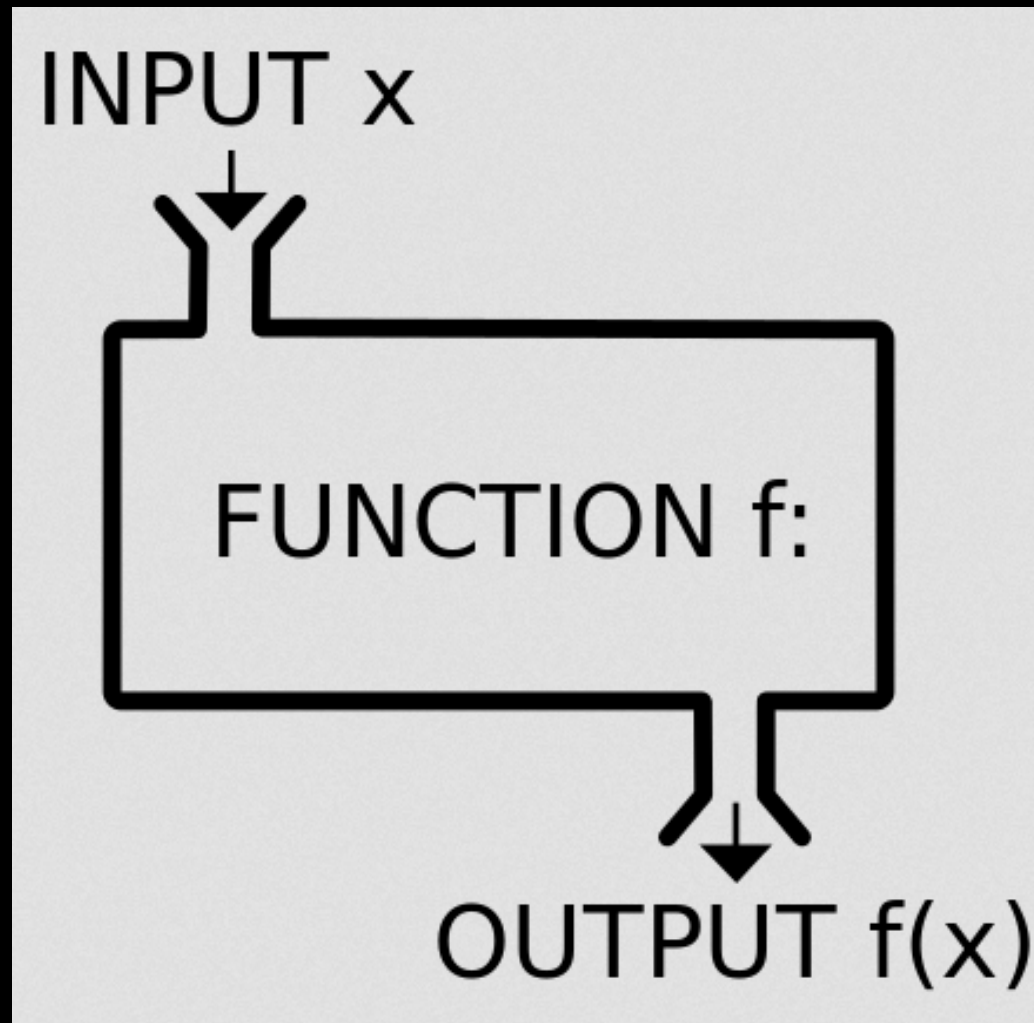


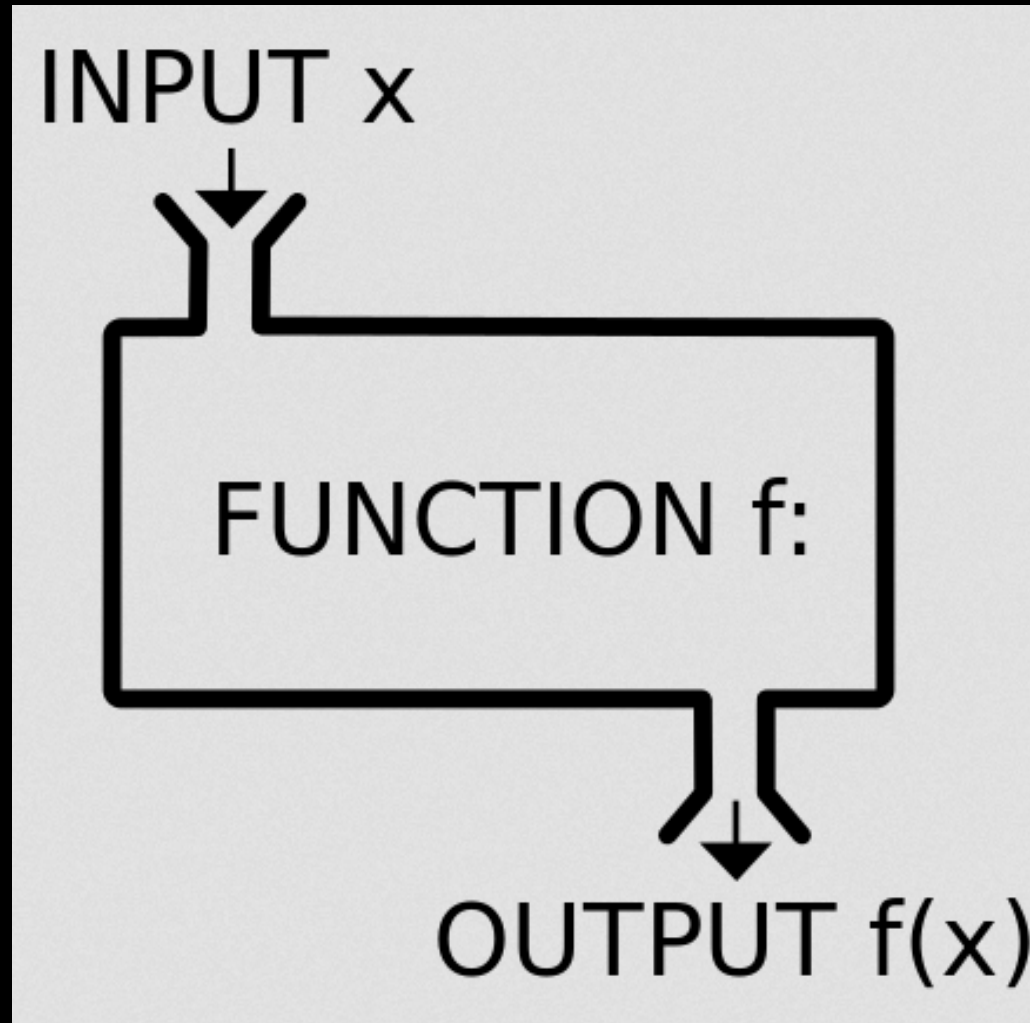
# CSE102

## Computer Programming

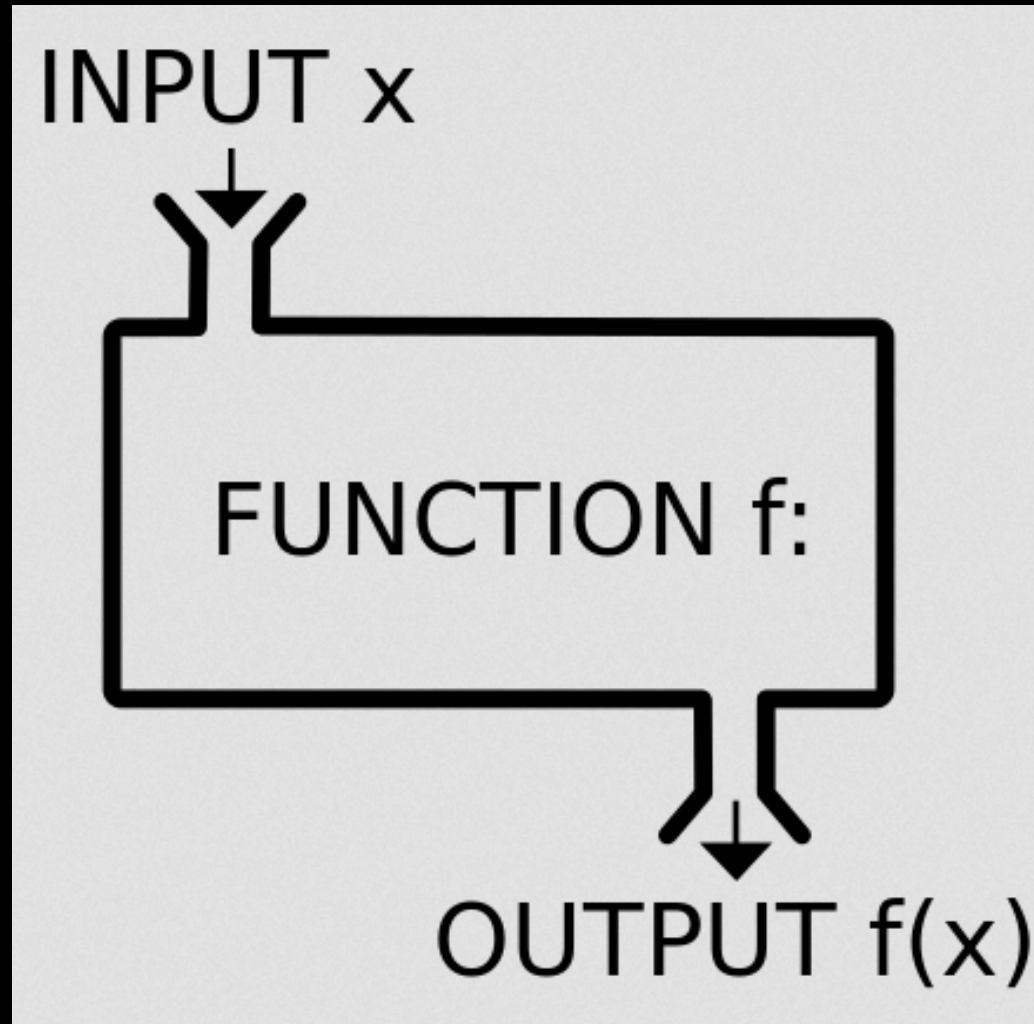


# What are Functions?

Also called methods, procedures etc.

