CSE 3100: Systems Programming

Lecture 4: Functions and Global Variables

Department of Computer Science and Engineering

University of Connecticut

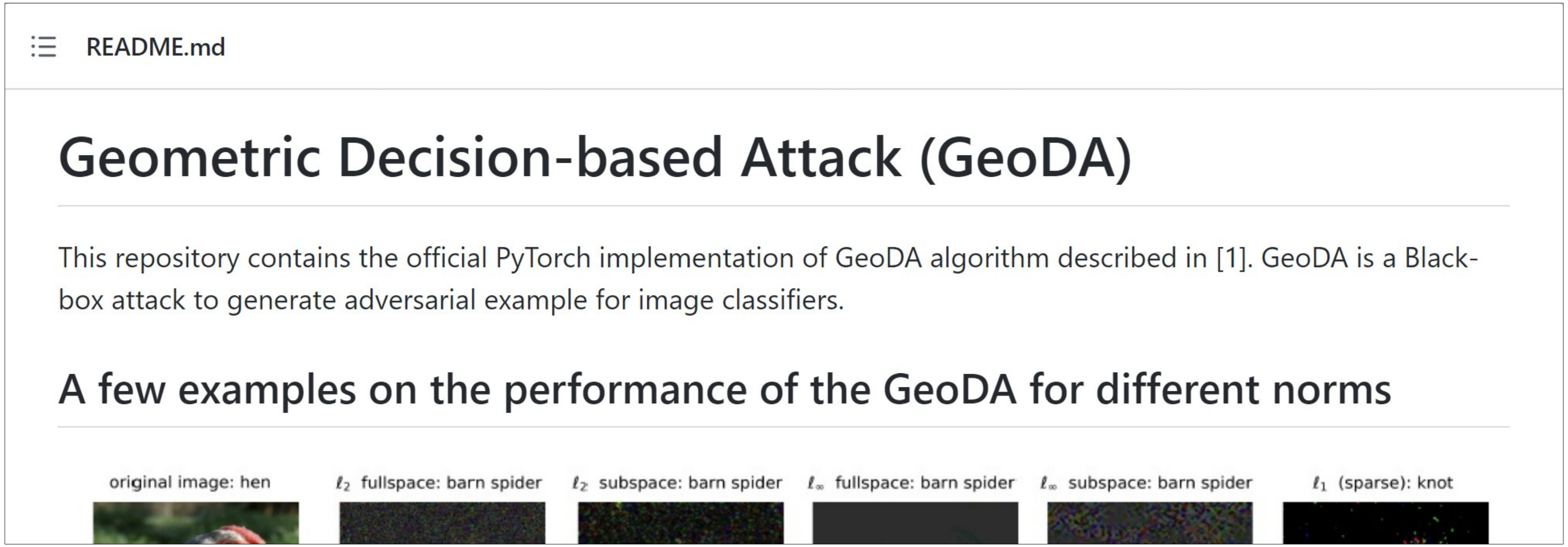
1. **The Basics of Functions**
2. **Types of Variables in C**
3. **Memory Organization**

1. The Basics of Functions

Function Story Time

- Once upon a time Professor Kaleel was a graduate student writing papers on machine learning.

Professor Kaleel in his youth:



README.md

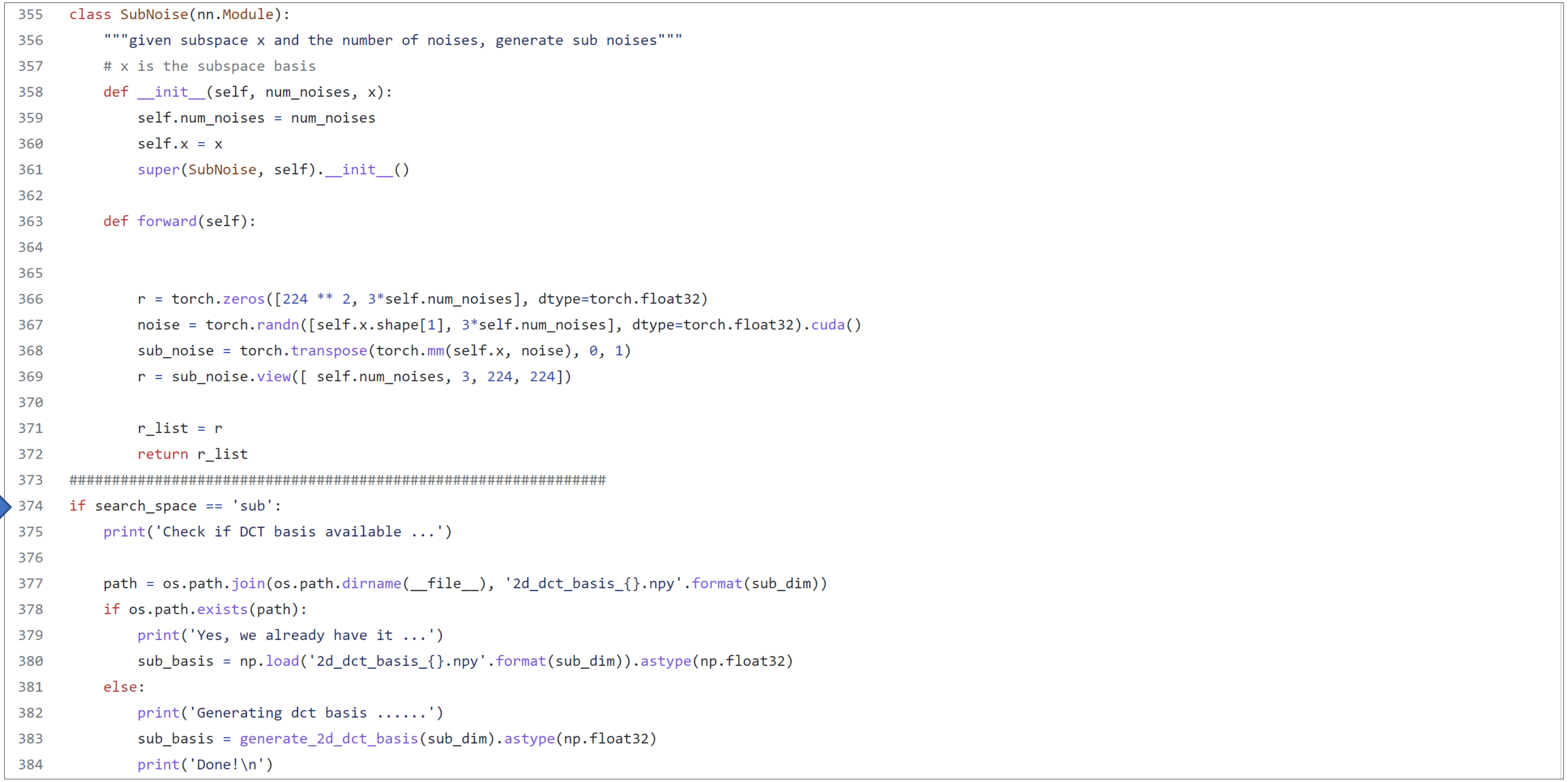
Geometric Decision-based Attack (GeoDA)

This repository contains the official PyTorch implementation of GeoDA algorithm described in [1]. GeoDA is a Black-box attack to generate adversarial example for image classifiers.

A few examples on the performance of the GeoDA for different norms

original image: hen
ℓ_2 fullspace: barn spider
ℓ_2 subspace: barn spider
ℓ_∞ fullspace: barn spider
ℓ_∞ subspace: barn spider
ℓ_1 (sparse): knot

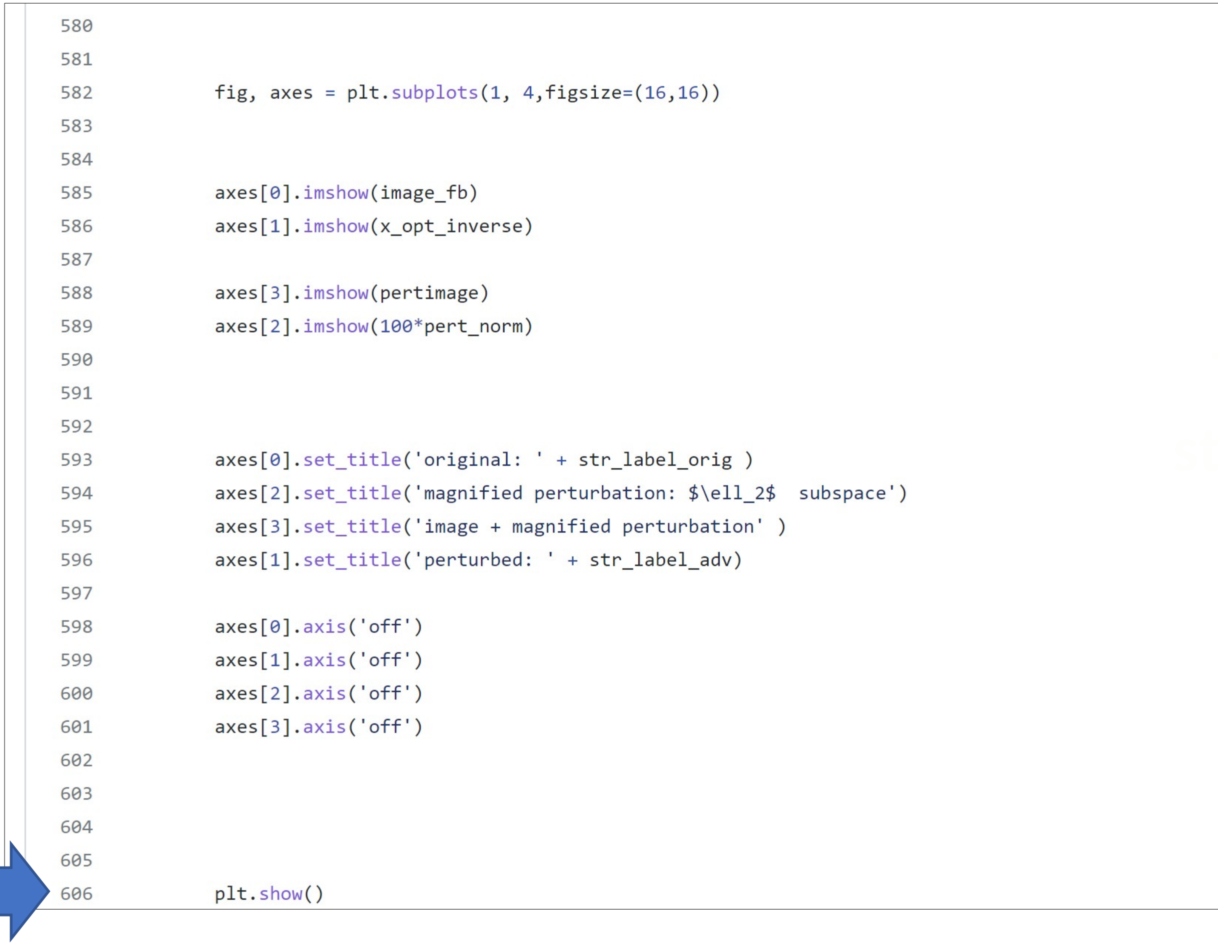
This is what the GeoDA code looks like:



```
355 class SubNoise(nn.Module):
356     """given subspace x and the number of noises, generate sub noises"""
357     # x is the subspace basis
358     def __init__(self, num_noises, x):
359         self.num_noises = num_noises
360         self.x = x
361         super(SubNoise, self).__init__()
362 
363     def forward(self):
364 
365 
366         r = torch.zeros([224 ** 2, 3*self.num_noises], dtype=torch.float32)
367         noise = torch.randn([self.x.shape[1], 3*self.num_noises], dtype=torch.float32).cuda()
368         sub_noise = torch.transpose(torch.mm(self.x, noise), 0, 1)
369         r = sub_noise.view([ self.num_noises, 3, 224, 224])
370 
371         r_list = r
372         return r_list
373 ################################################################
374 if search_space == 'sub':
375     print('Check if DCT basis available ...')
376 
377     path = os.path.join(os.path.dirname(__file__), '2d_dct_basis_{}.npy'.format(sub_dim))
378     if os.path.exists(path):
379         print('Yes, we already have it ...')
380         sub_basis = np.load('2d_dct_basis_{}.npy'.format(sub_dim)).astype(np.float32)
381     else:
382         print('Generating dct basis ......')
383         sub_basis = generate_2d_dct_basis(sub_dim).astype(np.float32)
384         print('Done!\n')
```

Main code starts on line 374.

And the code ends on line 606...

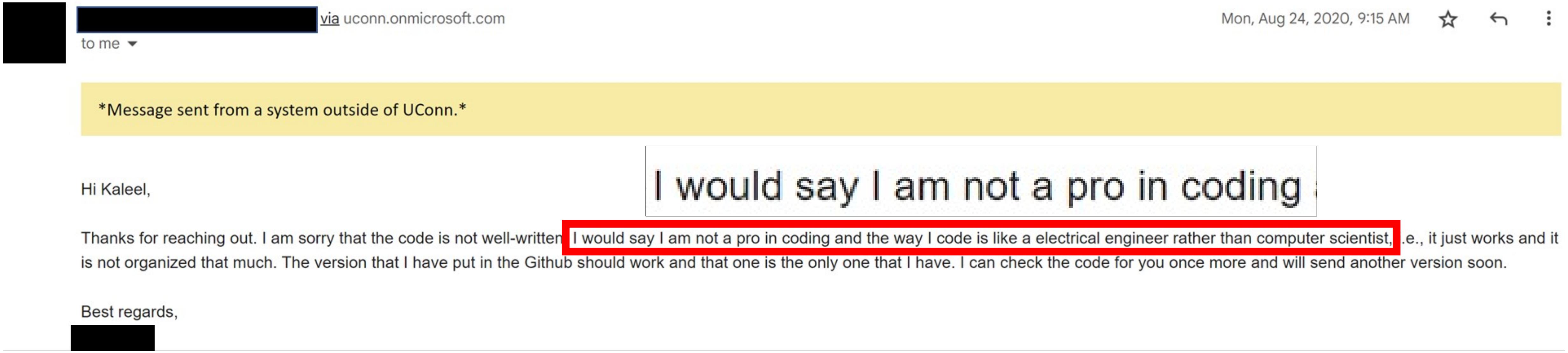


```
580
581
582         fig, axes = plt.subplots(1, 4,figsize=(16,16))
583
584
585         axes[0].imshow(image_fb)
586         axes[1].imshow(x_opt_inverse)
587
588         axes[3].imshow(pertimage)
589         axes[2].imshow(100*pert_norm)
590
591
592
593         axes[0].set_title('original: ' + str_label_orig )
594         axes[2].set_title('magnified perturbation: $\ell_2$  subspace')
595         axes[3].set_title('image + magnified perturbation' )
596         axes[1].set_title('perturbed: ' + str_label_adv)
597
598         axes[0].axis('off')
599         axes[1].axis('off')
600         axes[2].axis('off')
601         axes[3].axis('off')
602
603
604
605
606         plt.show()
```

End of the code.

