Functions (wrapping up…)

ESC101: Fundamentals of Computing

Nisheeth

Loose end: Flags

- Flags N[...]D – they are a programming style
- [...]gnal i[...]
- Can be used to avoid [...] break
- Can also be used to avoid using continue in loops
- Flags can be integer, long variables – usually 0/1
- You can give you[...] sensible name you[...]

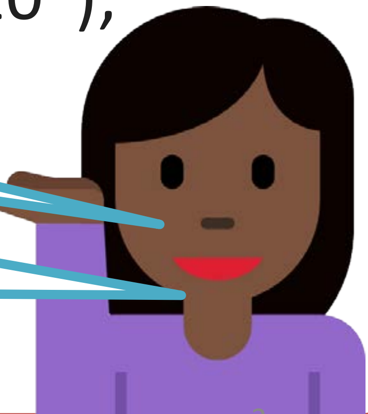
```
int num = 20;
int flag = 0; // Assume not div by 5
if(num %2 == 0){
    printf("Even");
    if(num % 5 == 0) flag = 1;
    if(flag)
    ...y 10");
}
```

CRITICAL: Always initialize your flags

If you don't initialize and num is not multiple of 5, flag may contain garbage value

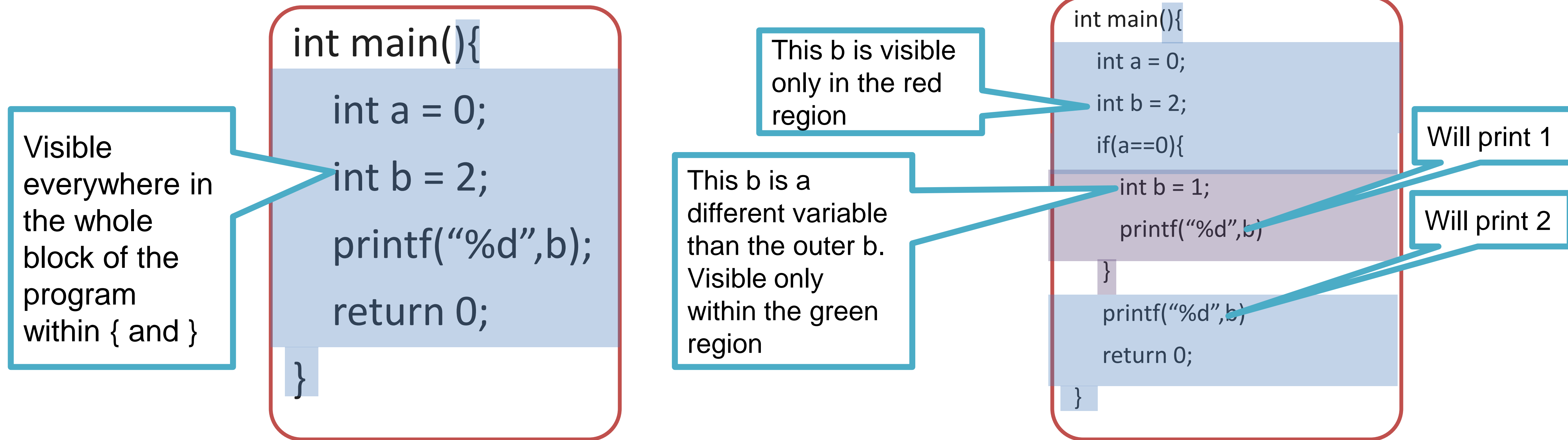
Could have also named this flag isDivBy5 – more descriptive name

Note: Could also avoid flag by using if(num%5==0){printf("Divisible by 10");} but using flag is a better practice in larger programs



Scope rules for variables

- Scope defines the regions in program where a variable is “visible”



- A pair of opening and closing curly braces creates a “block”
- Can re-declare a variable with same name if the name hasn’t been declared earlier in the same block. This variable will be visible until this block ends

Global Variables

Variable declared outside every function definition

Can be accessed by all functions in the program that follow the declaration

Also called *external* variable

What if a variable is declared inside a function that has the same name as a global variable?

