Functions

IC100

November 25, 2022

A Modern Smartphone

- Surf the net
  - Input: Web address
  - Output: Desired page
- Book tickets
  - Input: userid, password, booking info, bank info
  - Output: Ticket
- Send email
  - Input: email address of receiver, mail text
  - Output: --
- Take photos
  - Input: --
  - Output: Picture
- Talk (we can do that too!!)
  - Input: Phone number
  - Output: Conversation (if lucky)
- …

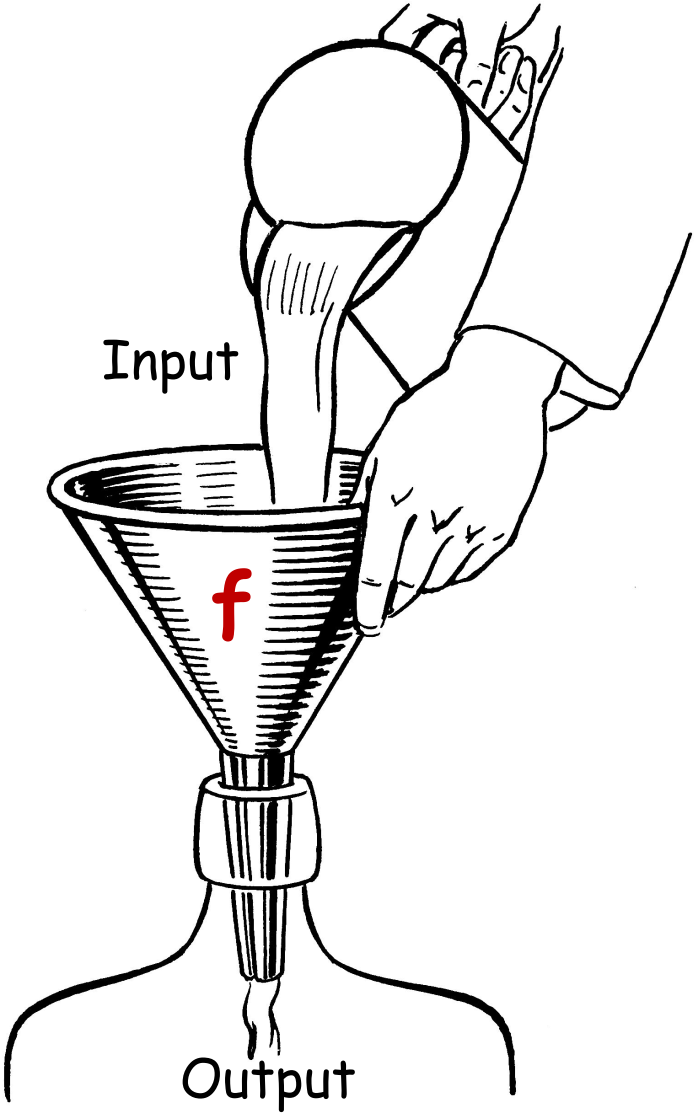
Lots of Related/Unrelated Task to Perform

- Divide and Conquer
  - Create well defined sub tasks
  - Work on each task independently
    - Development, Enhancements, Debugging
- Reuse of tasks
  - Email and Chat apps can share spell checker
  - Phone and SMS apps can share dialer
- C facilitates this using Functions

Function

- An independent, self-contained entity of a C program that performs a well-defined task
- It has
  - Name: for identification
  - Arguments: to pass information from outside world (rest of the program)
  - Body: processes the arguments do something useful
  - Return value: To communicate back to outside world
    - Sometimes not required
- A function will carry out its intended task whenever it is called or invoked
  - Can be called multiple times