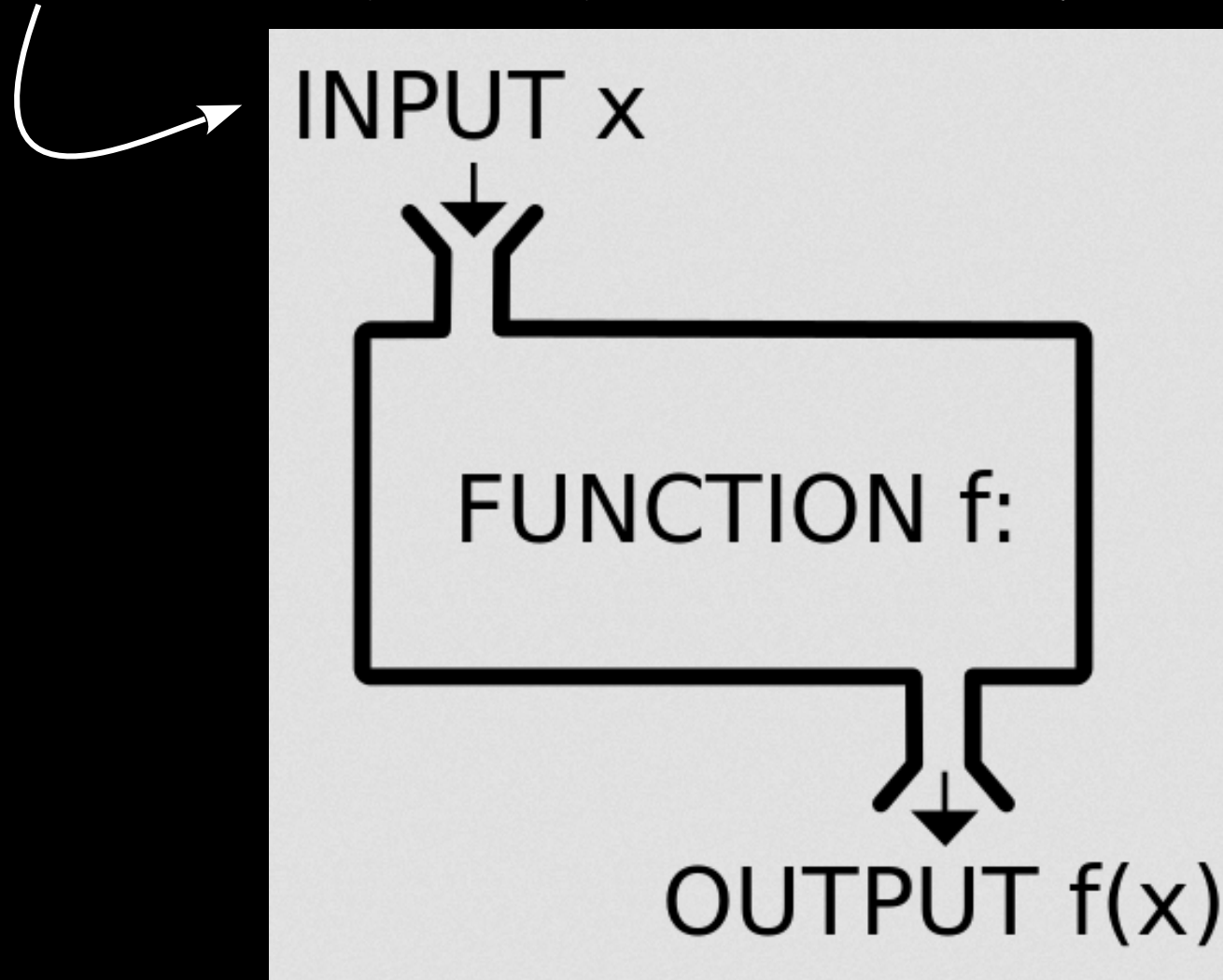


# What Should They Have?



# What Should They Have?

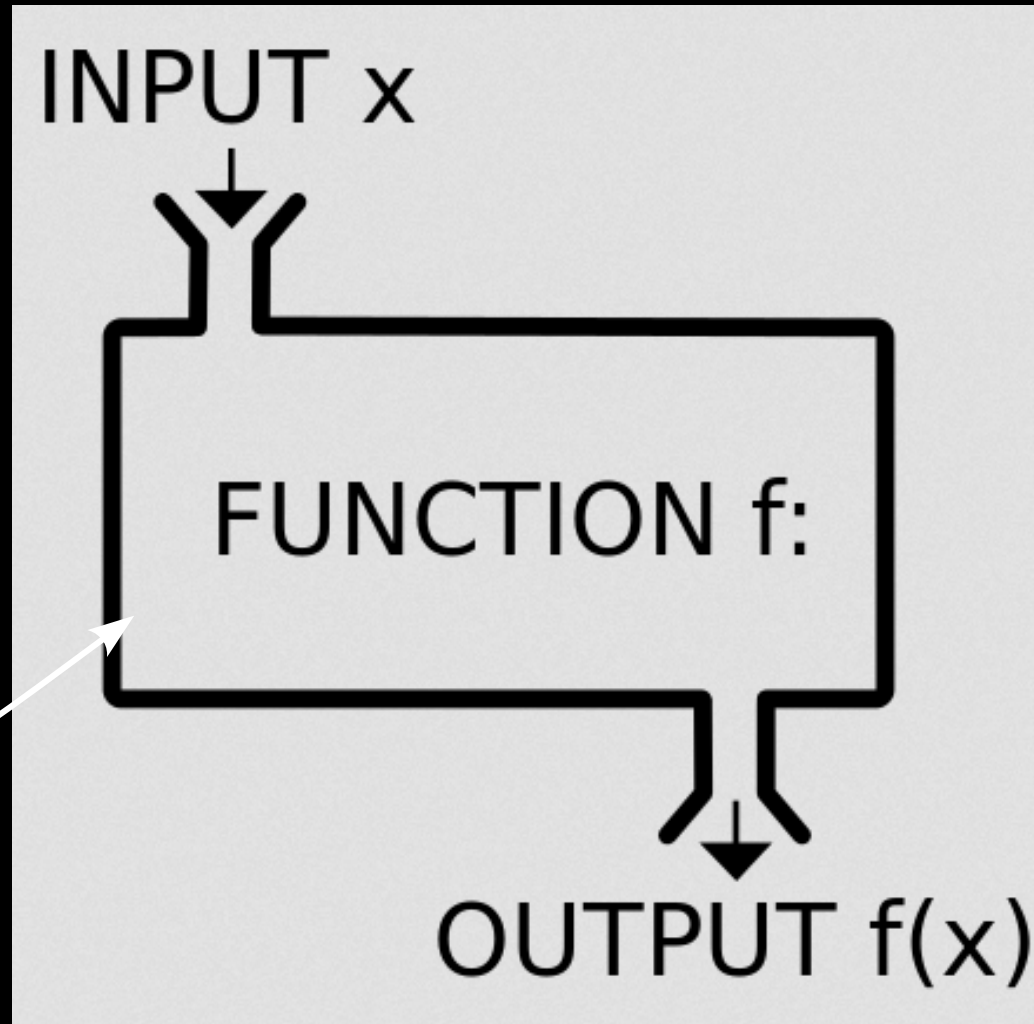
should accept inputs (really huh!?)