That is 232 lines of straight code to read!

What was the code writer's excuse?



[redacted] [redacted] via uconn.onmicrosoft.com

to me

Mon, Aug 24, 2020, 9:15 AM

Message sent from a system outside of UConn.

I would say I am not a pro in coding

Hi Kaleel,

Thanks for reaching out. I am sorry that the code is not well-written. I would say I am not a pro in coding and the way I code is like a electrical engineer rather than computer scientist, i.e., it just works and it is not organized that much. The version that I have put in the Github should work and that one is the only one that I have. I can check the code for you once more and will send another version soon.

Best regards,

[redacted]

The End Result:

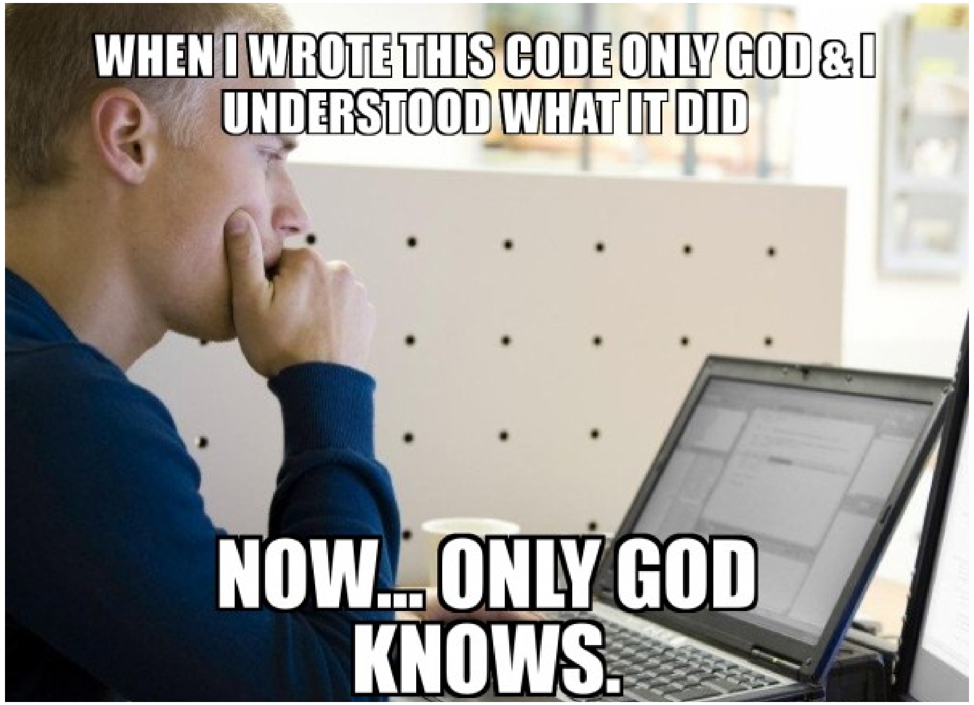
Our research group never used this person's code in our experiments.

We ended up using a competing paper's technique in our research.

Why? Because THE COMPETITOR'S CODE was written with functions.

The Importance of Functions

When you write code without functions:



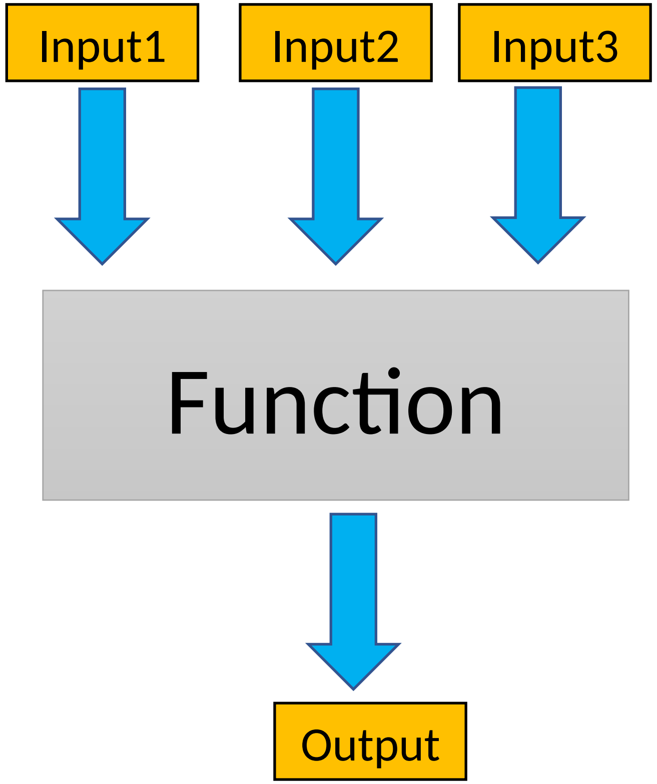
- Writing readable code in terms of functions is not some pie in the sky theoretical concept.

INDUSTRY: it *WILL* affect how easily other people can pick up and develop your code. It will affect how fast you can pick up old code and make progress.

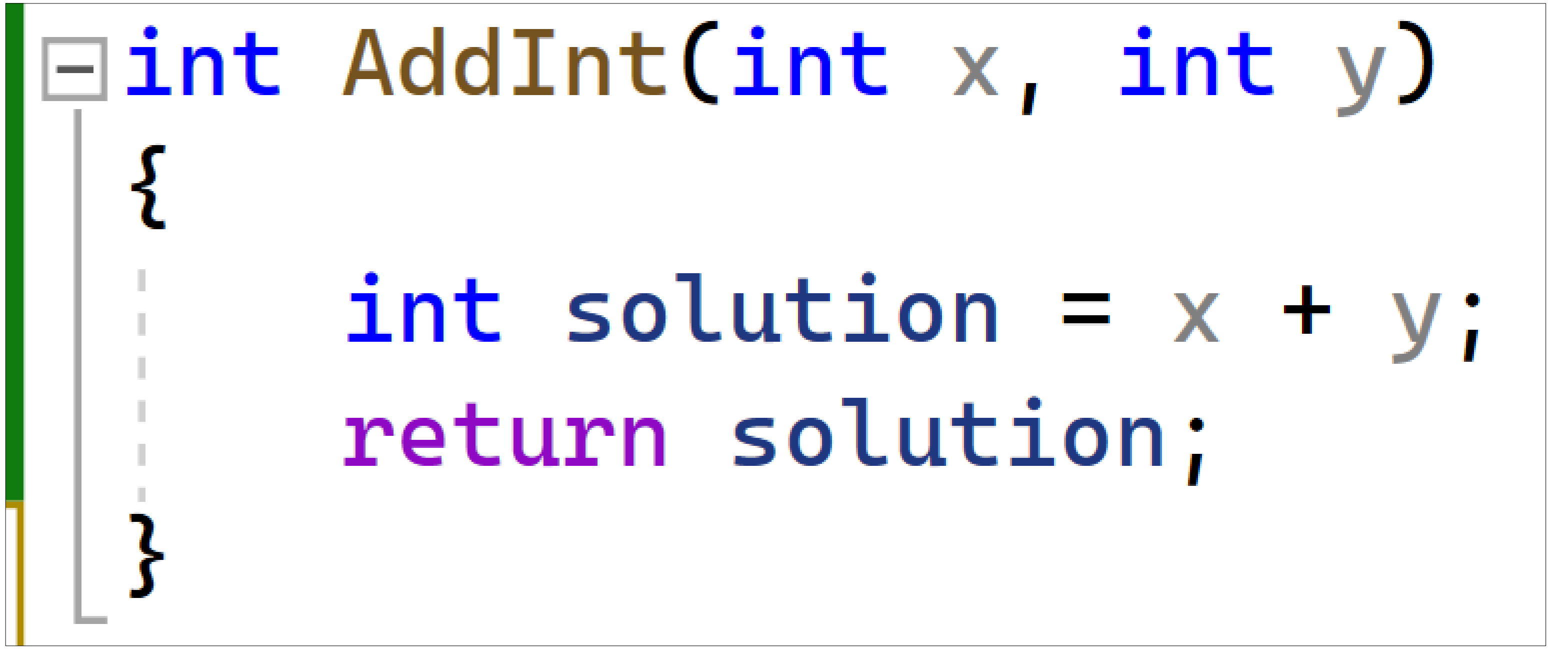
RESEARCH: it *WILL* affect who uses your code base and who cites you.

Functions: Building blocks for larger programs

- Effective programming requires problem decomposition into **manageable pieces**.
- A C program is made of functions
  - Starting from main()
- Idealized view: a function is a black box…
  - It SPECIFIES **what** the computation will do
  - It ABSTRACTS AWAY **how** the computational process works
  - It takes INPUTS
  - It produces an OUTPUT



Writing a function in C



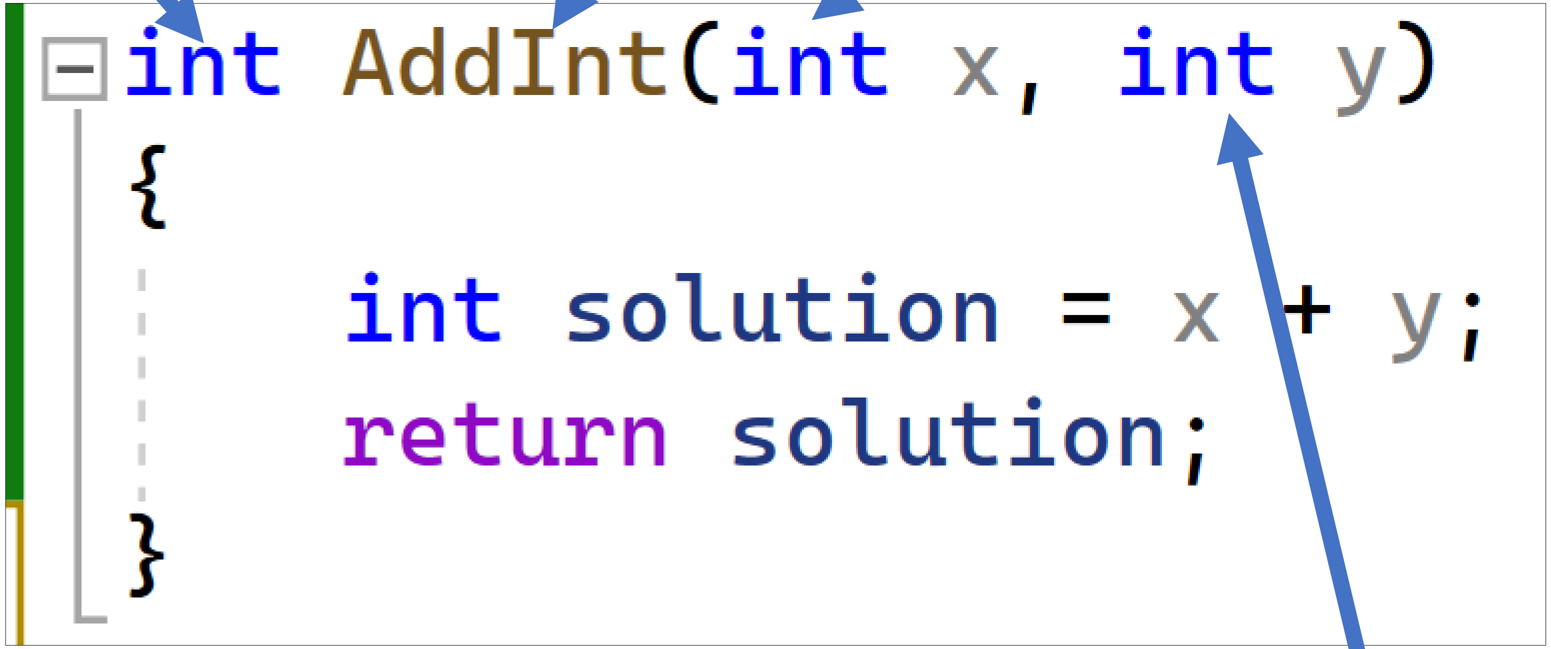
```c
int AddInt(int x, int y)
{
	int solution = x + y;
	return solution;
}
```