Parts of a Function

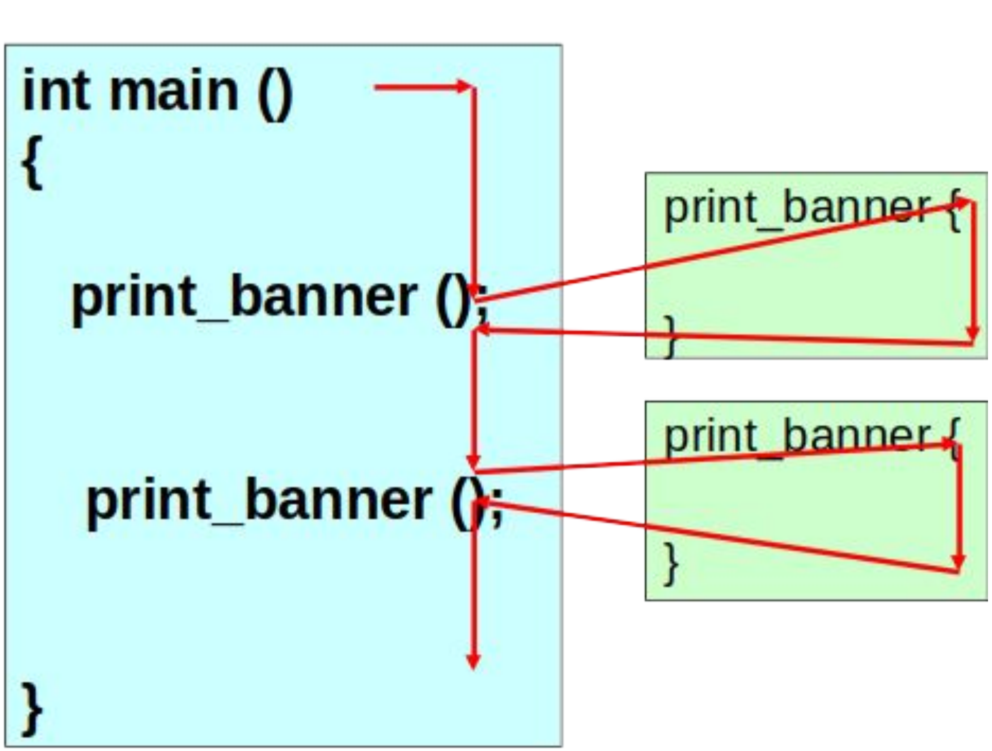


Function Control Flow

```
void print_banner ()
{
   printf(“************\n”);
}
```

```
void main ()
{
    . . .
    print_banner ();
    . . .
    print_banner ();

}
```