# What Should They Have?

should accept inputs!

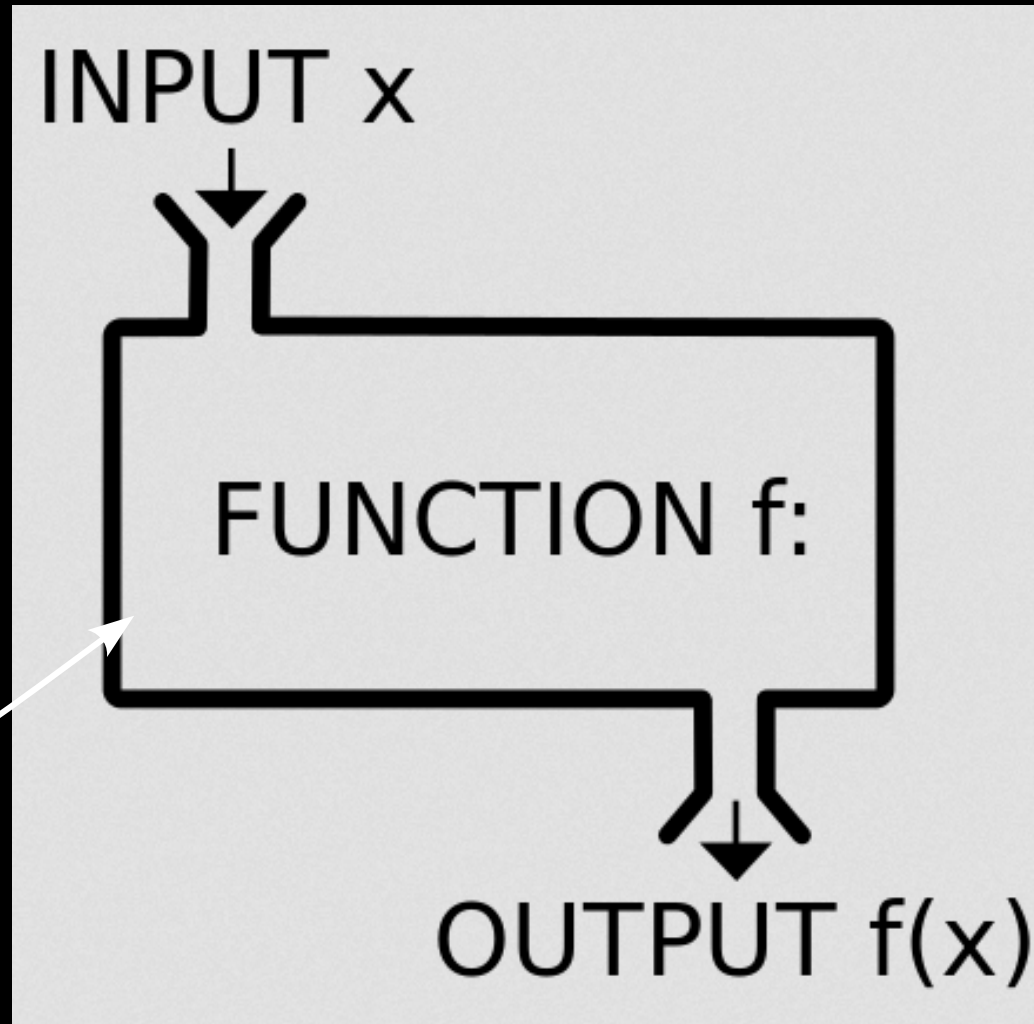
should  
manipulate  
inputs



# What Should They Have?

should accept inputs!

should  
manipulate  
inputs

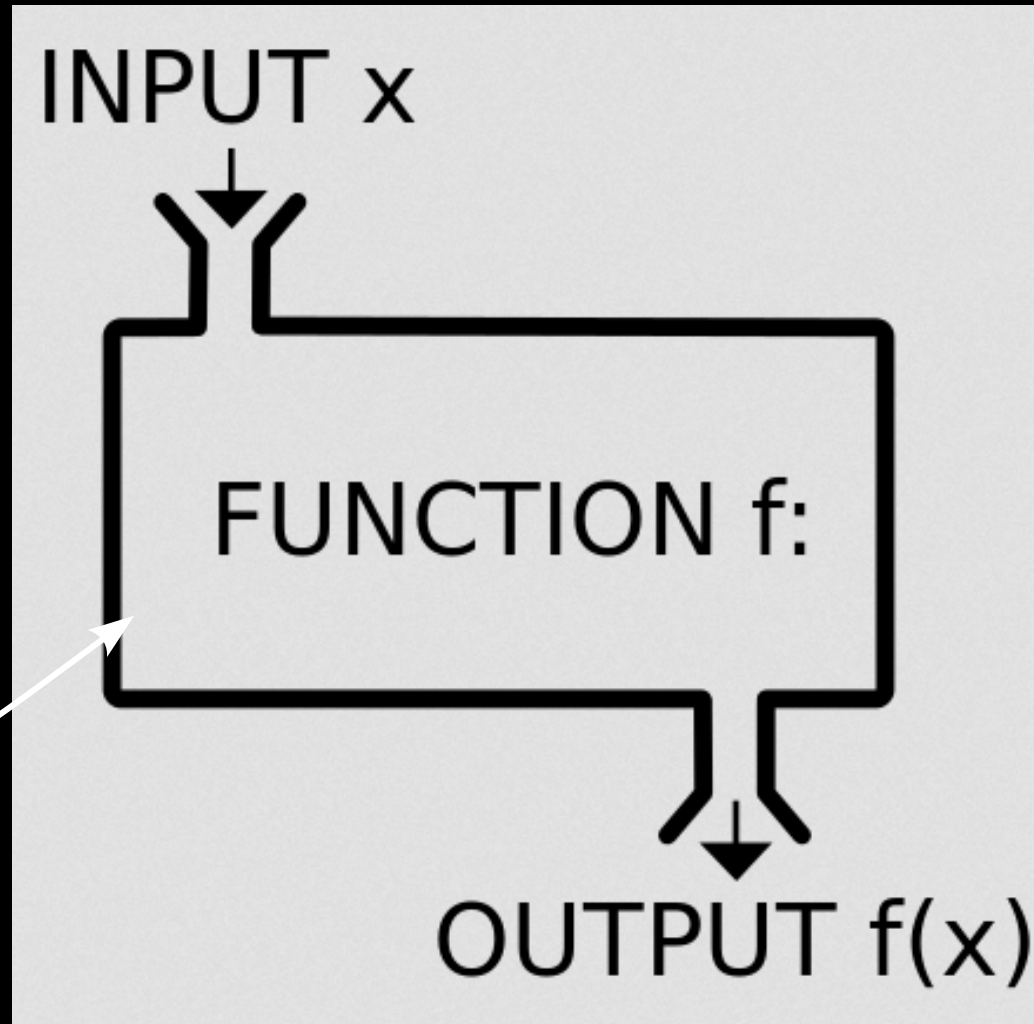


should  
return  
outputs  
(really!)

