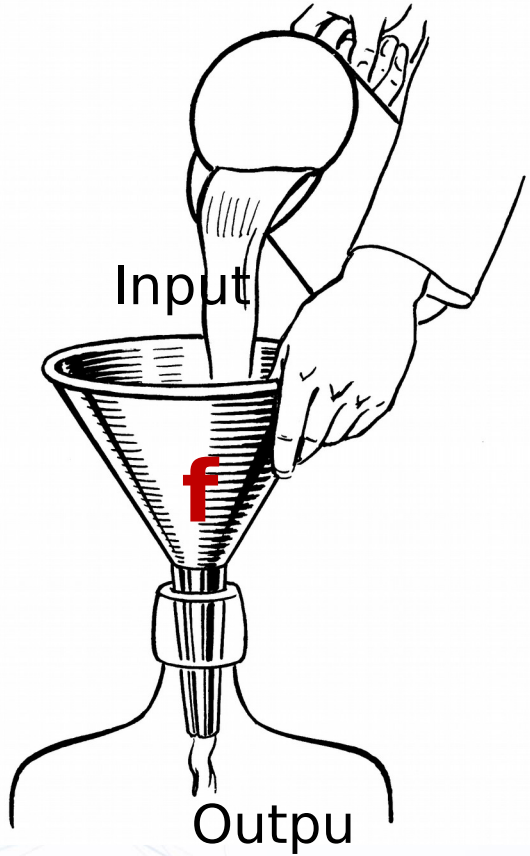
# Function

- An independent, self-contained entity of a C program that performs a well-defined task
- It has
  - Name: for identification [Hall 4 canteen]
  - Arguments: to pass information from outside world (rest of the program) [food orders]
  - Body: processes the arguments  do something useful [cooks the ordered food for you]
  - Return value: To communicate back to outside world [serves you the delicious food]
    - Sometimes not required