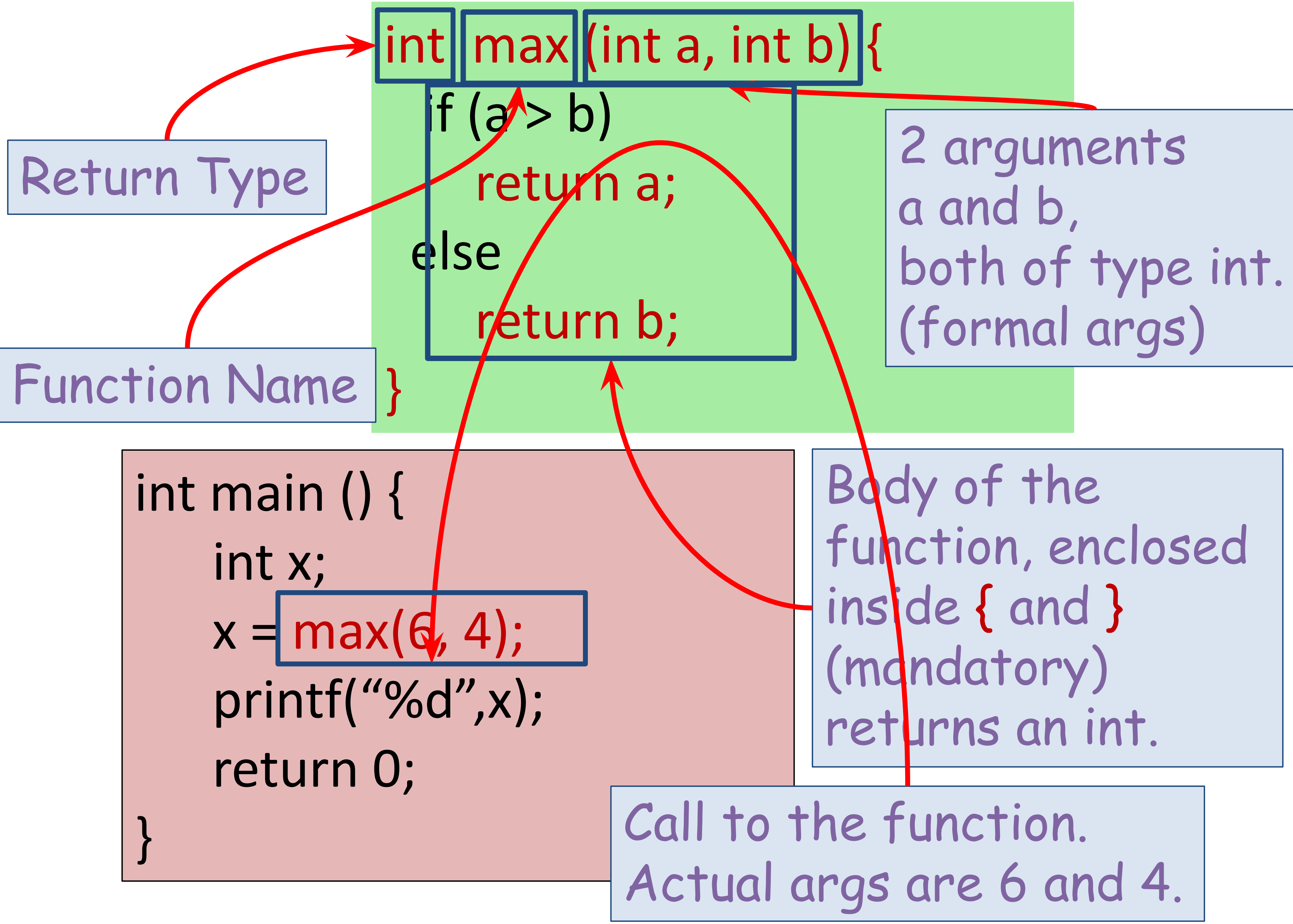


```
int main ()
{

   print_banner ();


    print_banner ();


}
```



```
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)

Body of the function, enclosed inside { and } (mandatory) returns an int.

```
int main () {
    int x;
    x = max(6, 4);
    printf("%d",x);
    return 0;
}
```

Call to the function. Actual args are 6 and 4.

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?

Example : Maximum of 3 numbers

```
int main(){
   int a, b, c, m;

   /* code to read
    * a, b, c */

   if (a>b){
     if (a>c) m = a;
     else m = c;
   }
   else{
     if (b>c) m = b;
     else m = c;
   }

   /* print or use m */

   return 0;
}
```

```
int max(int a, int b){
   if (a>b)
     return a;
   else
     return b;
}

int main() {
   int a, b, c, m;