# What Should They Have?

should accept inputs!

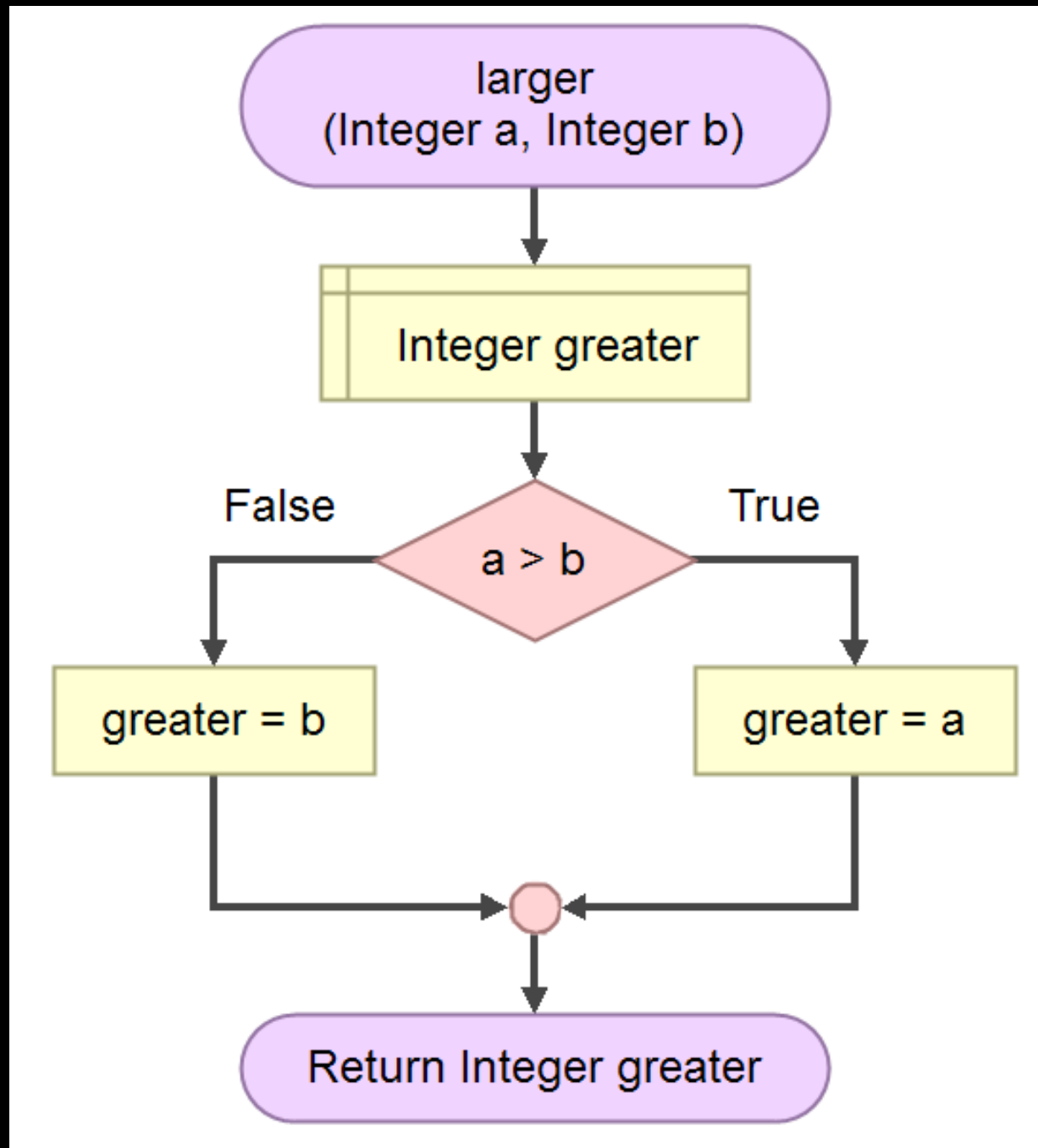
should  
manipulate  
inputs



should  
return  
outputs

should have a name!!


# How Do They Look Like?





# How Do They Look Like?

Returns an int value

 `int larger(int a, int b)`


{

    if (a > b)

        return a;

    return b;

}

 This function takes two arguments:  
a and b. Both arguments are ints.

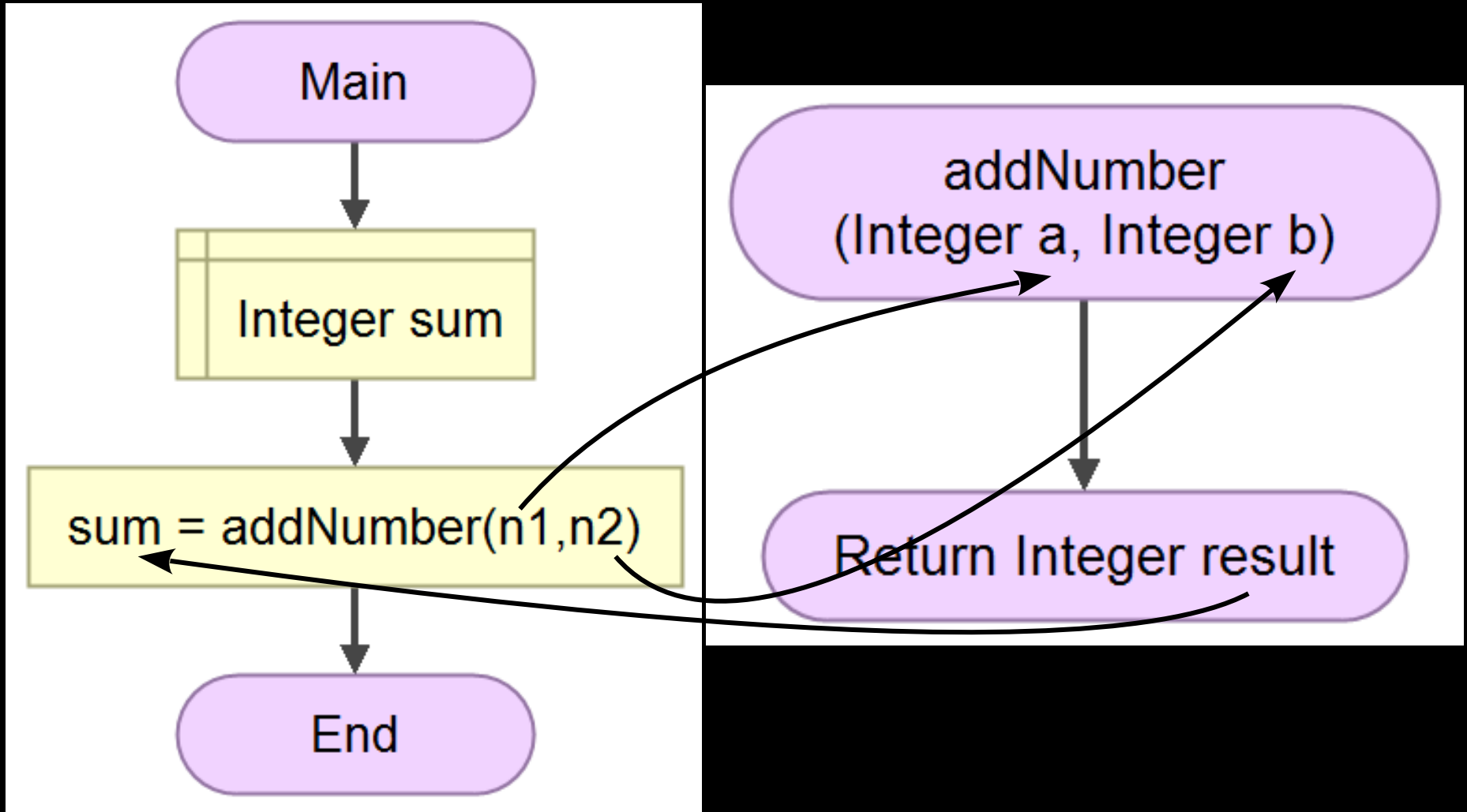
# How Do They Look Like?

```
Returns an int value
    ↪ int larger(int a, int b)
        should have a name
        {
            should if (a > b)
            manipulate return a;
            inputs
            return b; should return outputs
        }
```

should accept inputs

This function takes two arguments:  
a and b. Both arguments are ints.

