is executed. Otherwise, `statements2` is executed.
- `countdown_to_1` works as follows:
  - If the input is smaller than 1, do nothing.
  - Otherwise, print the current number `n` and initiate the countdown from `n-1` by calling itself.
- Recursive function is very useful and intuitive when the solution to a problem involves solving the same problem at a smaller scale repeatedly.

Calling a Function: Recursion

Recursive function must have an entering and exiting path.



Calling a Function: Recursion

- Recursive function must have a exiting condition:
  - It tells the function when to stop its recursive calls.
  - In the example, we stopped recursion when `n<1` .
  - Recursive call without an exiting condition will raise an execution error (commonly referred to as a "crash").

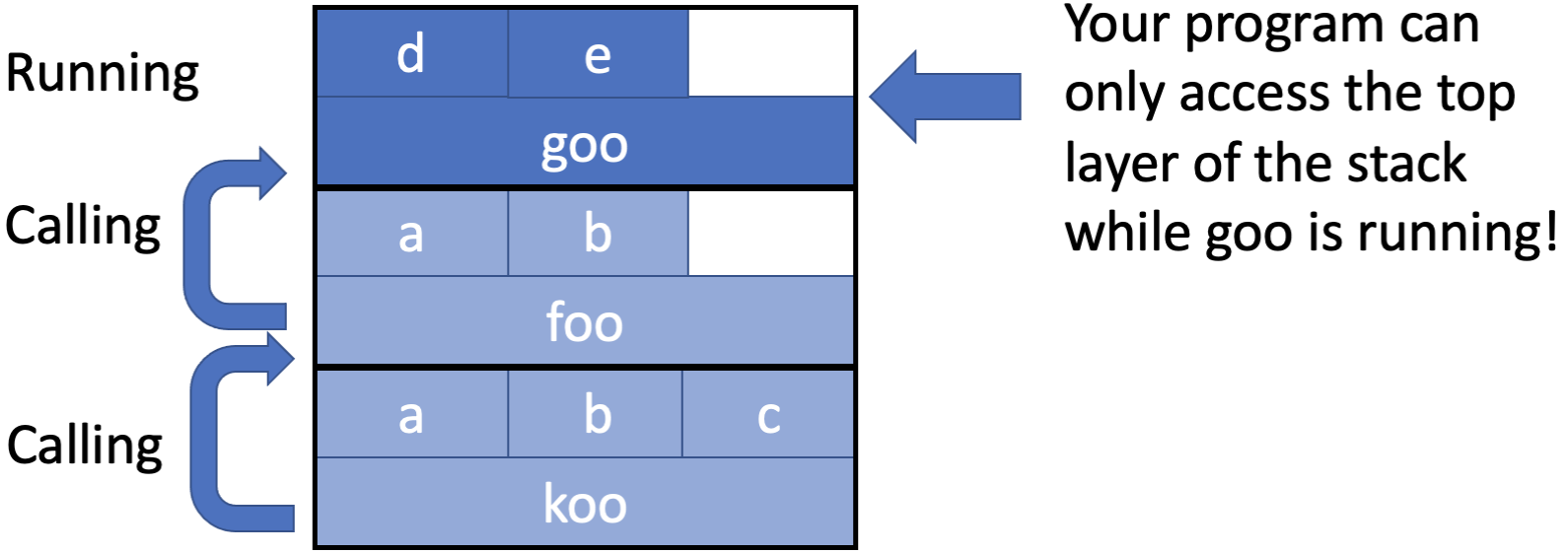
```c
#include <stdio.h>
void countdown_to_1(int n){
    printf("%d\n",n);
    countdown_n_to_1(n-1);
}
void main(){
    //It prints 10, 9, 8 ... 0, -1, ...
    // segmentation fault!
    countdown_to_1(10);
}
```

Stack Memory Allocation

- When the function is being executed on the CPU, its data (such as variables declared in the function) are temporarily stored in the memory.
- The memory region for storing function data in the current program is called "stack".
- When a function is called, its data is added to the top of the stack. You program can access them.
- When a function finishes its execution, its data is removed from the stack and the space it occupies is freed for future calls of functions.