Writing a function in C

What type of variable the function returns

Name of the function

Variable type of the first input

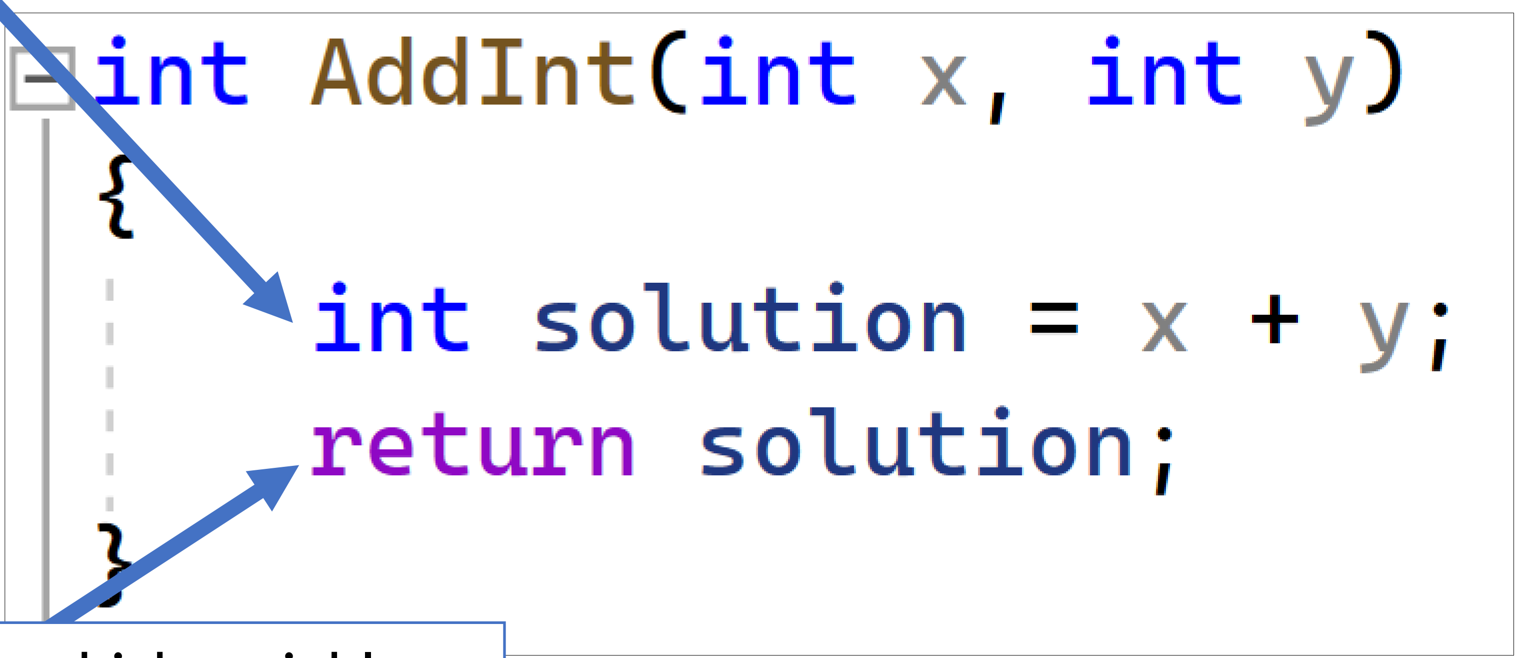


```
int AddInt(int x, int y)
{
    int solution = x + y;
    return solution;
}
```

Variable type of the second input

Writing a function in C: Return Statement

Body of the function to be executed.

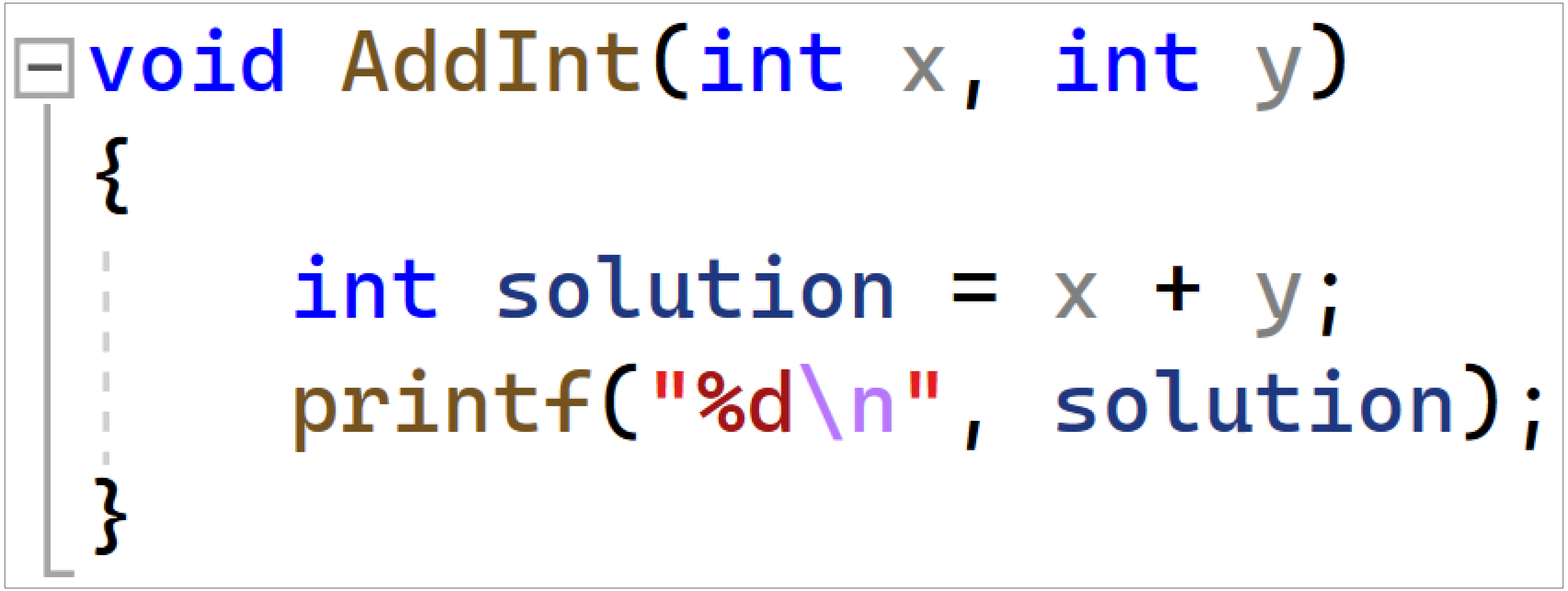


```
int AddInt(int x, int y)
{
    int solution = x + y;
    return solution;
}
```

Return indicates which variable should be passed out once the function ends.

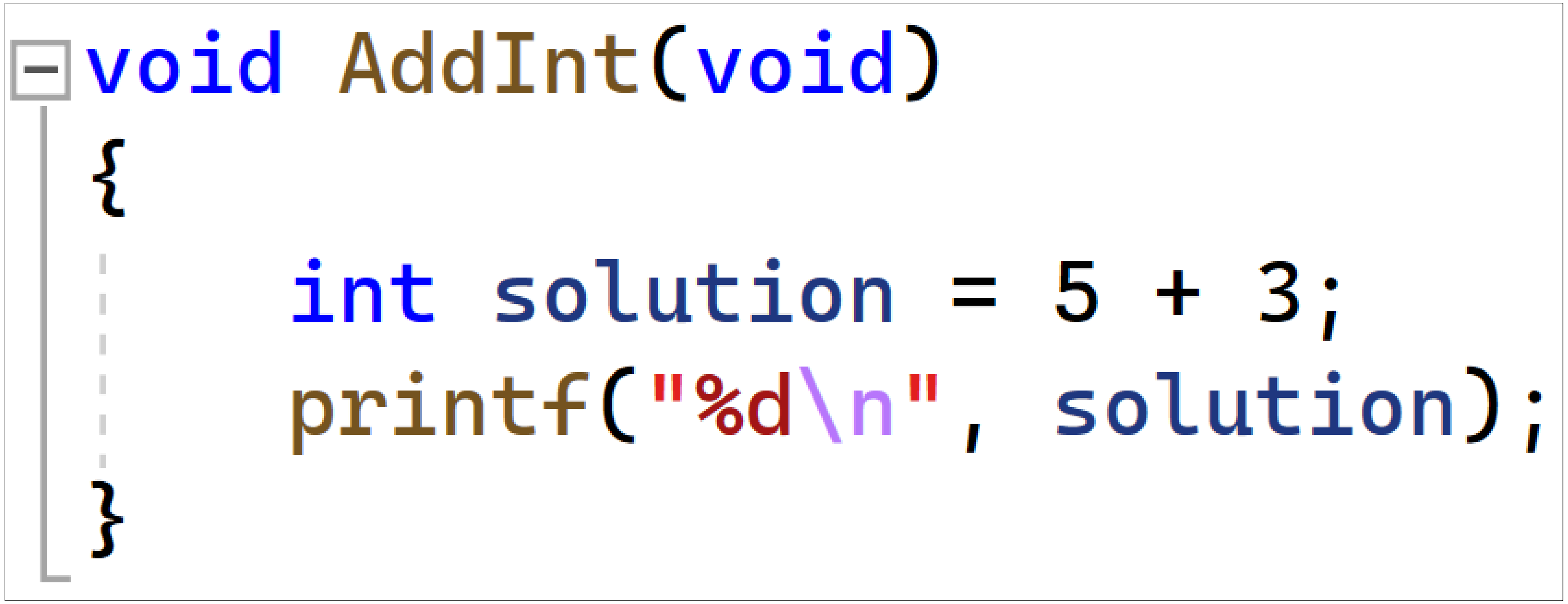
What happens if you don’t want the function to return anything?

Use the void keyword!



```
void AddInt(int x, int y)
{
    int solution = x + y;
    printf("%d\n", solution);
}
```

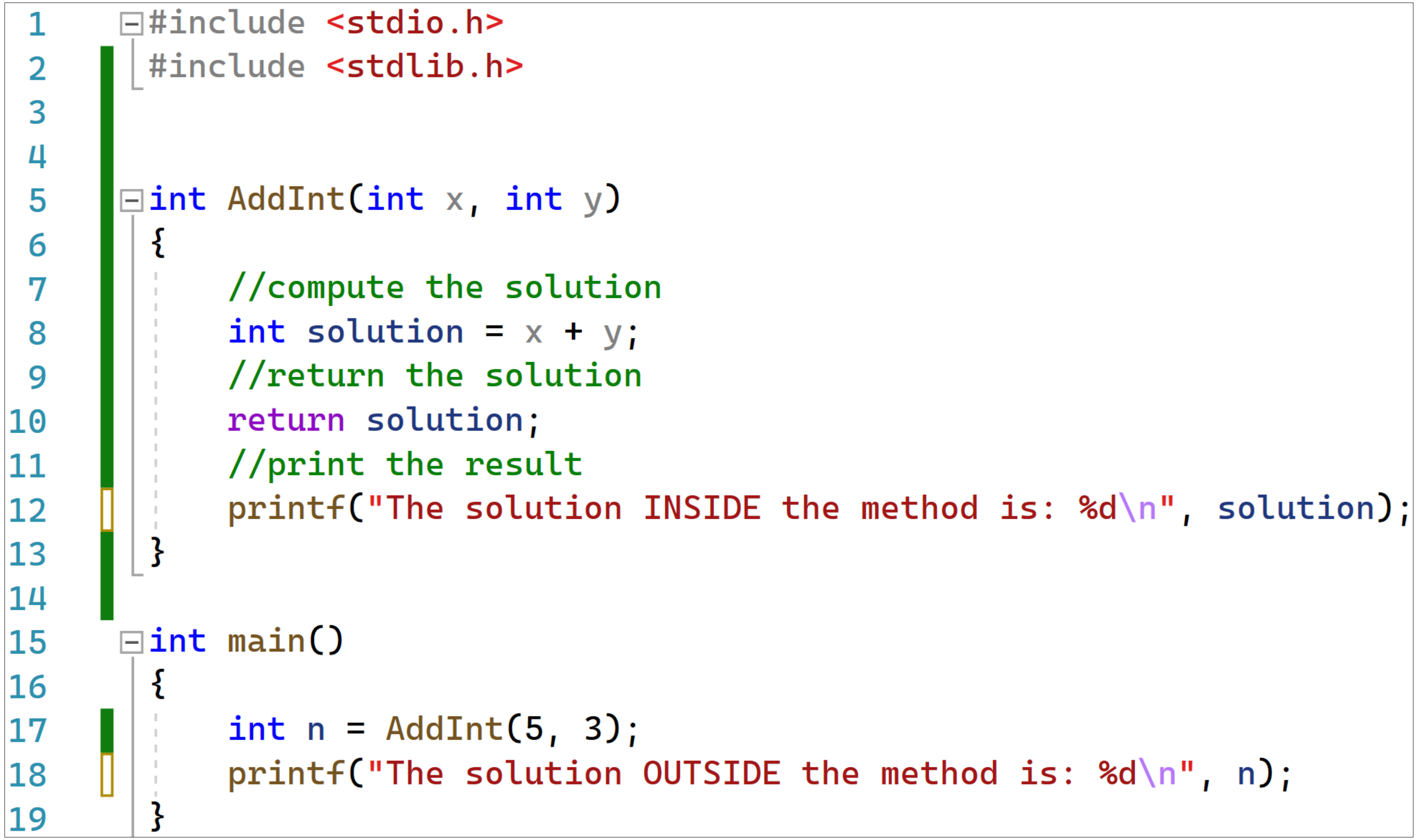
Side note: void can also be used if you don’t want to pass anything into the function