# How are They Used?



# How are They Used?

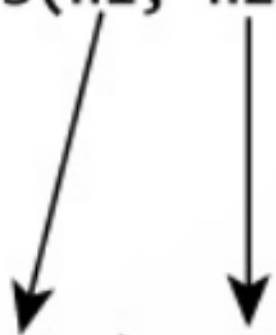
```
#include <stdio.h>

int addNumbers(int a, int b);

int main()
{
    ... ..

    sum = addNumbers(n1, n2);
    ... ..
}

int addNumbers(int a, int b)
{
    ... ..
    ... ..
}
```



The diagram consists of two arrows. The first arrow originates from the parameter 'n1' in the function call 'addNumbers(n1, n2);' within the 'main' function and points down to the parameter 'a' in the function definition 'int addNumbers(int a, int b)'. The second arrow originates from the parameter 'n2' in the same function call and points down to the parameter 'b' in the function definition. This illustrates how arguments are passed from the caller to the callee.

# How are They Used?

Function prototyping  
declares all info.  
about function  
(name, no. of args,  
return type) to  
the compiler  
like  
variable declaration

```
#include <stdio.h>
Function Prototyping
int addNumbers(int a, int b);

int main()
{
    ... ..
    sum = addNumbers(n1, n2);
    ... ..
}

int addNumbers(int a, int b)
{
    ... ..
    ... ..
}
```

parameters

Arguments

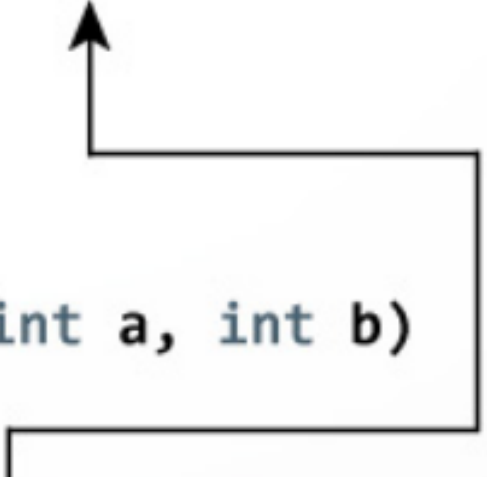
# How are They Used?

```
#include <stdio.h>

int addNumbers(int a, int b);

int main()
{
    ... ..
    sum = addNumbers(n1, n2);
    ... ..
}

int addNumbers(int a, int b)
{
    ... ..
    return result;
}
```



The diagram illustrates the flow of data between the `main` function and the `addNumbers` function. A line originates from the `return result;` statement in the `addNumbers` function definition, extends horizontally to the right, then vertically upwards, and finally horizontally to the left, ending with an arrow pointing to the `sum = addNumbers(n1, n2);` statement in the `main` function. This visualizes how the value returned by the function is passed back to the caller.

sum = result

# Our larger Function

Returns an int value

```
int larger(int a, int b)
```

```
{
```

```
    if (a > b)
```

```
        return a;
```

```
    return b;
```

```
}
```

This function takes two arguments:  
a and b. Both arguments are ints.

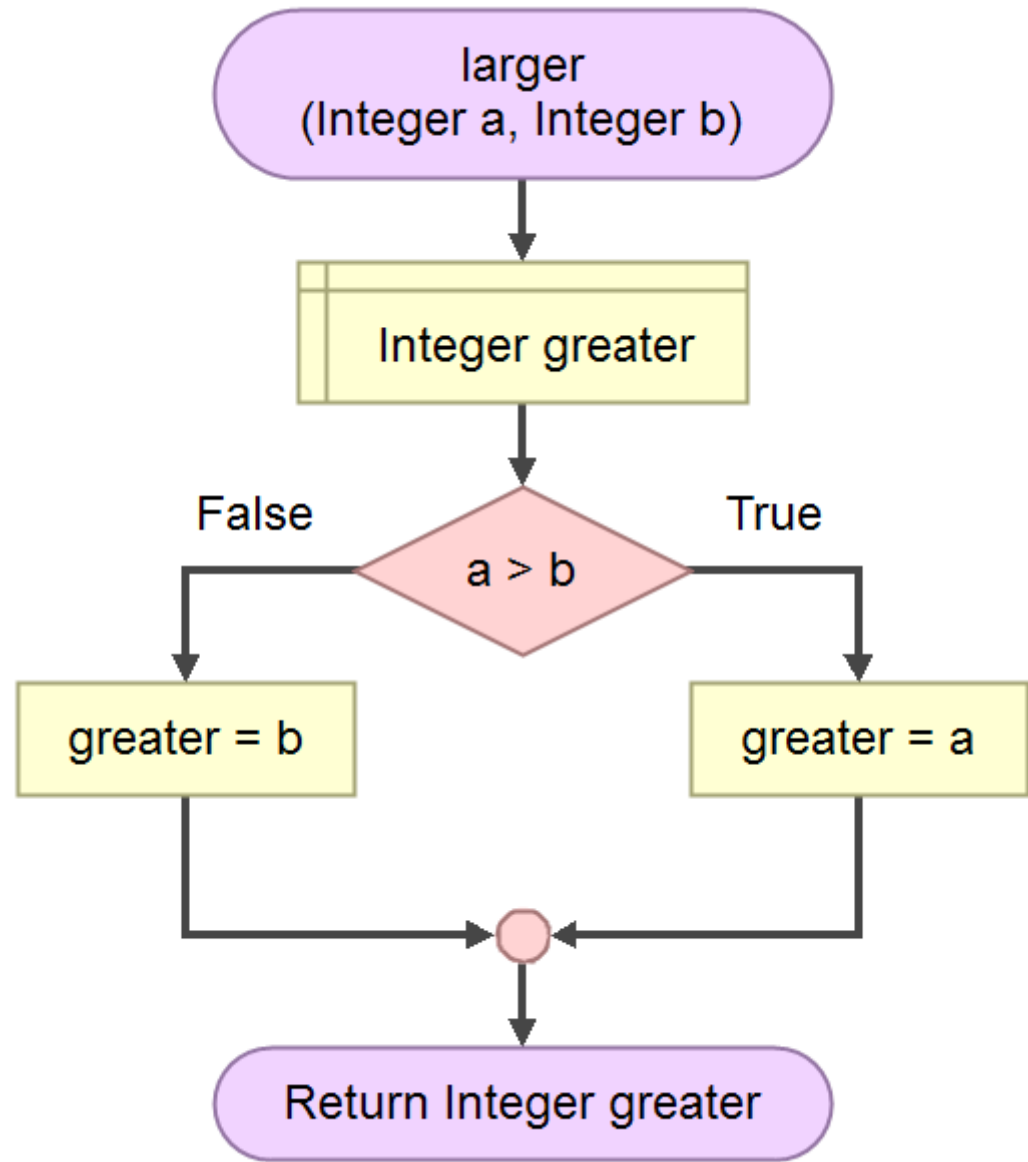
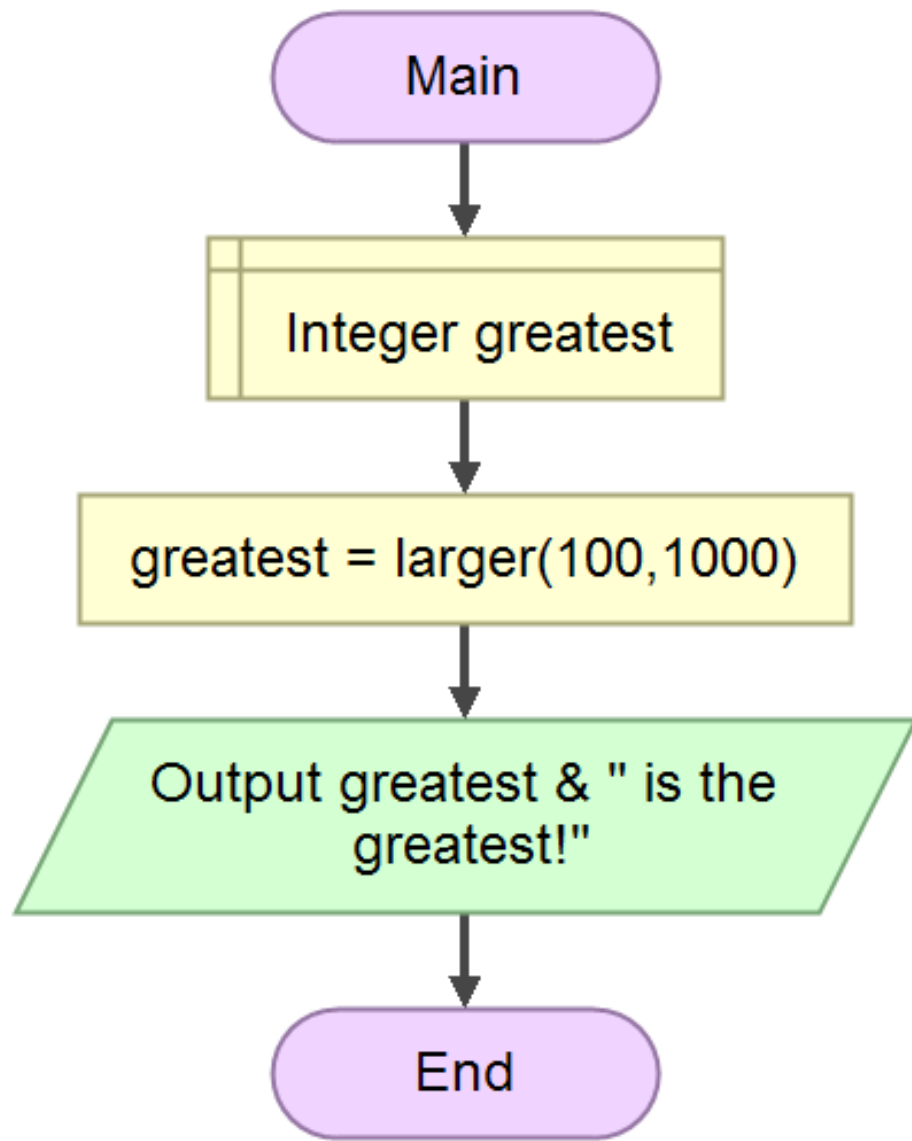
# Is Called Here!!