   /* code to read
    * a, b, c */

   m = max(a, b);
   m = max(m, c);
   /* print or use m */

   return 0;
}
```

This code can scale easily to handle large number of inputs (e.g.: max of 100 numbers!)

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 (Divide-and-conquer approach)
- The main tool in C for modular programming
- Abstraction: hide internal details (library functions)

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.
- sqrt(…), pow(…) are math functions in math.h

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
  - Called by specifying the function name and parameters in an instruction in the calling function
- Similar to operator application

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

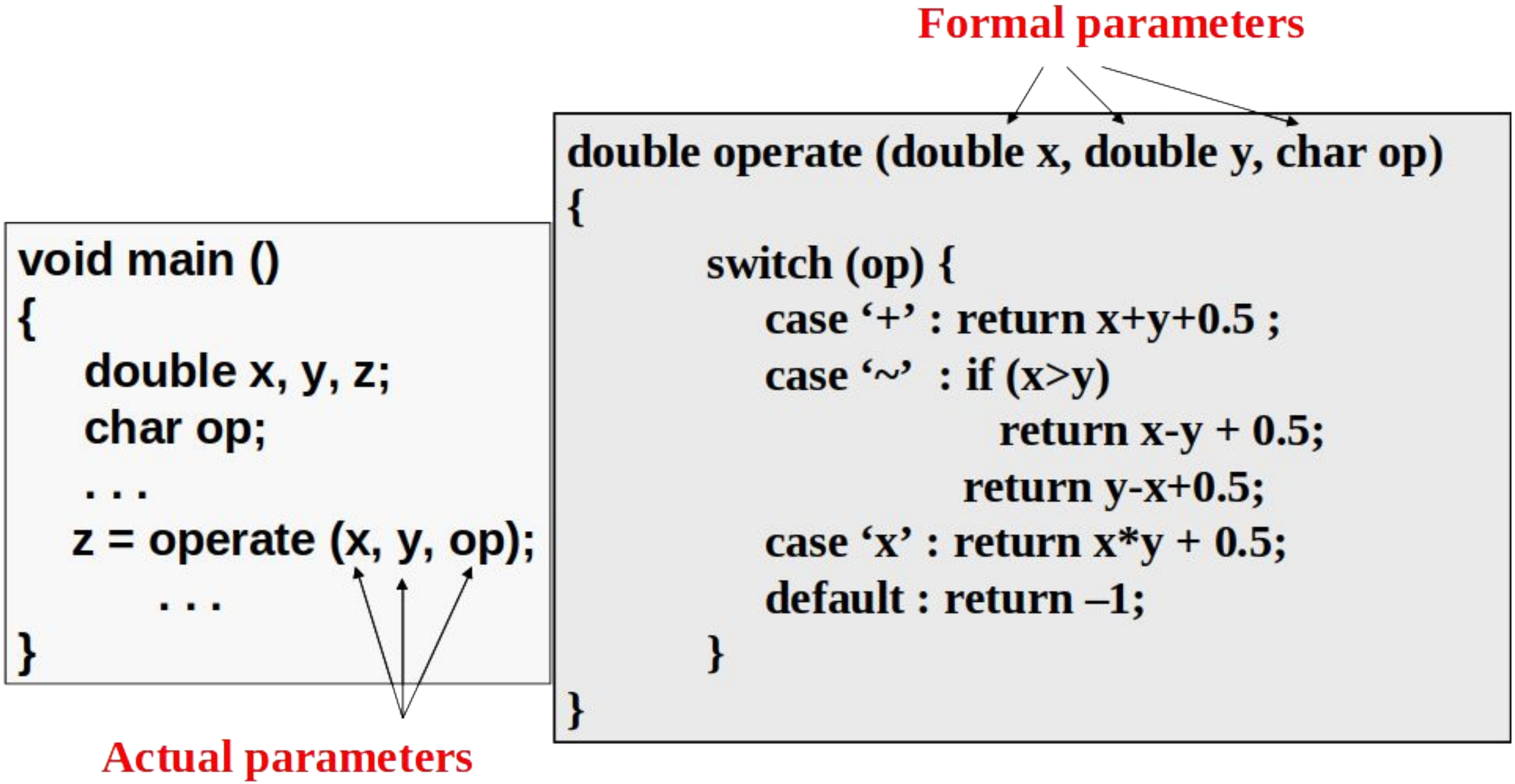
max(5, 3) is an expression of type integer that evaluates to 5

Function Call

- Since a function call is an *expression*
  - it can be used anywhere an expression can be used
  - subject to type restrictions
  - The function call must include a matching actual parameter for each formal parameter
  - Position of an actual parameters in the parameter list in the call must match the position of the corresponding formal parameter in the function definition
  - The formal and actual arguments must match in their data types

printf(“%d”, max(5,3));	prints 5
max(5,3) – min(5,3)	evaluates to 2
max(x, max(y, z)) == z	checks if z is max of x, y, z
if (max(a, b)) printf(“Y”);	prints Y if max of a and b is not 0.

Example



Formal parameters

```
double operate (double x, double y, char op)
{
        switch (op) {
            case ‘+’ : return x+y+0.5 ;
            case ‘~’  : if (x>y)
                          return x-y + 0.5;
                        return y-x+0.5;
            case ‘x’ : return x*y + 0.5;
            default : return –1;
        }
}
```