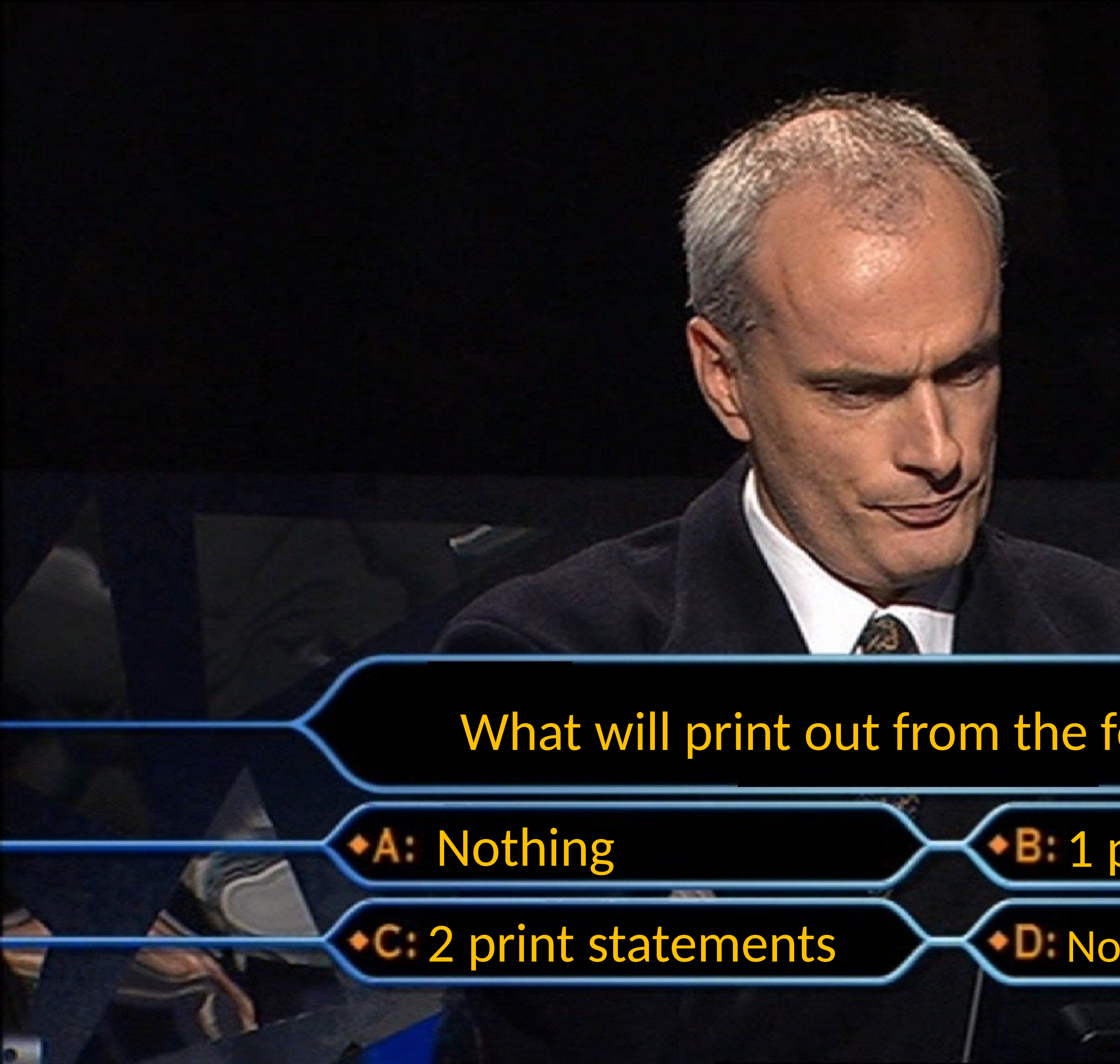
```
void AddInt(void)
{
    int solution = 5 + 3;
    printf("%d\n", solution);
}
```

Understanding the return statement

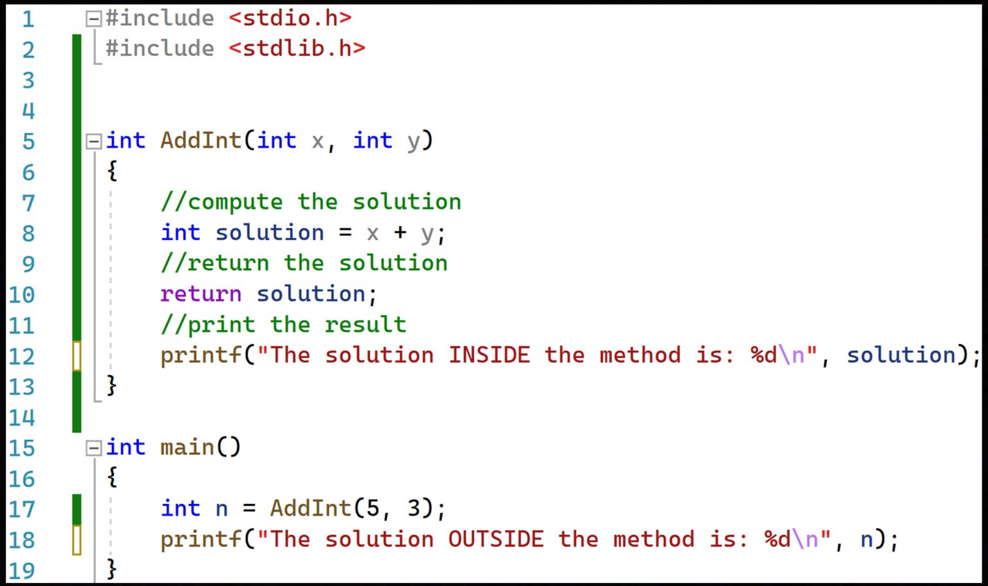
- Given the following code: *what will print out?*



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


int AddInt(int x, int y)
{
    //compute the solution
    int solution = x + y;
    //return the solution
    return solution;
    //print the result
    printf("The solution INSIDE the method is: %d\n", solution);
}

int main()
{
    int n = AddInt(5, 3);
    printf("The solution OUTSIDE the method is: %d\n", n);
}
```



What will print out from the following code?

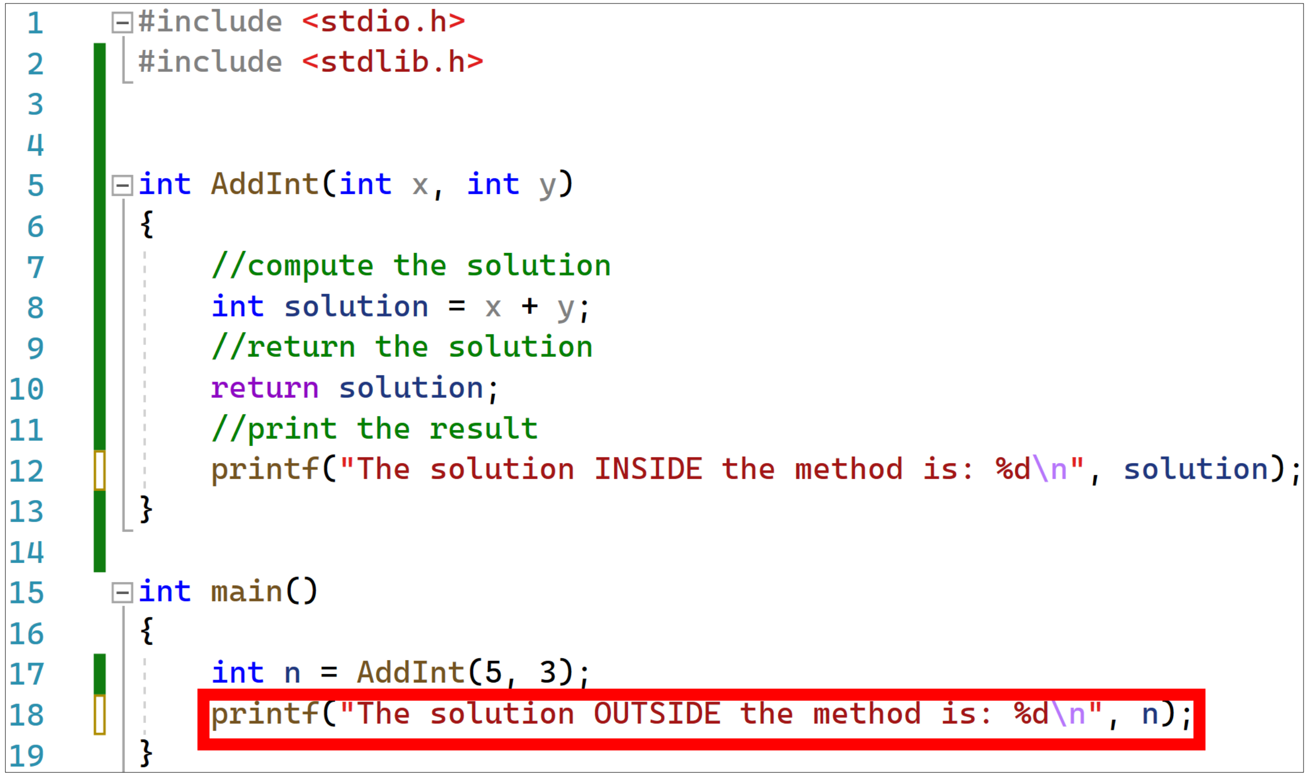


```c
#include <stdio.h>
#include <stdlib.h>


int AddInt(int x, int y)
{
    //compute the solution
    int solution = x + y;
    //return the solution
    return solution;
    //print the result
    printf("The solution INSIDE the method is: %d\n", solution);
}

int main()
{
    int n = AddInt(5, 3);
    printf("The solution OUTSIDE the method is: %d\n", n);
}
```

- A: Nothing
- B: 1 print statements
- C: 2 print statements
- D: Not sure. I am illiterate.

Understanding the return statement

Only the print statement highlighted in red will print. *Why*?

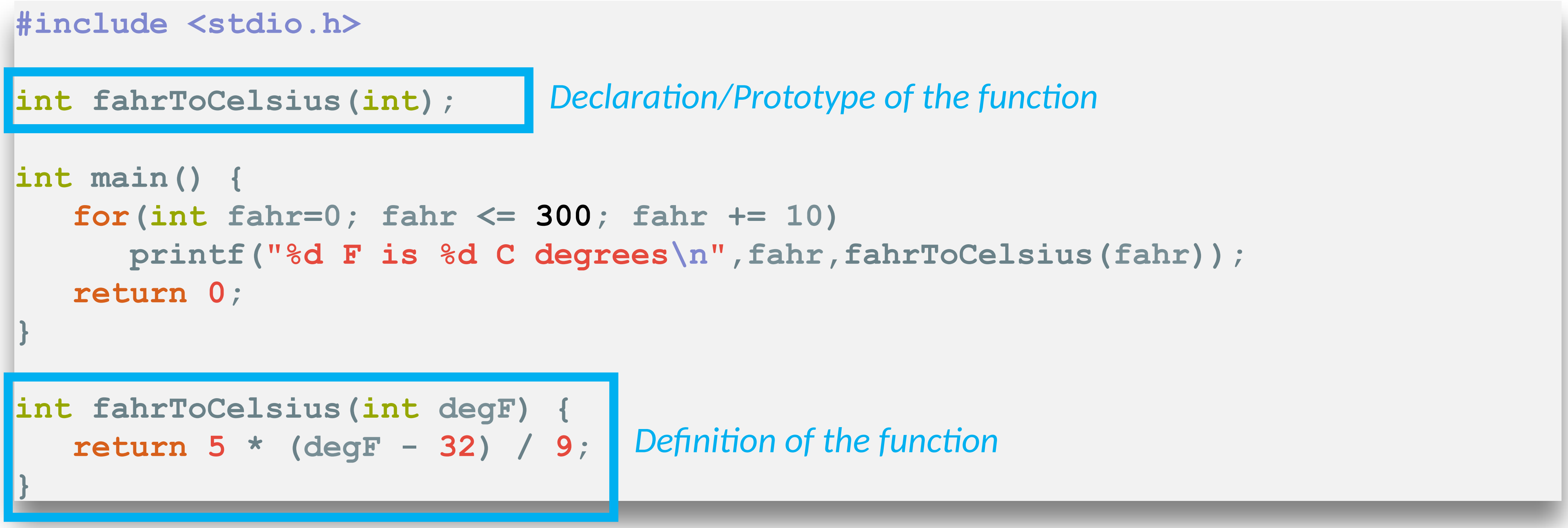


```c
#include <stdio.h>
#include <stdlib.h>


int AddInt(int x, int y)
{
    //compute the solution
    int solution = x + y;
    //return the solution
    return solution;
    //print the result
    printf("The solution INSIDE the method is: %d\n", solution);
}

int main()
{
    int n = AddInt(5, 3);
    printf("The solution OUTSIDE the method is: %d\n", n);
}
```

Once the return statement runs, the rest of the body of the function will NOT execute. We return to main.

Function Prototypes

- Functions can be defined in any order
  - Declare a function before first use if definition comes later
  - Function prototypes often placed in header files (and reused)

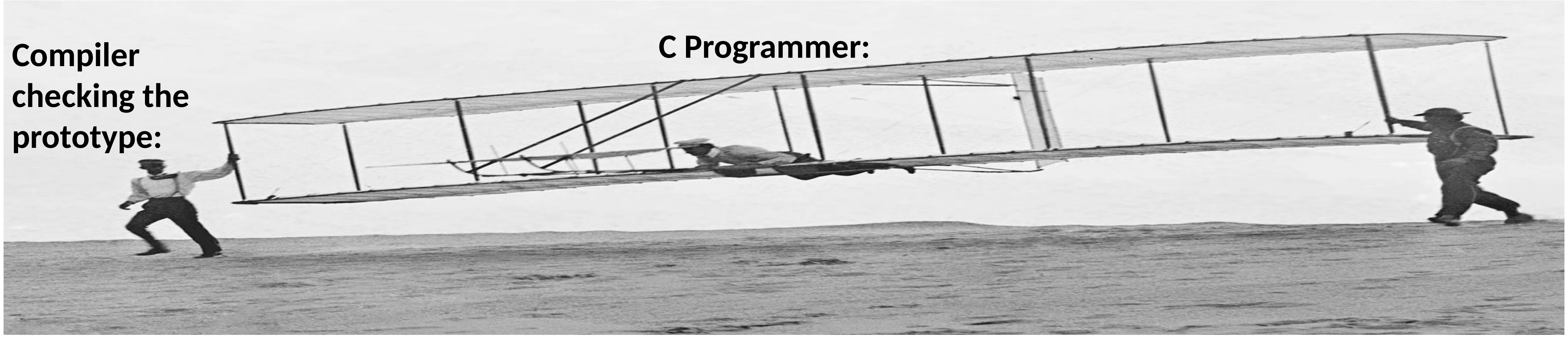