# Parts of a Function



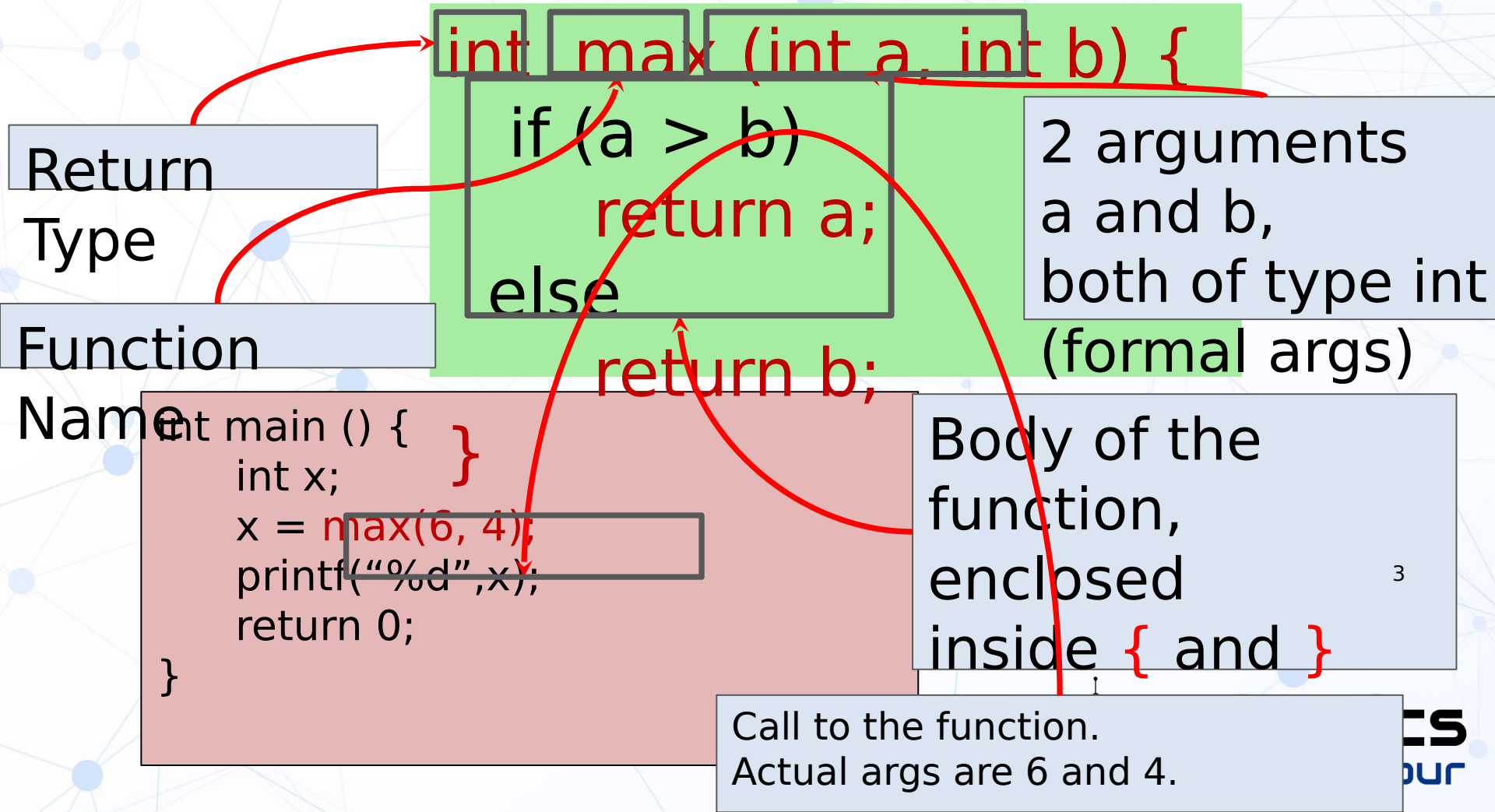

Similar to math functions

$$\sin(x)$$

$$f(x_1, x_2, \ldots, x_n)$$



Robotics Club IITKanpur


int max (int a, int b) {
if (a > b)
return a;
else
return b;
}
Return Type
Function Name
2 arguments a and b, both of type int (formal args)
int main () {
int x;
x = max(6, 4);
printf("%d",x);
return 0;
}
Body of the function, enclosed inside { and }
Call to the function.
Actual args are 6 and 4.
3

# Functional Terminology

**Function Name**: must be a valid identifier abc, a124, _ab1

**Arguments**: can be int, long, float, double, char

Can also have more structures - will discuss when we learn them

**Return type**: what does the function *return*

When you use a function, we say you have *called* that function. If the function outputs something, we say the function *returned* that output back to you

The English word return has two meanings:
*I returned from th*
*I returned the two*

Functions return back values to you just

as you return books

You must define the function **before** using the function (within main or your own functions)

# Arguments

- Input to the function
  - Should have matching type
  - Type should be declared

- A new copy of these arguments is made
  - Function works on these new copies

# Why use functions?

- Break up complex problem into small sub-problems
- Solve each of the sub-problems separately as a function, and combine them together in another function
- The main tool in C for modular programming

# Advantages of Using Functions

- **Code Reuse**: Allows us to reuse a piece of code as many times as we want, without having to write it
  - Think of the printf function!
- **Procedural Abstraction**: Different pieces of your algorithm can be implemented using different functions
- **Distribution of Tasks**: A large project can be broken into components and distributed to multiple people
- **Easier to debug**: If your task is divided into smaller subtasks, it is easier to find errors
- **Easier to understand**: Code is better organized and hence easier for an outsider to understand it

# We Have Seen Functions Before

- main() is a special function. Execution of program starts from the beginning of main().
- scanf(...), printf(...) are standard input-output library functions.

# Function Call

- A function call is an *expression*
  - feeds the necessary values to the function arguments
  - directs a function to perform its task
  - receives the return value of the function
- Similar to operator application

5 + 3 is an expression of type integer that evaluates to 8

max(5, 3) is an expression of type integer that evaluates to 5

# Returning from a function: Type

- Return type of a function tells the type of the result of function call
- Any valid C type
  - int, char, float, double, …
  - **void**
- Return type is void if the function is not supposed to return

```
void print_one_int(int n) {
    printf("%d", n);
}
```

# Function Declaration- **Prototype**

- A function declaration is a statement that tells the compiler about the different properties of that function
  - name, argument types and return type of the function
- Structure:

  return_type function_name (list_of_args);

- Looks very similar to the first line of a function definition, but NOT the same
  - has semicolon at the end instead of BODY

# Nested Function Calls

- Functions can call each other
- A declaration or definition (or both) must be visible before the call
  - Help compiler detect any inconsistencies in function use
  - Compiler warning, if both (decl & def) are missing

```
#include<stdio.h>
int min(int, int); //declaration
int max(int, int); //of max, min

int max(int a, int b) {
   return (a > b) ? a : b;
}
// a “cryptic” min, uses max
int min(int a, int b) {
   return a + b – max (a, b);
}

int main() {
  printf(“%d”, min(6, 4));
}
```

# Scope vs Storage Duration

- *Scope* determines where a name can be accessed
  - *Local* variables
  - *Global* variables
- *Storage duration* determines when a variable is created and destroyed
  - *Automatic* variables
  - *Static* variables

# Scope of a Name

- Two variables can have the same name only if they are declared in separate scopes
- A variable can not be used outside its scope
- C program has
  - function/**block** scope
  - **file** scope
  - **global**/external scope

# Local Variable

- E.g., declared within a function / conditional statement / loop
- The assigned value is lost when the function is exited
- Could also be declared in a single compound statement

# Global Variable

- 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

# Global Variables

```
#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.

# 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().**

**Static variables are not allocated on stack. So they are not destroyed when f() returns.**

# Summary

- Local Variable
  - Visible in scope
  - Lives in scope (destroyed at the point where we leave the scope)
- Global Variable
  - Visible everywhere
  - Lives everywhere (never destroyed)
- Static Variable
  - Visible in Scope
  - Lives everywhere! (but can not be accessed outside scope)