```
void main ()
{
    double x, y, z;
    char op;
    . . .
   z = operate (x, y, op);
        . . .
}
```

Actual parameters

When the function is executed, the value of the actual parameter is copied to the formal parameter

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 any value

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

Function Not Returning Any Value

- Example: A function which prints if a number is divisible by 7 or not

```
void  div7 (int n)
{
   if  ((n % 7) == 0)
      printf (“%d is divisible by 7”, n);
   else
      printf (“%d is not divisible by 7”, n);
   return;
}
```

- Return type is void here.

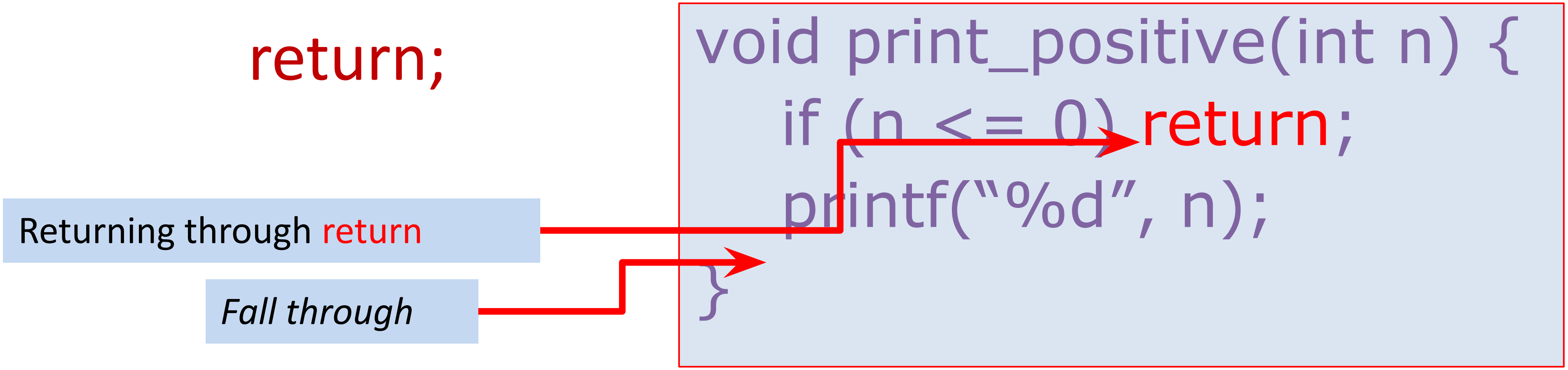
Returning from a Function: return Statement

- If return type is not void, then the function should return a value:

  return return_expr;

- If return type is void, the function may *fall through* at the end of the body or use a return without return_expr:

  return;



```
void print_positive(int n) {
    if (n <= 0) return;
    printf("%d", n);
}
```

Returning through return

Fall through

Returning from a Function: return Statement

- When a return statement is encountered in a function definition
  - control is immediately transferred back to the statement making the function call in the parent function
- A function in C can return only ONE value or NONE.
  - Only one return type (including void)

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

Function Declaration

return_type function_name (list_of_args);

- Examples:
  - int max(int a, int b);
  - int max(int x, int y);
  - int max(int , int);

All 3 declarations are equivalent! Since there is no BODY here, argument names do not matter, and are optional.

- Position in program: Before the call to the function
  - allows compiler to detect inconsistencies
  - Header files (stdio.h, math.h,…) contain declarations of frequently used functions
  - #include <…> just copies the declarations

Another Example

```
int  factorial (int m)
{
   int i, temp=1;
   for (i=1; i<=m; i++)
           temp = temp
   * i;
   return (temp);
}
```

```
void main()
{
   int  n;
   for  (n=1; n<=10; n++)
           printf (“%d! = %d \n”,
              n, factorial (n) );
}
```

Output

```
1! = 1

2! = 2

3! = 6  …….. upto 10!
```

Some more points

- A function cannot be defined within another function
  - All function definitions must be disjoint
- Nested function calls are allowed
  - A calls B, B calls C, C calls D, etc.
- The function called last will be the first to return
- A function can also call itself, either directly or in a cycle
  - A calls B, B calls C, C calls back A.
  - Called recursive call or recursion