The global variable is "shadowed" inside that particular function only.

```c
#include<stdio.h>
int g=10, h=20;

int add(){
  return g+h;
}

void fun1(){
  int g=200;
  printf("%d\n",g);
}

int main(){
  fun1();
  printf("%d %d %d\n",
          g, h, add());
  return 0;
}
```

```
200
10 20 30
```

1. The variable g and h have been defined as global variables.
2. The use of global variables is normally discouraged. Use local variables of functions as much as possible.
3. Global variables are useful for defining constants that are used by different functions in the program.

Constant Global Variables


```
const double PI = 3.14159;
double circum_of_circle(double r) {
  return 2 * PI * r; }
double area_of_circle (double r) {
  return PI * r * r;
}
```

defines PI to be of type double with value 3.14159. Qualified by const, which means that PI is a constant. The value inside the box associated with PI cannot be changed anywhere.

Static Variables

- We have seen two kinds of variables: local variables and global variables.
- There are static variables too.

```
int f () {
   static int ncalls = 0;
   ncalls  = ncalls + 1;
/* track the number of
times f() is called  */
   … body of f() …
}
```

- Use a local variable?
  - gets destroyed every time f returns
- Use a global variable?
  - other functions can change it! (dangerous)

GOAL: count number of calls to f()

SOLUTION: define ncalls as a static variable inside f().

It is created as an integer box the first time f() is called.

Once created, it never gets destroyed, and retains its value across invocations of f().

It is like a global variable, but visible only within f().

Its value persists across different calls to the function

Macros in C

Marcos are defined outside functions

Macros are handled by the C pre-processor (not compiler)

All marco statements begin with # (like #include)

Can use macros to also define constants using #define, e.g.,

#define PI 3.14159 (note that no "=" between name and value)

The macro maps an input sequence to an output sequence before the program has compiled (PI mapped to 3.14159 in the above example)

Macros in C

Object/constant-like macros

#define BUFFER_SIZE 1024

Pre-defined macros in C, e.g. __DATE__ , __TIME__ etc (note there are two underscores each before and after)

Function-like macros

#define min(X, Y) ((X) < (Y) ? (X) : (Y))

x = min(a,b) will be expanded to

x = ((a) < (b) ? (a) : (b))

Before compilation

Macros in C: Be Careful

Macro is a simple copy-paste

Without parentheses, can make operator precedence betray your code's logic

```
#define SQR(x) (x*x)

int main() {
  int a, b=3;
  a = SQR(b+5);
  printf("%d\n",a);
 return 0;
}
```

23

Recursion in function use

Process of solving a problem using solutions to "smaller" versions of the same problem!

You have already encountered recursion in mathematics

Factorial function is defined in terms of factorial itself!

$$\mathrm{fac}(0) = 1 \text{ and } \mathrm{fac}(n) = n \cdot \mathrm{fac}(n-1), \text{ for } n > 0$$

Proof by induction is basically a *recurs*

We used the proof for the case n-1 to prove the case n

Claim: 1 + 2 + 3 + … + n = n(n+1)/2

Proof: *Base case*: for n = 1 true by inspection

Inductive case: (1 + … + n) = (1 + … + n-1) + n = (n-1)n/2 + n = n(n+1)/2 ☺

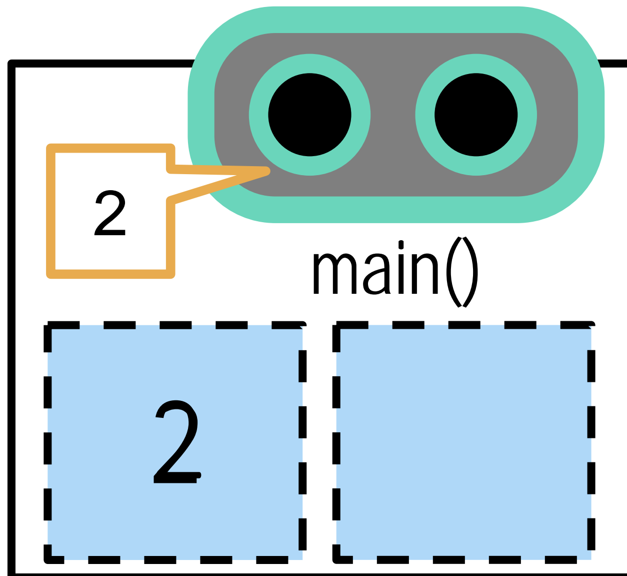
Notice that we need a base case and recursive case