```c
#include <stdio.h>

int fahrToCelsius(int);

int main() {
   for(int fahr=0; fahr <= 300; fahr += 10)
      printf("%d F is %d C degrees\n",fahr,fahrToCelsius(fahr));
   return 0;
}

int fahrToCelsius(int degF) {
   return 5 * (degF - 32) / 9;
}
```

Declaration/Prototype of the function

Definition of the function

What is the purpose of function prototypes?



- A function prototype tells the compiler the number and type of arguments that are to be passed to the function and the type of the value that is to be returned by the function.
- Function prototypes allow the compiler to check the code more thoroughly.

Side Note 1: Macros in C

- You can actually put the entire function in a single line if you so desire:

```
#define MIN(x,y) ((x) <  (y) ? (x) : (y))
#define MAX(x,y) ((x) >= (y) ? (x) : (y))

int main()
{
     int a = 10, b = 20;
     int x = MIN(a,b);
}
```

Side Note 2: Recursion in C

C does support recursive programming…

Non recursive way to write power function:

```
int power(int base,int n) {
    int rv = 1;
    while (n>0) {
        rv *= base;
        n--;
    }
    return rv;
}
```

Recursive way to write power function:

```
int power(int base,int n) {
    if (n==0)
        return 1;
    else {
        int p = power(base,n-1);
        return base * p;
    }
}
```

Summary of the Basic Function Examples

```
type function_name ( parameter_list ) {
  declarations & statements
}
```

- No nesting (cannot define functions in a function)
- Return type can be “void” (no return value expected)
  - If missing, compiler assumes int
- Return statements:

```
return;          // terminate execution and return control to caller
return expr;     // terminate and pass value of expr back to caller
```

- Execution also terminates if end of function body reached
  - Returned value undefined

2. Types of Variables in C

Types of Variables in C (based on scope)

1. **Local variables**
2. **Static local variables**
3. **Global variables**
4. **Static global variables**

Types of Variables in C: Local Variables

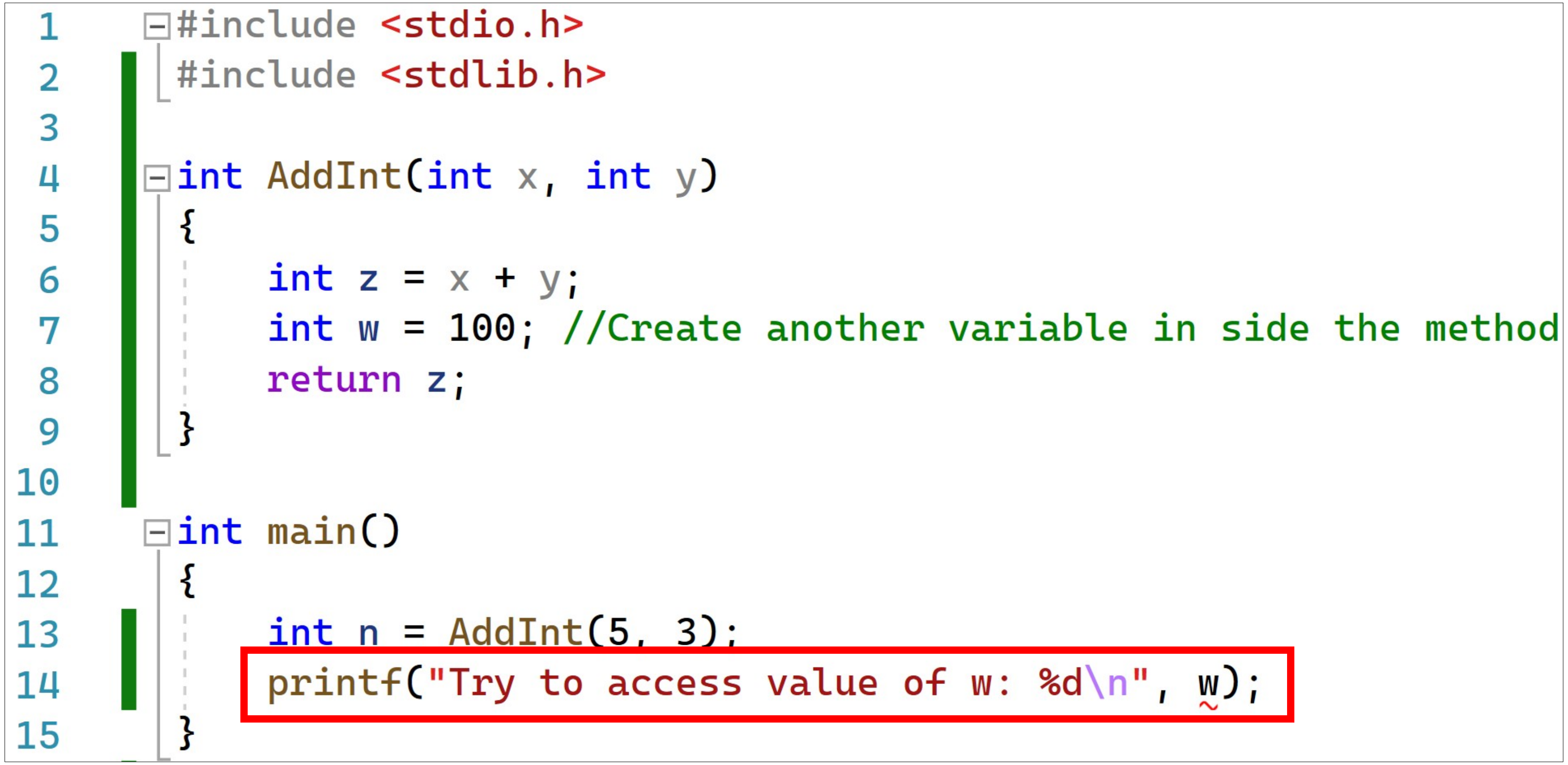
1. **Local variables**
   - Only visible inside the function.
   - Value is **not retained** across function calls.

I have no memory of this place...

Local Variable Code Example

- Local variables: only accessible in the method where they are created.

What happens if we try to print the value of w in the main?



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

int AddInt(int x, int y)
{
    int z = x + y;
    int w = 100; //Create another variable in side the method
    return z;
}

int main()
{
    int n = AddInt(5, 3);
    printf("Try to access value of w: %d\n", w);
}
```

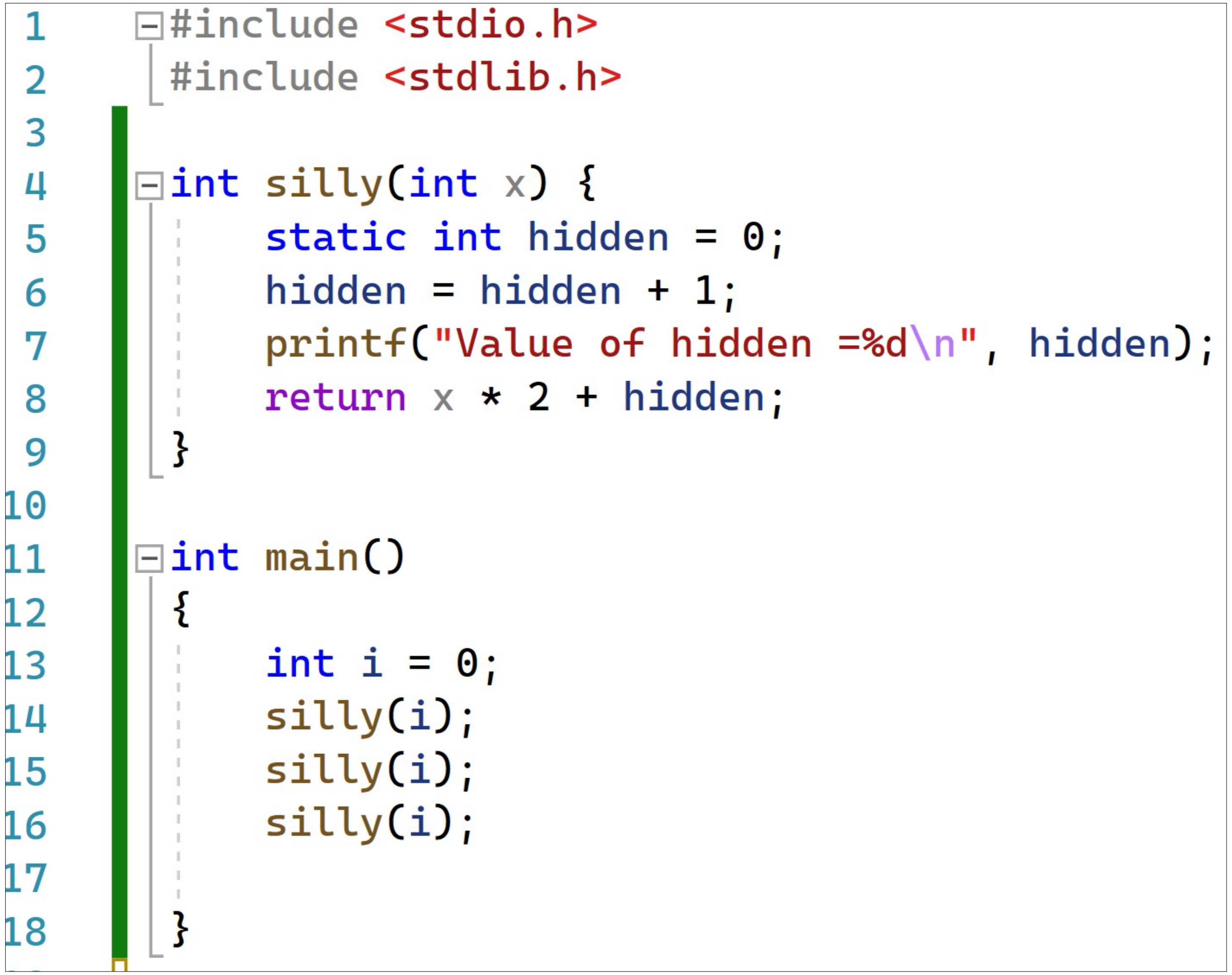
Types of Variables in C: Static Local Variables

2. Static local variables

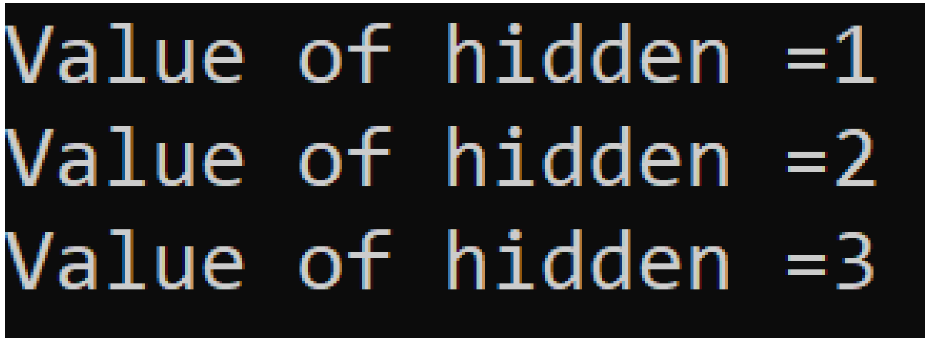
- Not visible outside function, but **retain** value across function calls.

Local Static Variable Code Example 1

What is the value of hidden after the code finishes?



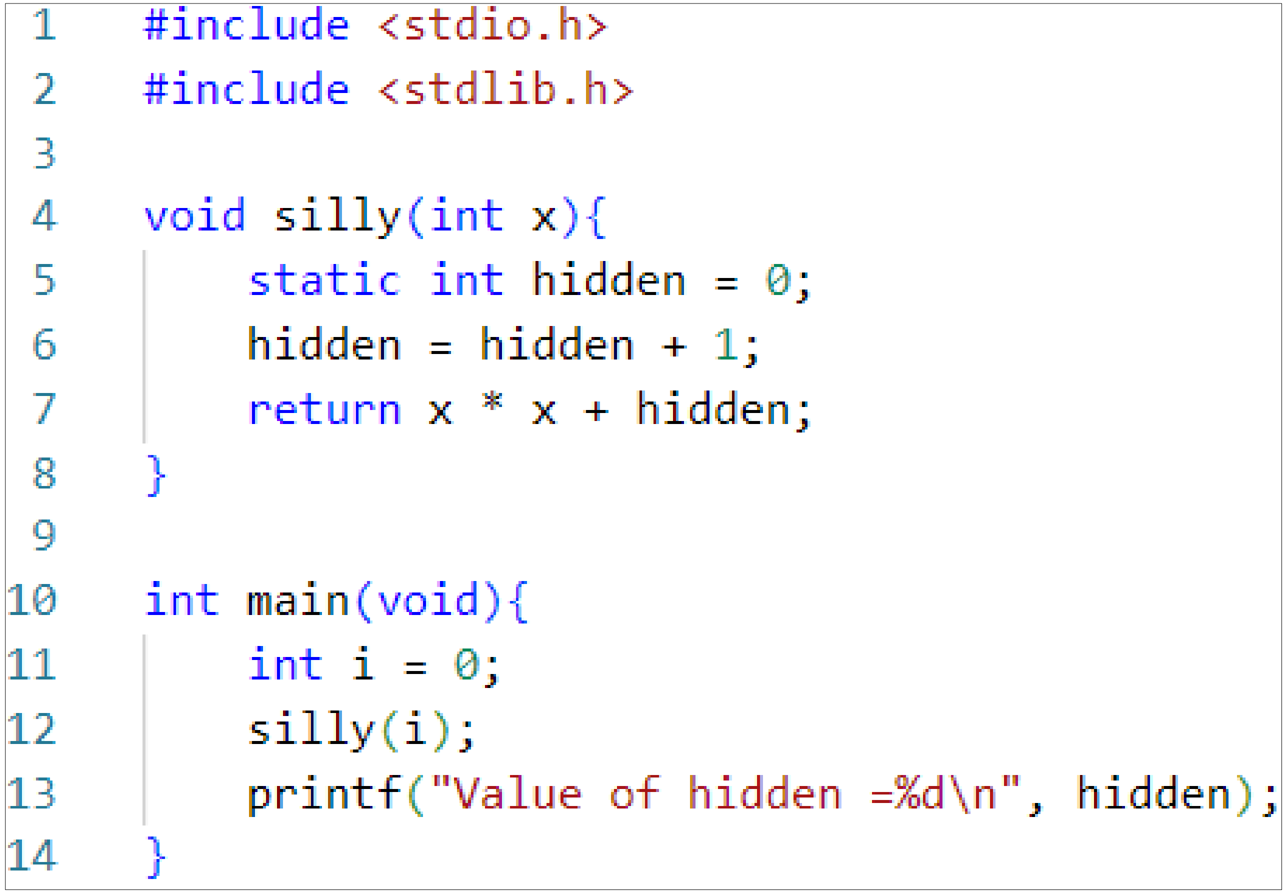
```c
#include <stdio.h>
#include <stdlib.h>

int silly(int x) {
    static int hidden = 0;
    hidden = hidden + 1;
    printf("Value of hidden =%d\n", hidden);
    return x * 2 + hidden;
}

int main()
{
    int i = 0;
    silly(i);
    silly(i);
    silly(i);

}
```



```
Value of hidden =1
Value of hidden =2
Value of hidden =3
```

Local Static Variable Code Example 2

What will be the output of this code?