Assuming that function prototyping is done already

```
int main()  
{  
    Calling the function here  
    ↓  
    int greatest = larger(100, 1000);  
    printf("%i is the greatest!\n", greatest);  
    return 0;  
}
```



# Function Call



# How Functions Work?

equivalent  
to

having  
this code  
here

Functions are named  
piece of code

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

```
void functionName()  
{
```

```
... ..  
... ..
```

```
}
```

```
int main()  
{
```

```
... ..  
... ..
```

```
functionName();
```

```
... ..  
... ..
```

```
}
```

# What's void in Function?

```
#include <stdio.h>

void functionName()
{
    ... ..
    ... ..
}

int main()
{
    ... ..
    ... ..
    functionName();
    ... ..
    ... ..
}
```

The diagram illustrates the execution flow between two functions. A circle highlights the `void` keyword in the `functionName()` definition. An arrow originates from the `functionName();` call inside the `main()` function and points to the opening curly brace of the `functionName()` definition. Another arrow originates from the closing curly brace of the `functionName()` definition and points to the line of code immediately following the `functionName();` call in the `main()` function, indicating the return path.

# What's void in Function?

The void return type means the function won't return anything.

➔ `void complain()`

{

`puts("I'm really not happy");`

}



There's no need for a return statement because it's a void function.

# there are no Dumb Questions

**Q:** If I create a `void` function, does that mean it can't contain a `return` statement?

**A:** You can still include a `return` statement, but the compiler will most likely generate a warning. Also, there's no point to including a `return` statement in a `void` function.

**Q:** Really? Why not?

**A:** Because if you try to read the value of your `void` function, the compiler will refuse to compile your code.

# Types of Functions

Standard library functions:

- built-in functions

- often defined in header files

- available if header files are included

User-defined functions:

- custom created based on requirements

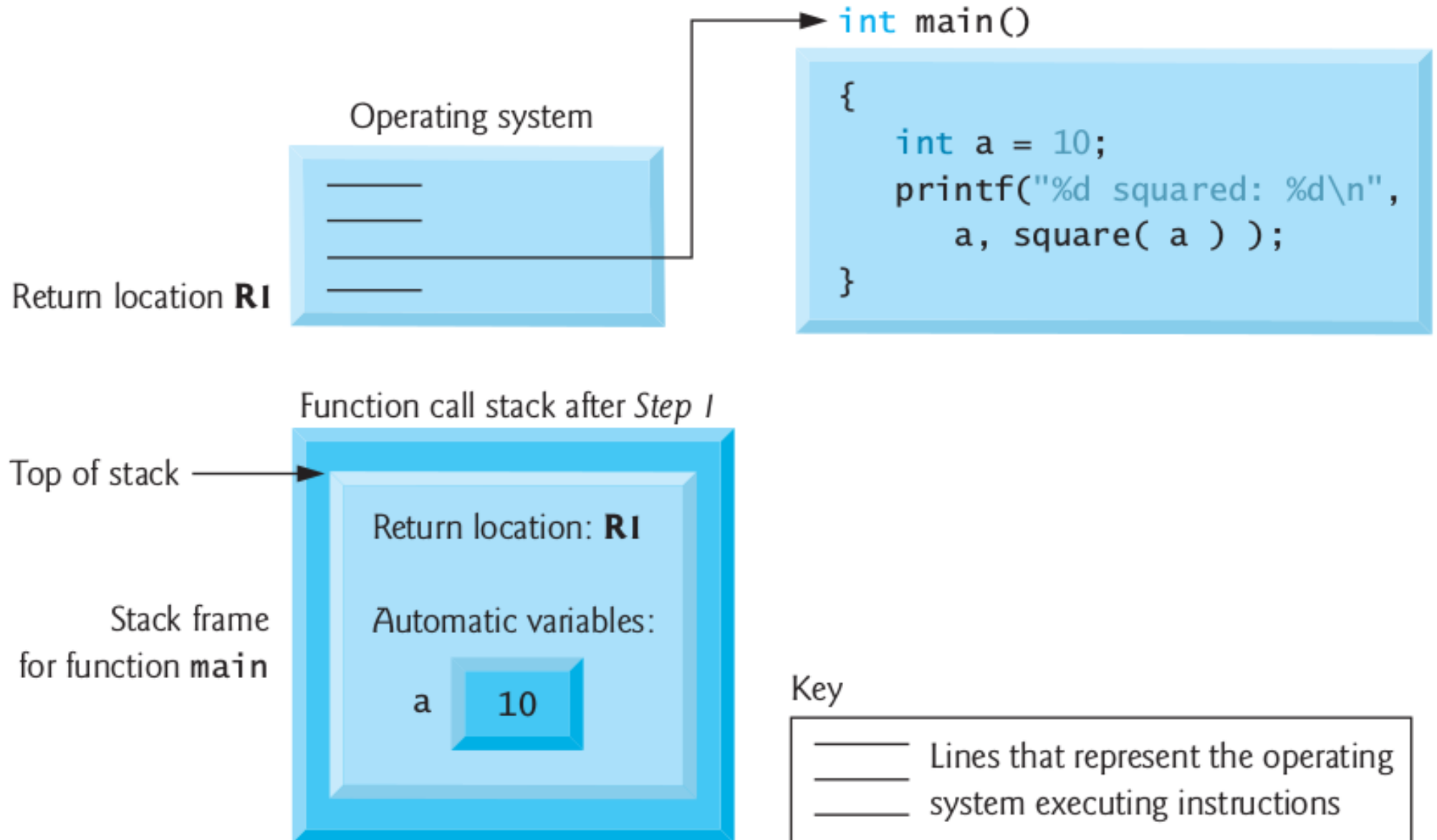
# Types of User-Defined Functions

User-defined functions with

- no arguments and no return value
- no arguments and a return value
- arguments and no return value
- arguments and a return value