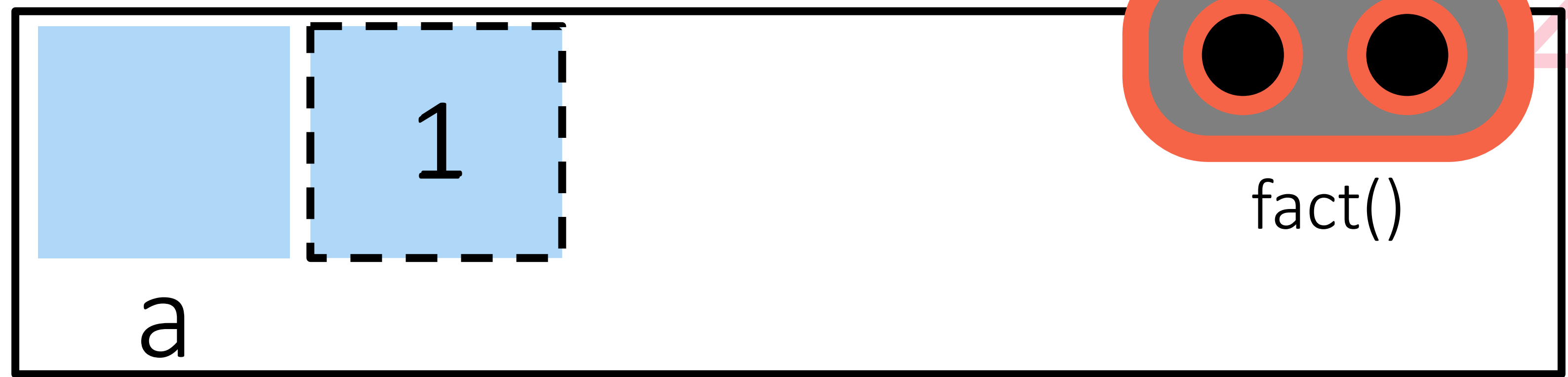
In case of factorial, fac(0) was the *base case*.

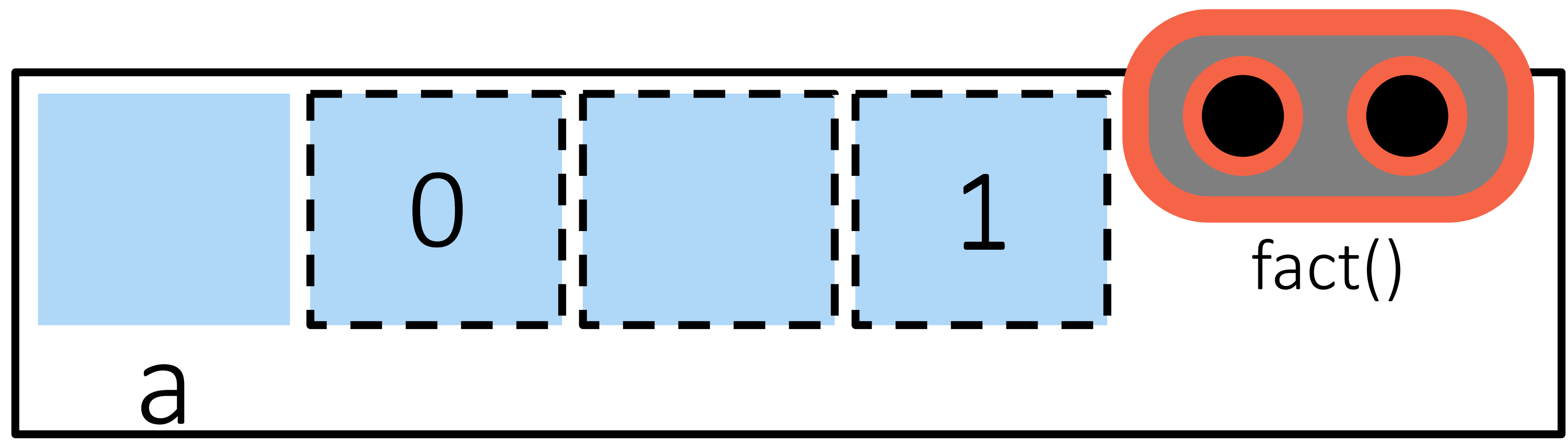
This is true when writing recursive functions in C language as well