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

void silly(int x){
    static int hidden = 0;
    hidden = hidden + 1;
    return x * x + hidden;
}

int main(void){
    int i = 0;
    silly(i);
    printf("Value of hidden =%d\n", hidden);
}
```

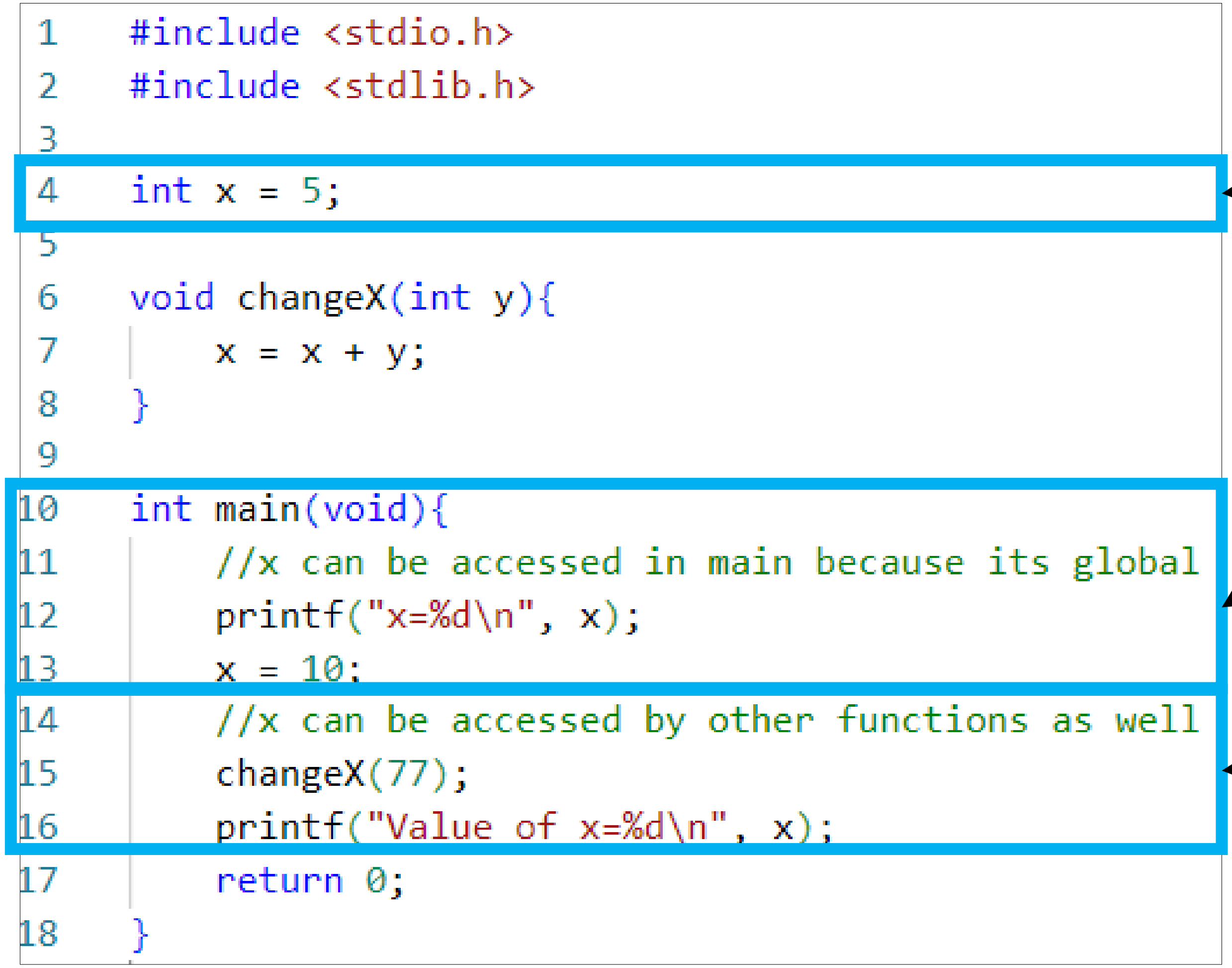
Won’t compile! Local static functions can’t be accessed outside of the function they are created in.

Types of Variables in C: Global Variables

3. Global variables

- Declared outside functions.
- Variable retained for entire duration of program.

Global Variable Code Example



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

int x = 5;

void changeX(int y){
    x = x + y;
}

int main(void){
    //x can be accessed in main because its global
    printf("x=%d\n", x);
    x = 10;
    //x can be accessed by other functions as well
    changeX(77);
    printf("Value of x=%d\n", x);
    return 0;
}
```

Declare global variable here.

Global variables can be accessed in main without being passed as a parameter.

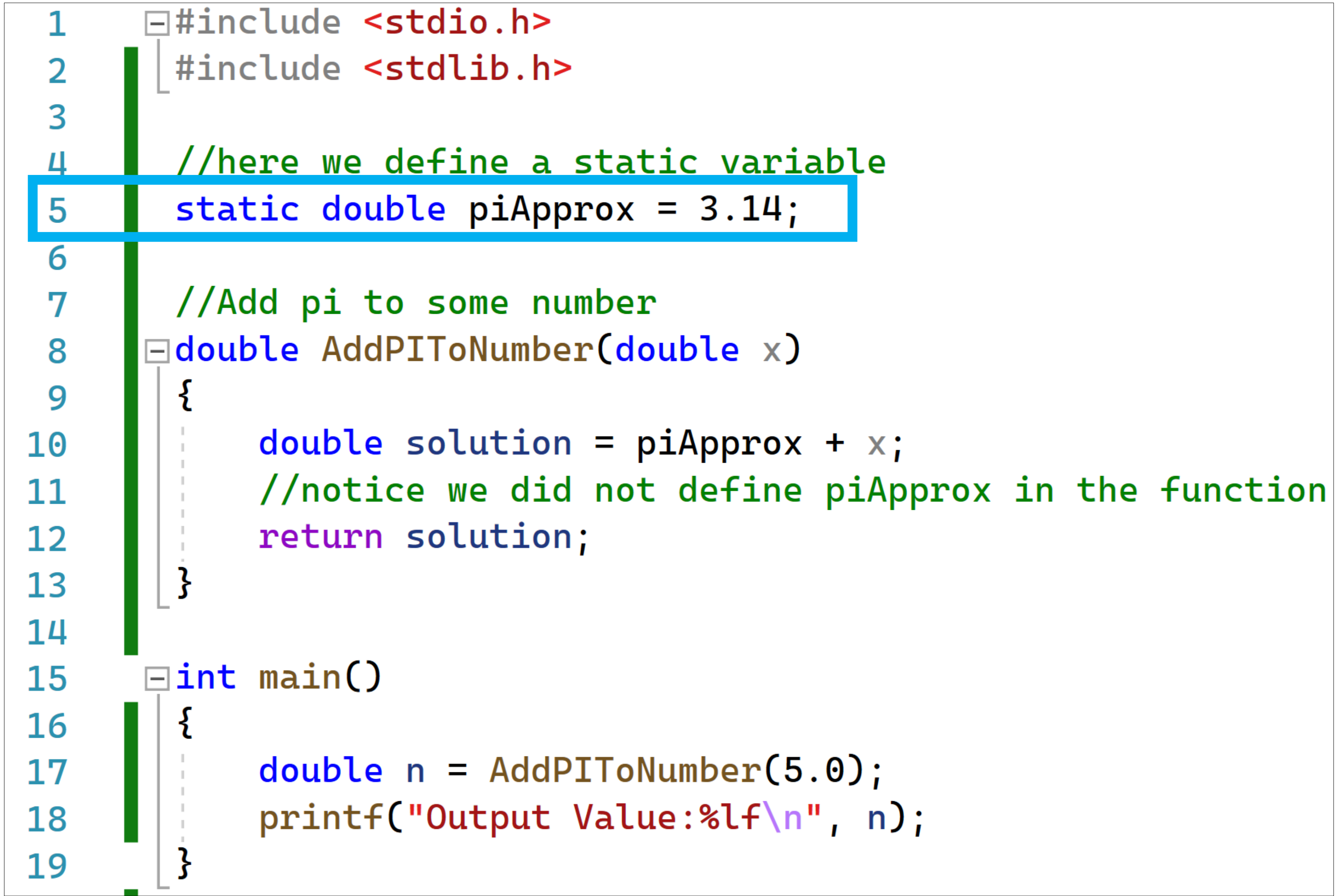
Global variables can be accessed by other methods without being passed as a parameter.

Types of Variables in C: Global Variables

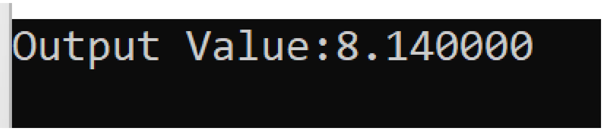
4. Global static variables

- Declared outside functions.
- Variable retained for entire duration of program.
- Visible only in functions defined in the same file following variable declaration.

Global Static Variable Code Example

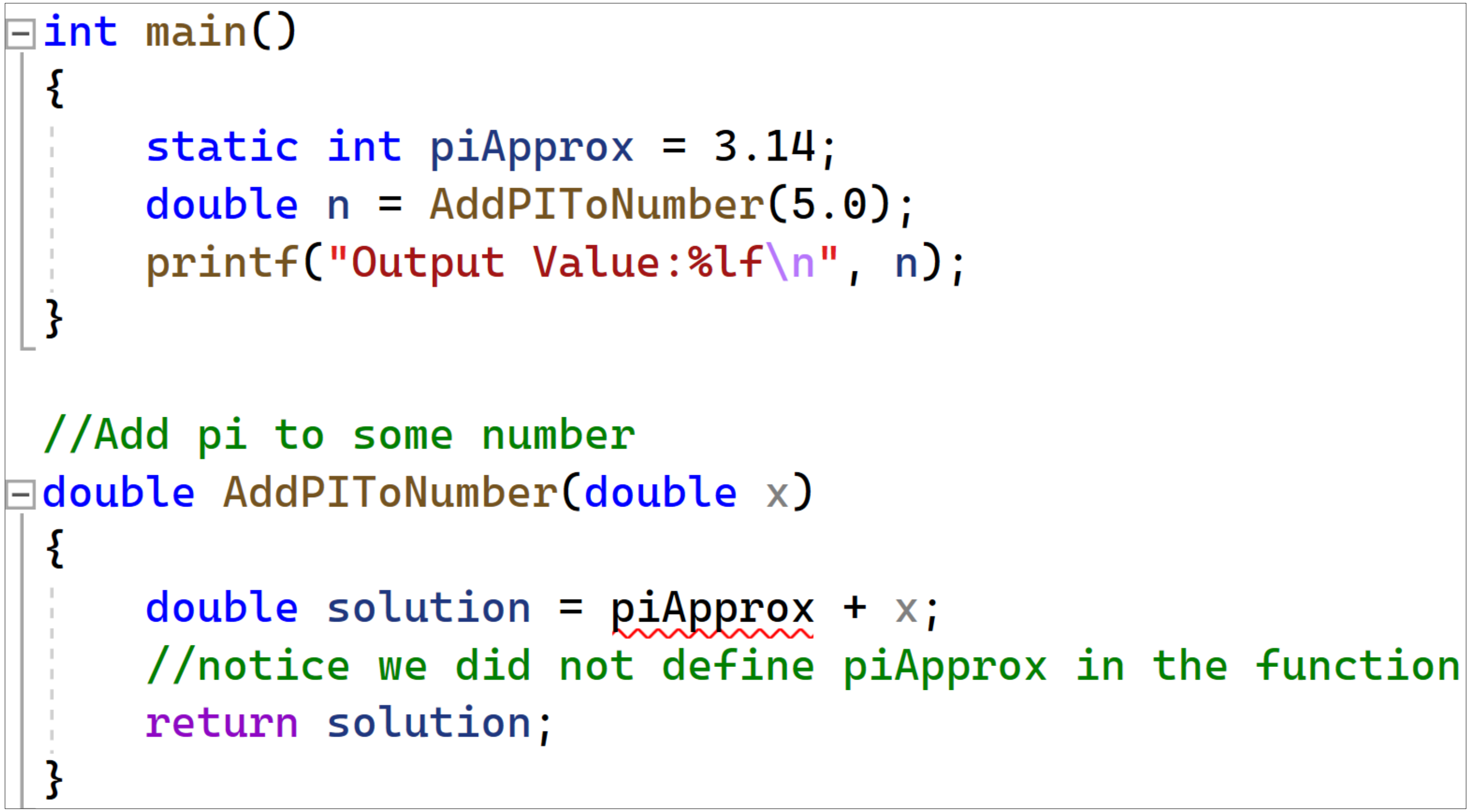


```c
#include <stdio.h>
#include <stdlib.h>

//here we define a static variable
static double piApprox = 3.14;

//Add pi to some number
double AddPIToNumber(double x)
{
    double solution = piApprox + x;
    //notice we did not define piApprox in the function
    return solution;
}

int main()
{
    double n = AddPIToNumber(5.0);
    printf("Output Value:%lf\n", n);
}
```