# Under the Hood!!

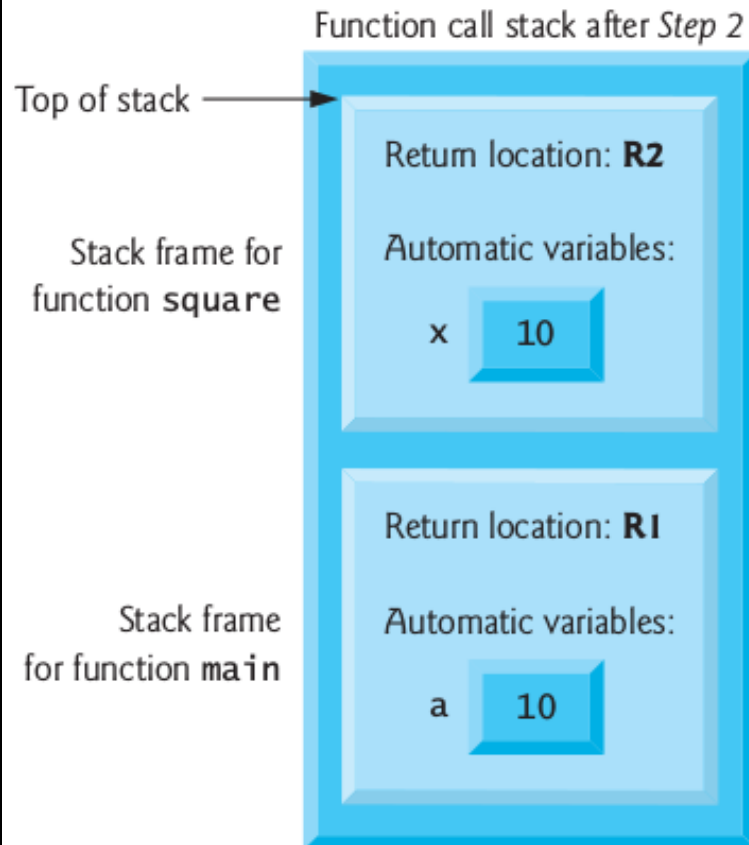
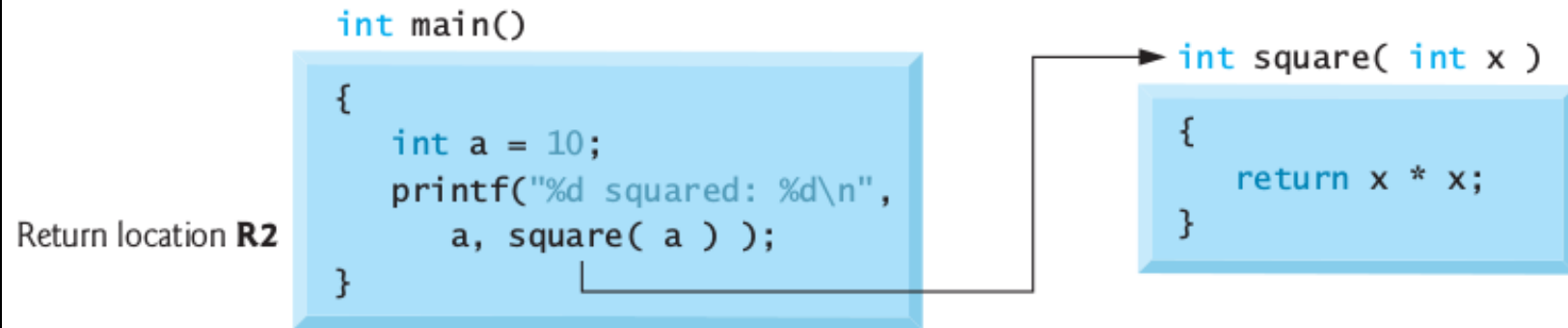
Step 1: Operating system invokes `main` to execute application





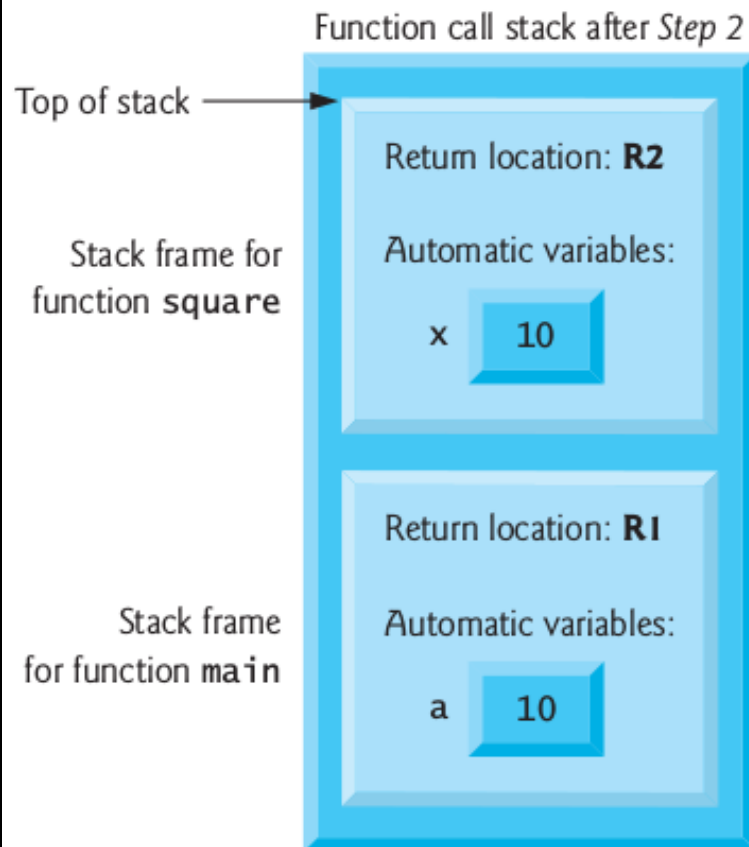
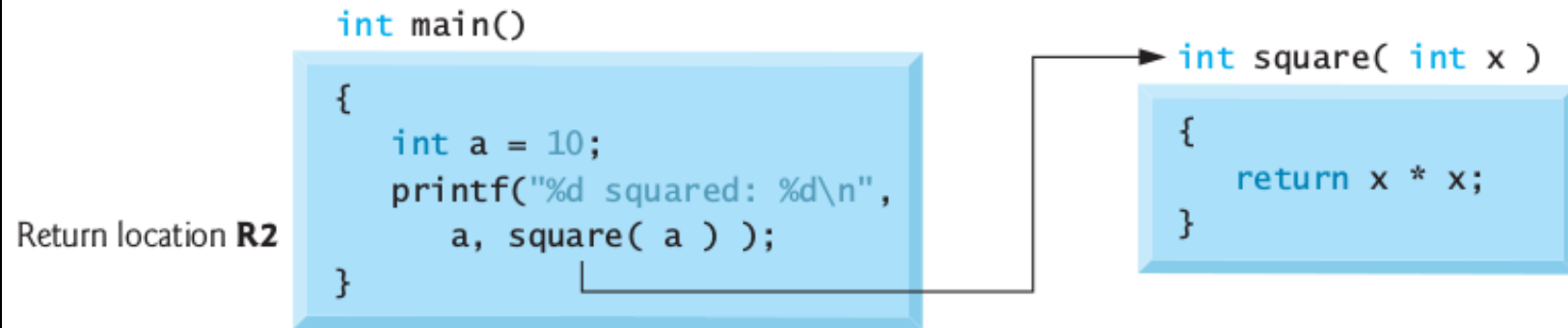
# Under the Hood!!