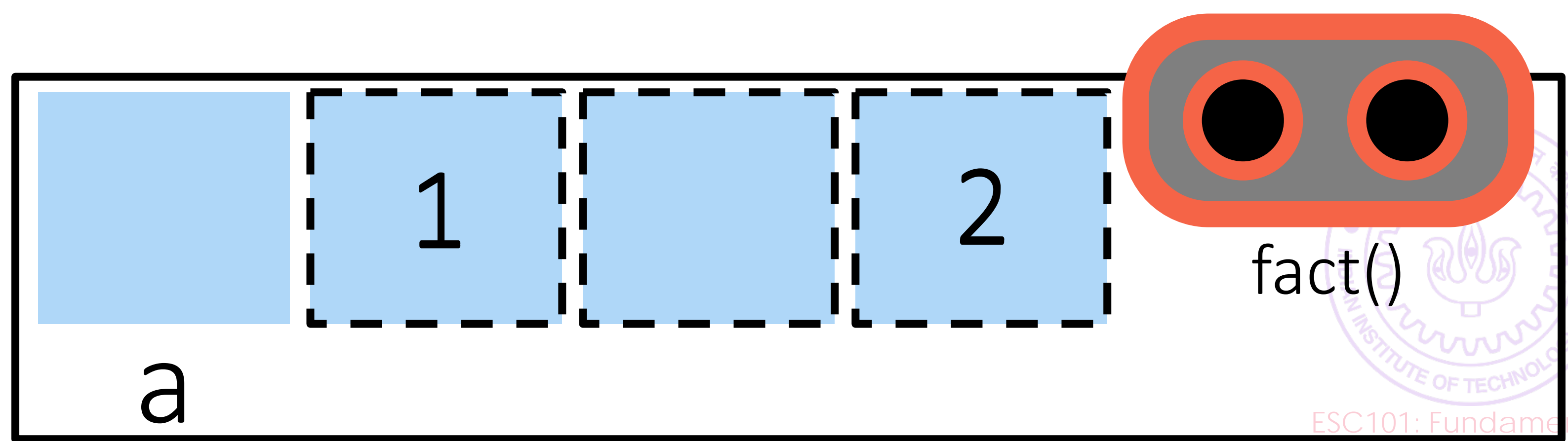
Factorial

```
int fact(int a){
    if(a == 0) return 1;
    return a * fact(a - 1);
}
int main(){
    printf("%d", fact(1+1));
}
```









Recap – 6 basic rules of C functions

RULE 1: When we give a variable as input, the value stored inside that variable gets passed as an argument

RULE 2: When we give an expression as input, the value generated by that expression gets passed as argument

RULE 3: In case of a mismatch b/w type of arg promised and type of arg passed, typecasting will be attempted

RULE 4: All values passed to a function get stored in a fresh variable inside that function (changes made to this variable won't change the original var regardless of whether it is a normal var or pointer)

RULE 5: Value returned by a function can be used freely in any way values of that data-type could have been used

RULE 6: All clones share the memory address space

Take home question

What will the output of this code?

```
#include<stdio.h>
int recursive(int i) {
        static int count = 0;
        count = count + i;
        return count;
}

int main() {
        int i, j;
        for (i = 0; i <= 5; i++)
                j = recursive(i);
        printf("%d\n", j);
        return 0;
}
```