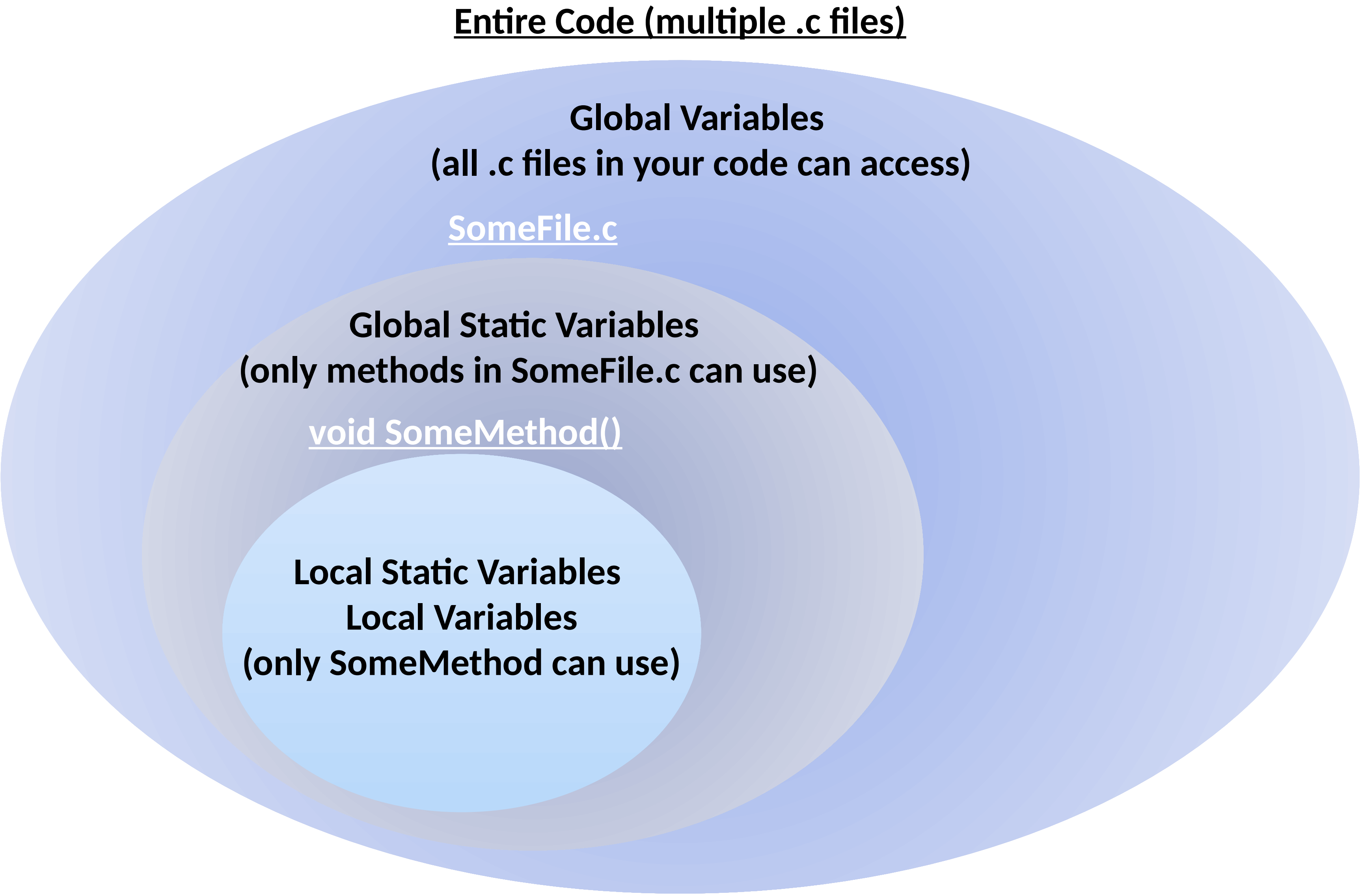
```
Output Value:8.140000
```

Test your knowledge: Is piApprox a global or local static variable here? Would this code run?



```c
int main()
{
    static int piApprox = 3.14;
    double n = AddPIToNumber(5.0);
    printf("Output Value:%lf\n", n);
}

//Add pi to some number
double AddPIToNumber(double x)
{
    double solution = piApprox + x;
    //notice we did not define piApprox in the function
    return solution;
}
```

Summary of the types of variables



Entire Code (multiple .c files)

Global Variables
(all .c files in your code can access)

SomeFile.c

Global Static Variables
(only methods in SomeFile.c can use)

void SomeMethod()

Local Static Variables
Local Variables
(only SomeMethod can use)

Be cautious with static and global variables

Declares all variables globally:
30 seconds later when errors occur:

- “Nice” functions **only** depend on their inputs
- Static and global variables have side-effects
  - Retain values across function calls
  - Change the meaning of the function at each call!
  - You cannot understand the function without holding **all** the code in your head.

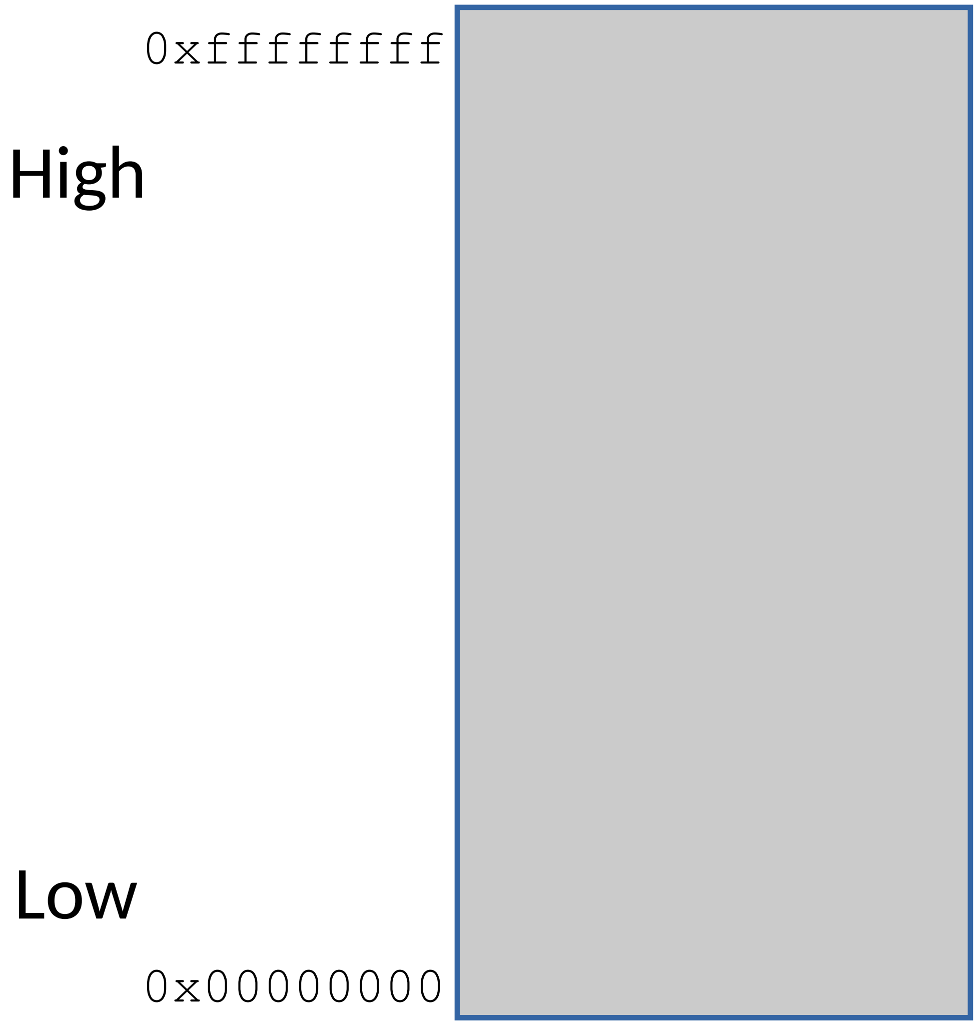
Function Call Context

- Function call context includes
  - Copies of function arguments (call-by-value)
  - (Auto) local variables
  - Return address, …
- Call contexts managed automatically using the **execution stack**
  - A stack frame is created automatically for each function call
  - The frame lasts for the duration of the call
  - Discarded automatically when the function terminates
  - NOTHING in the frame survives the call

3. Memory Organization

Memory Organization

- Memory….
  - Every *Process* has an
    - **Address Space**



0xffffffff

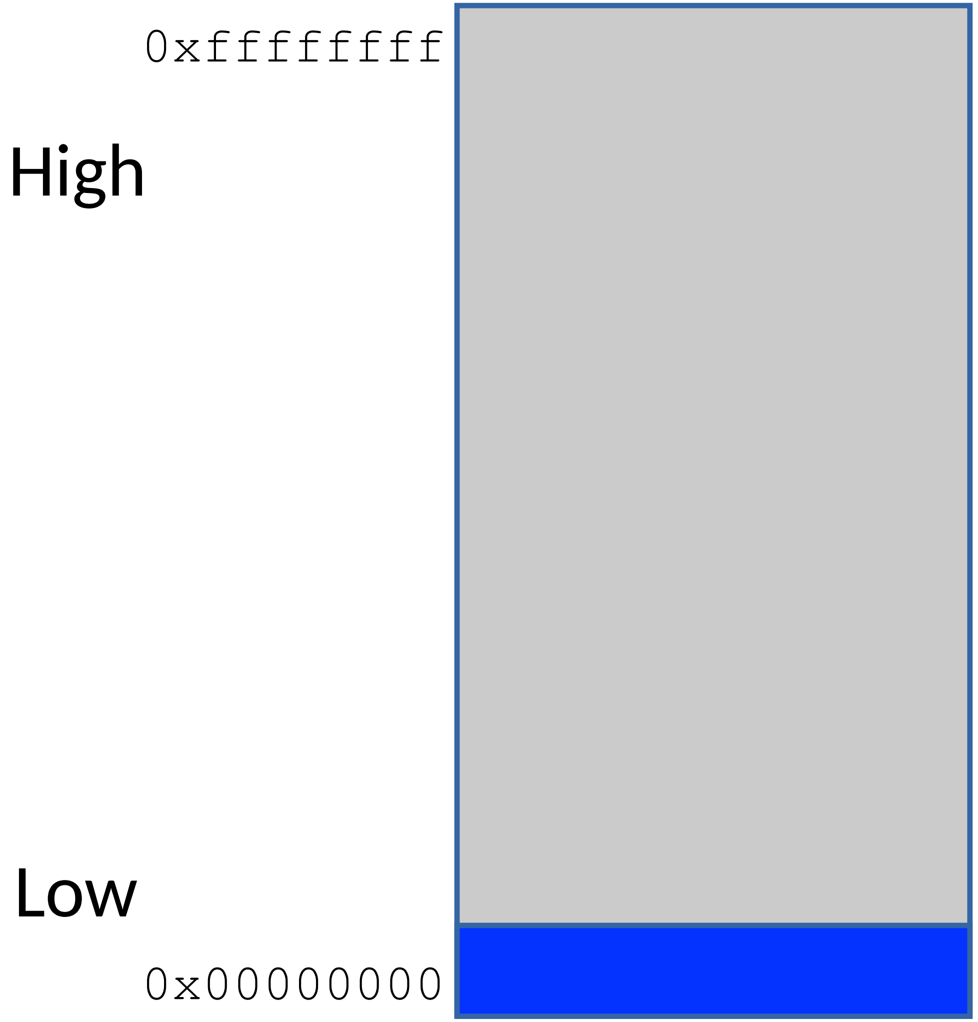
High

Low

0x00000000

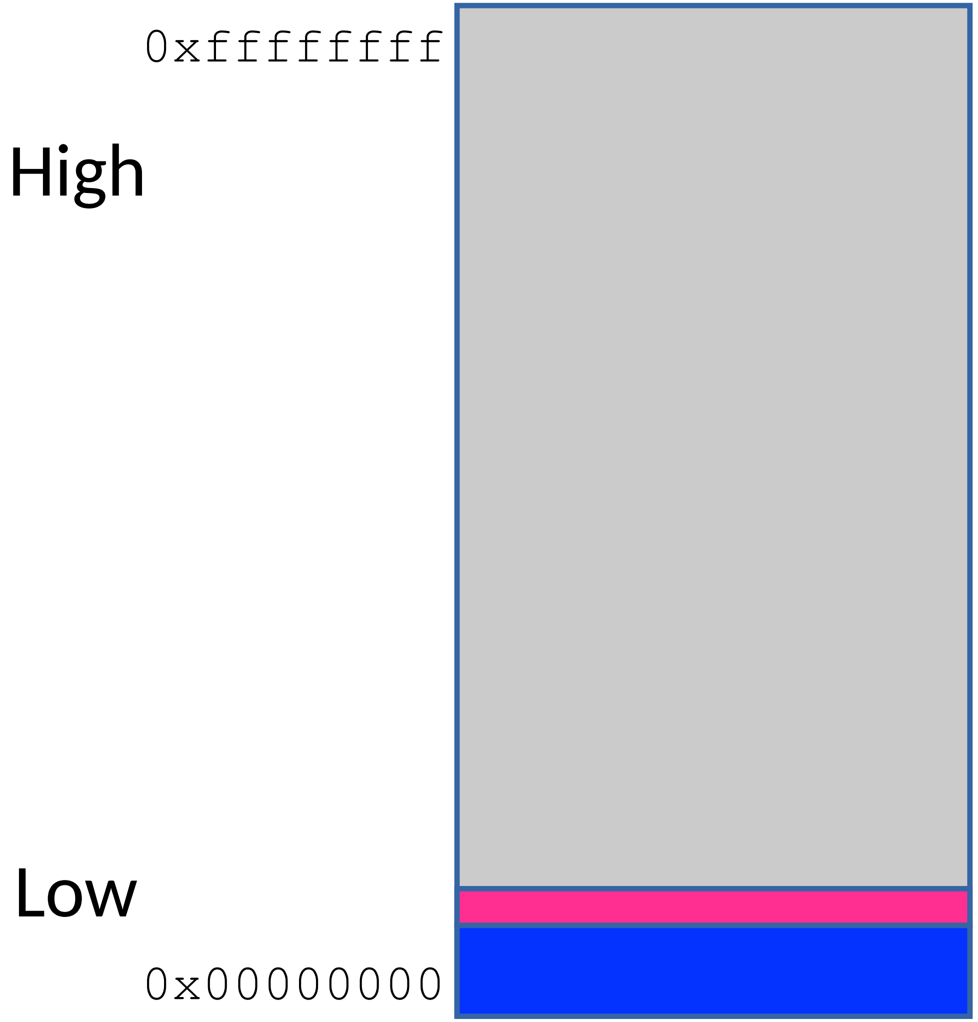
Memory Organization

- Memory….
  - Every *Process* has an
    - Address Space
  - **Executable** code is at the bottom
    - Not exactly from address 0