Step 2: `main` invokes function `square` to perform calculation



# Under the Hood!!

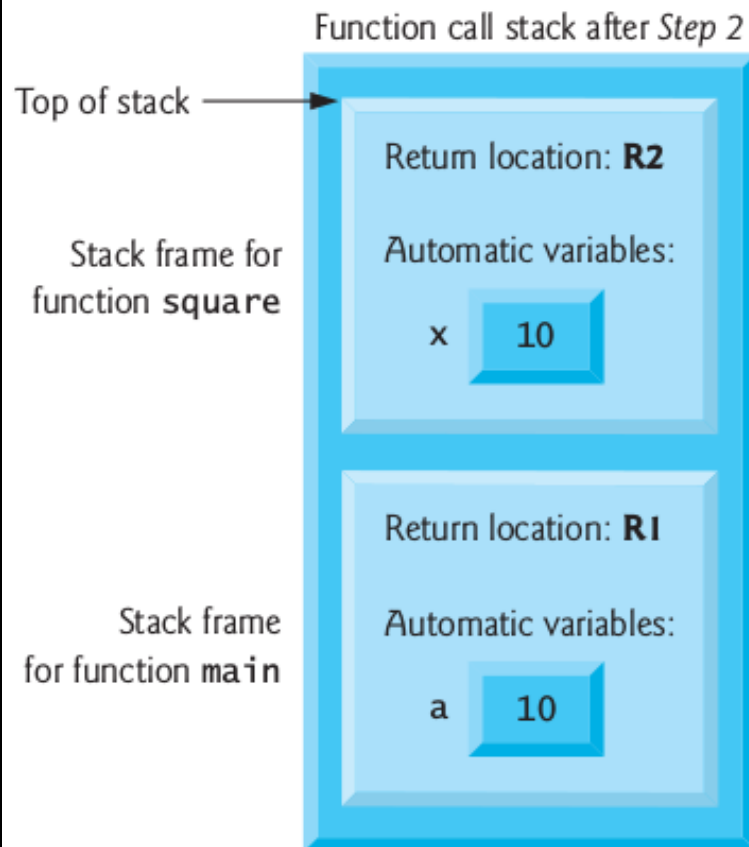
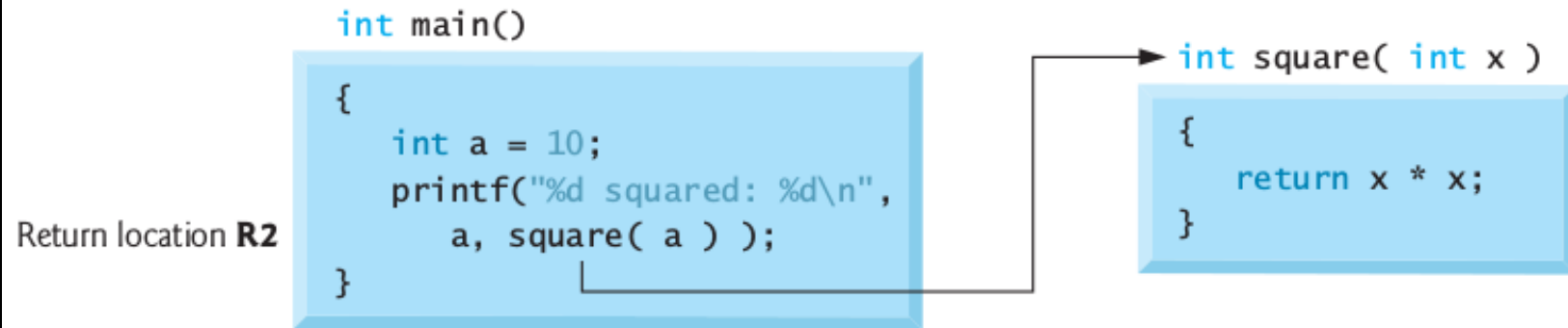
Step 2: `main` invokes function `square` to perform calculation



Observe the scope of variables `a` and `x`. Even if both functions use the same variable say `x` there will be no conflict!!

# Under the Hood!!

Step 2: `main` invokes function `square` to perform calculation



Is it possible to have variables whose scope is both in `main()` and `square()` functions?

# Under the Hood!!

Step 3: `square` returns its result to `main`

```
int main()
```

```
{  
    int a = 10;  
    printf("%d squared: %d\n",  
        a, square( a ) );  
}
```

Return location **R2**

```
int square( int x )
```

```
{  
    return x * x;  
}
```

Function call stack after Step 3

Top of stack

Return location: **R1**

Automatic variables:

a 10

Stack frame  
for function `main`

# Under the Hood!!

Step 3: square returns its result to main

```
int main()
```

```
{  
    int a = 10;  
    printf("%d squared: %d\n",  
        a, square( a ) );  
}
```

Return location **R2**

```
int square( int x )
```

```
{  
    return x * x;  
}
```

Function call stack after Step 3

Top of stack

Return location: **R1**

Automatic variables:

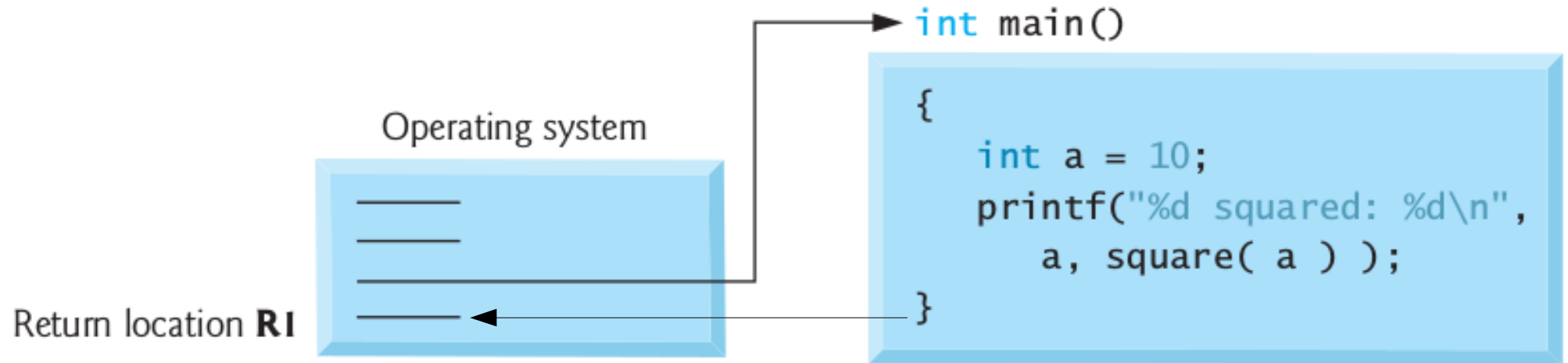
a 10

Stack frame  
for function main

Observe the life of variable `x`. After the function `square()` returns value to `main()` the variable `x` is non-existent!!

# Under the Hood!!

Step 1: Operating system invokes `main` to execute application



Once the `main()` returns, the OS proceeds as usual picking up from return location `R1` as shown above

Key

— Lines that represent the operating system executing instructions

# CSE102

## Computer Programming

(Next Topic)



$$A + B = C$$