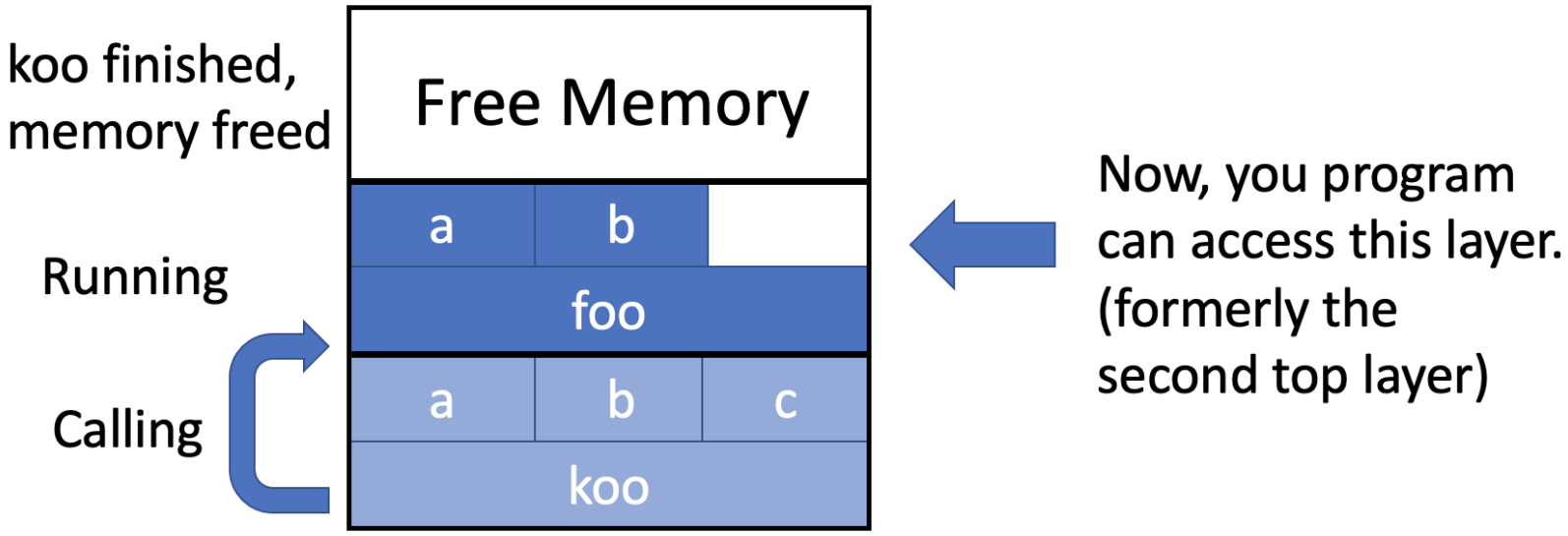
Stack Memory Allocation

Consider a situation where function `koo` calls `foo` and `foo` calls `goo`. Below is the stack while `goo` is running.



Stack Memory Allocation

When `goo` finishes running, its memory is freed.



When `foo` finishes running, its memory will be freed too and only variables in `koo` will be accessible.

Local Variables

Variables declared inside the function are called **local variables** (This includes all input argument variables!).

- They can only be accessed by the program when **the function is running**.
- **Why?** The program can only access the top layer of the stack, which stores variables of the function that is **currently running**.
- In the "koo-foo-goo" example, your program cannot access a and b defined in `koo` while `foo` is running.

Stack Memory Allocation

Stack is a highly efficient memory allocation/release mechanism.

- The memory allocation and release are all automatically handled by the OS.
- However, there is only a limited stack space for each program (determined by the OS). If a single function occupies a large memory space, or the call stack gets too "tall", we may run out of stack memory and an execution error will be raised by the OS.
    - This out-of-memory error is called "Stack Overflow".
    - Can you guess why the "recursion without an exiting path" example gets an execution error?

To Sum Up

- Writing functions are great ways to split your program into smaller, and more specific tasks.
- Functions can take input variables and return output value.
- Function body includes variables declarations.
- Functions are called using their names, with input variables in the same order arranged in the function definition
- Functions are temporarily stored in the "stack" when the code is running.

Homework

0. Read this tutorial on If-Else statement.
    i. What does it mean when we say an expression is true?
    ii. What is "If-Else" ladder?

Homework

1. Write a function, takes 2 integer inputs `n` and `t`. If `n` can be divided by `t`, output 1. Otherwise, output 0.
    i. Hint: modulus operator in C is `%`.

```c
#include <stdio.h>
void main(){
    printf("%d\n", 10%3);
} //display "1".
```

Homework

2. (Submit) Write a function, takes 2 integer inputs `x1`, `x2` and `r`.
    i. Output integer 2 if the 2D point (x_1, x_2) is **strictly** inside a circle which centers at the origin with the radius. `r`.
    ii. Output integer 0 if the 2D point (x_1, x_2) is **strictly** outside of the aforementioned circle.
    iii. Output integer 1 if the 2D point (x_1, x_2) is on the circle boundary.
    iv. Write a `main` function, including test calls of your function **covering all above scenarios** and print out the output of each call using `printf`.

Homework

3. Change the countdown code slightly, so that it prints out 1, 2, 3... 10. (Keep the modification as simple as possible!).

Homework

4. (Submit) Write a function, takes integer inputs `n` and `t`, `t<n`. Returns the number of integers from `2` to `n` which can be divided by `t`.
    i. Use recursion (see the skeleton below),
    ii. Do not use loop.

Homework

- Using the following skeleton function (filling the blanks)

```
int count(int n, int t){
    if(n<t){
        return 0; // if n<t, no need to count.
    }else{
        if(n%t ==0){
            return _____;
        }else{
            return _____;
        }
    }
}
```

Homework

5. (Challenge) Slightly change the code of `count` so that it returns 0 if n is a prime and returns some positive value if n is NOT a prime.
6. (Challenge) Can you make use of the `count` function you just wrote in question 5, and write a function `int is_prime(n)`:
    i. Takes one input value `n`,
    ii. Returns `0` if `n` is not a prime number, `1` if `n` is a prime number.