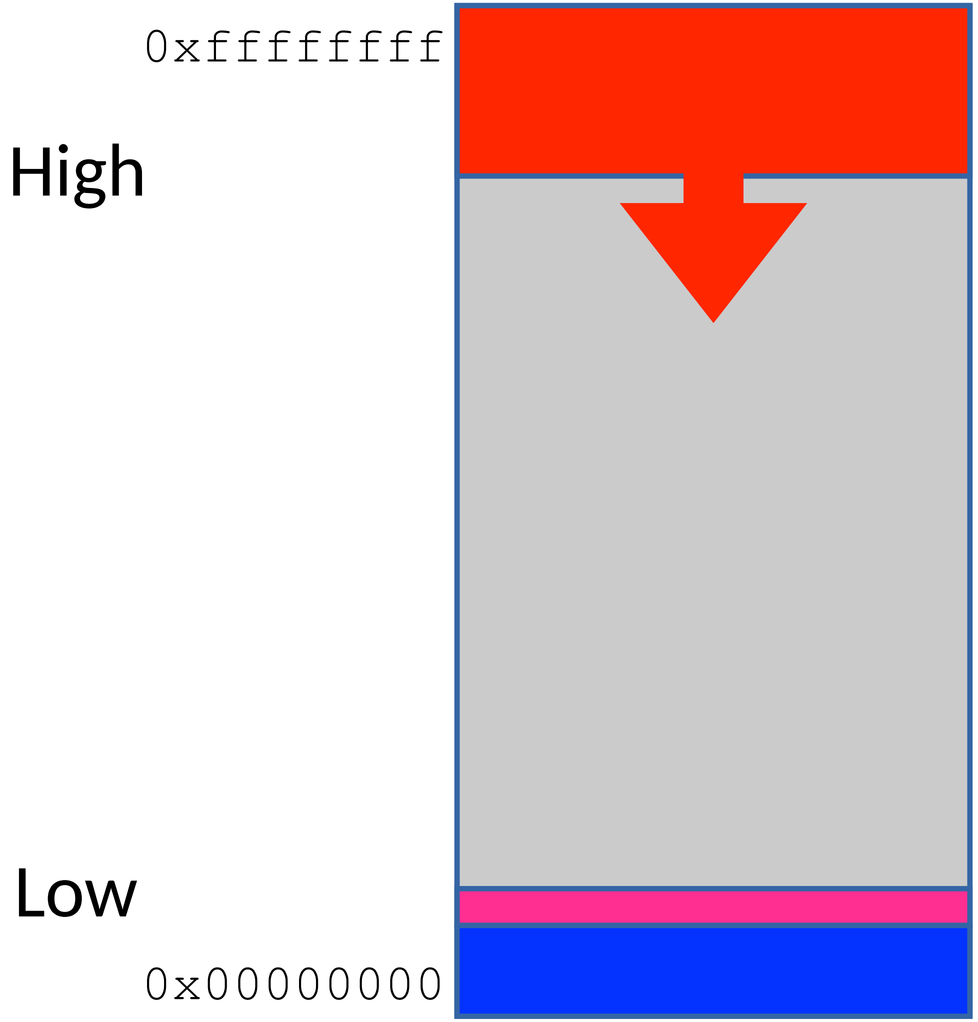
Memory Organization

- Memory….
  - Every *Process* has an
    - **Address Space**
  - **Executable** code is at the bottom
  - **Statics** and globals are just above



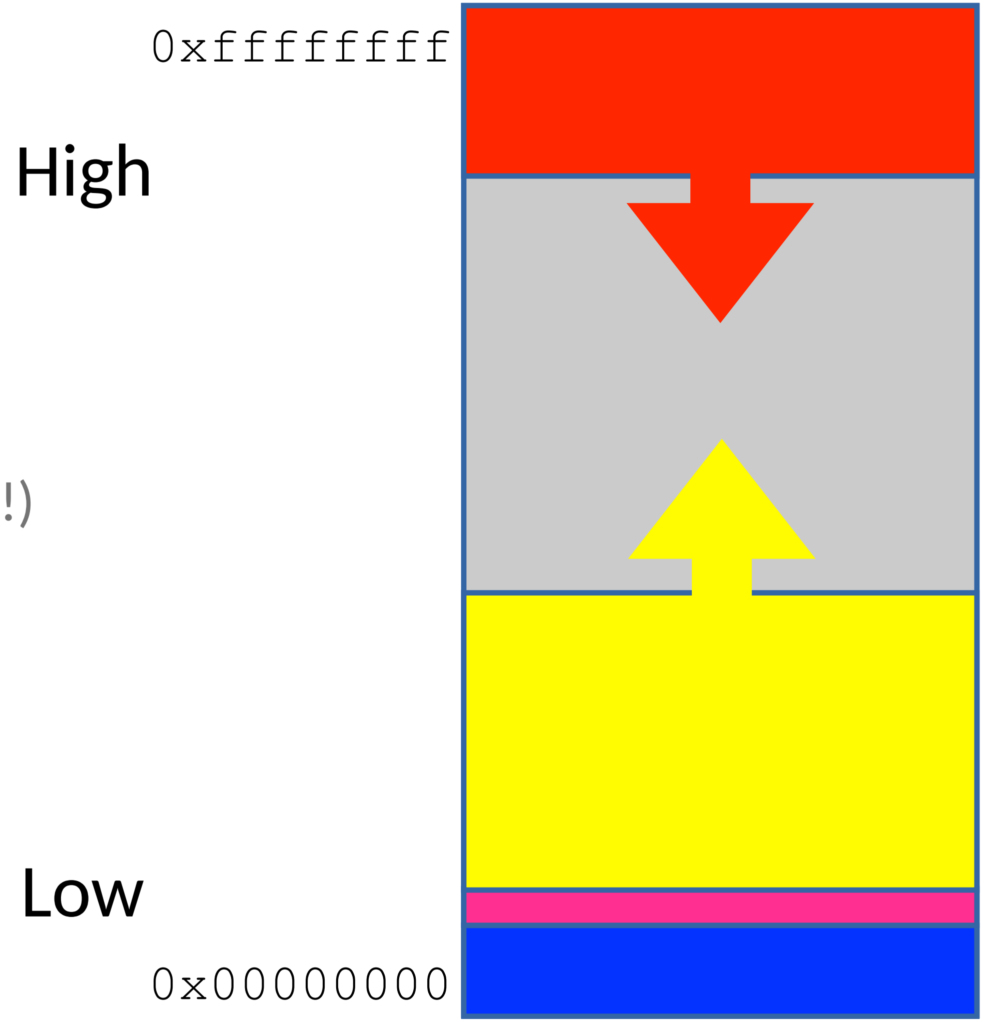
Memory Organization

- Memory….
    - Every *Process* has an
        - **Address Space**
    - **Executable** code is at the bottom
    - **Statics** and globals are just above
    - **Stack** is at the top (going down!)



Memory Organization

- Memory….
  - Every *Process* has an
    - **Address Space**
  - **Executable** code is at the bottom
  - **Statics** and globals are just above
  - **Stack** is at the top (going down!)
  - **Heap** grows from the bottom (going up!)

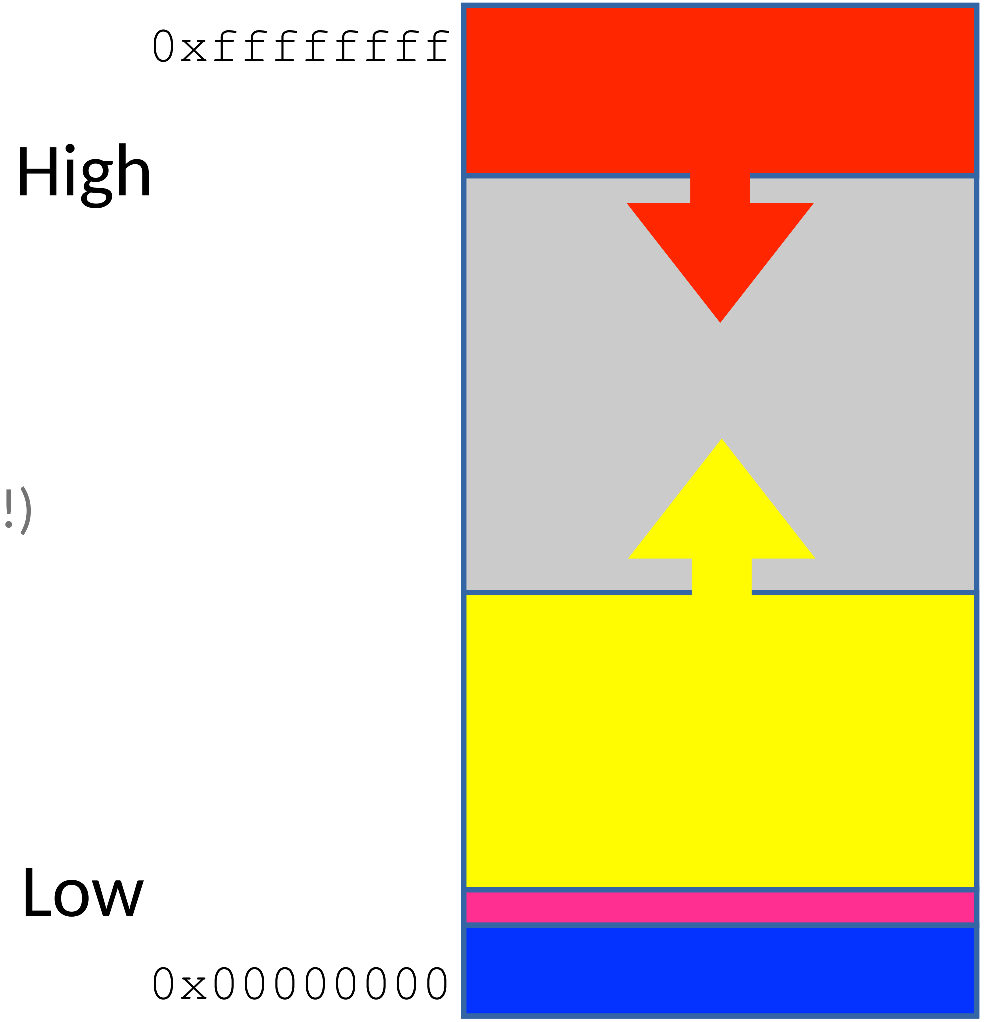


Memory Organization

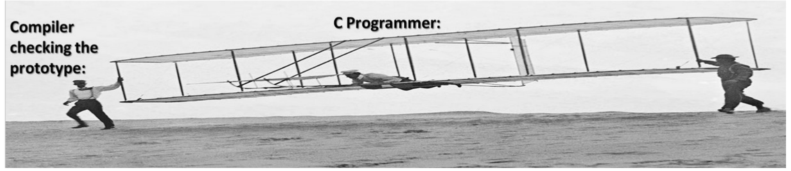
- Memory….
  - Every *Process* has an
    - **Address Space**
  - **Executable** code is at the bottom
  - **Statics** and globals are just above
  - **Stack** is at the top (going down!)
  - **Heap** grows from the bottom (going up!)
  - **Gray** no-man's land is up for grab

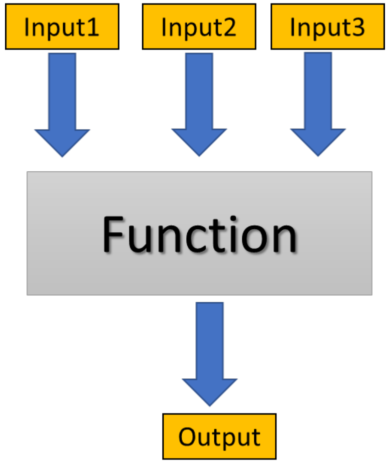
Low end may not start from 0.

High end may not be the 0xff…ff.



Function Lecture Conclusions




Input1

Input2

Input3

Function

Output

- Code should be split into different functions for readability and ease of use.
- C has global and local variables, as well as the static keyword.
- Use of global variables may be necessary from a design point of view, but should be used sparingly.

Figure Sources

1. https://i.pinimg.com/originals/2a/74/33/2a7433d23ce752166b7abc910299ff15.jpg
2. https://pbs.twimg.com/profile_images/3092028479/15db929f6739606a1aefc74277951af8_400x400.jpeg
3. https://media.makeameme.org/created/when-i-wrote-a49411.jpg
4. https://i.imgflip.com/17t6zh.jpg
5. https://img-9gag-fun.9cache.com/photo/am5j20j_460s.jpg
6. https://media.tenor.com/BiUtqfsTcqcAAAAC/memory-no-memory.gif