Programming I

Functions

Dimitrios Michail

Dept. of Informatics and Telematics
Harokopio University of Athens

Functions

Until now we wrote our programs as a monolithic piece of code inside the `main` function.

1. What would happen if our program was 200000 lines of code?
2. How could we execute the same or similar code several times?

Functions

1. Split our program into parts.
2. "Divide and Conquer" philosophy
3. The parts represent smaller subproblems

Analogy with real-world:

- An employer assigns a task to an employee without knowing much about the details of the assignment.

Functions

1. Subproblems are easier to handle
2. Create one function for each subproblem
3. The code is looking better and is easier to manage and maintain
4. Use existing functions in order to write new programs which solve new problems
5. Do not repeat the same code multiple times in our programs
6. Abstraction: Hide complex details which are not relevant to the programmer or to the problem we are trying to solve.

Functions

1. Subproblems are easier to handle
2. Create one function for each subproblem
3. The code is looking better and is easier to manage and maintain
4. Use existing functions in order to write new programs which solve new problems
5. Do not repeat the same code multiple times in our programs
6. Abstraction: Hide complex details which are not relevant to the programmer or to the problem we are trying to solve.

Functions

1. Subproblems are easier to handle
2. Create one function for each subproblem
3. The code is looking better and is easier to manage and maintain
4. Use existing functions in order to write new programs which solve new problems
5. Do not repeat the same code multiple times in our programs
6. Abstraction: Hide complex details which are not relevant to the programmer or to the problem we are trying to solve.

Functions

1. Subproblems are easier to handle
2. Create one function for each subproblem
3. The code is looking better and is easier to manage and maintain
4. Use existing functions in order to write new programs which solve new problems
5. Do not repeat the same code multiple times in our programs
6. Abstraction: Hide complex details which are not relevant to the programmer or to the problem we are trying to solve.

Functions

1. Subproblems are easier to handle
2. Create one function for each subproblem
3. The code is looking better and is easier to manage and maintain
4. Use existing functions in order to write new programs which solve new problems
5. Do not repeat the same code multiple times in our programs
6. Abstraction: Hide complex details which are not relevant to the programmer or to the problem we are trying to solve.

Functions

1. Subproblems are easier to handle
2. Create one function for each subproblem
3. The code is looking better and is easier to manage and maintain
4. Use existing functions in order to write new programs which solve new problems
5. Do not repeat the same code multiple times in our programs
6. Abstraction: Hide complex details which are not relevant to the programmer or to the problem we are trying to solve.

Functions in the C Library

The standard library in C provides a lot of functions. Some of them we have already seen:

1. `printf()` prints stuff in standard output (screen)
2. `scanf()` reads from standard input (keyboard)
3. `getchar()` reads character from standard input (keyboard)

When we write programs we try not to invent the wheel! Use the already provided library functions.

Defining a Function in C

The general form of a function definition can be seen below:

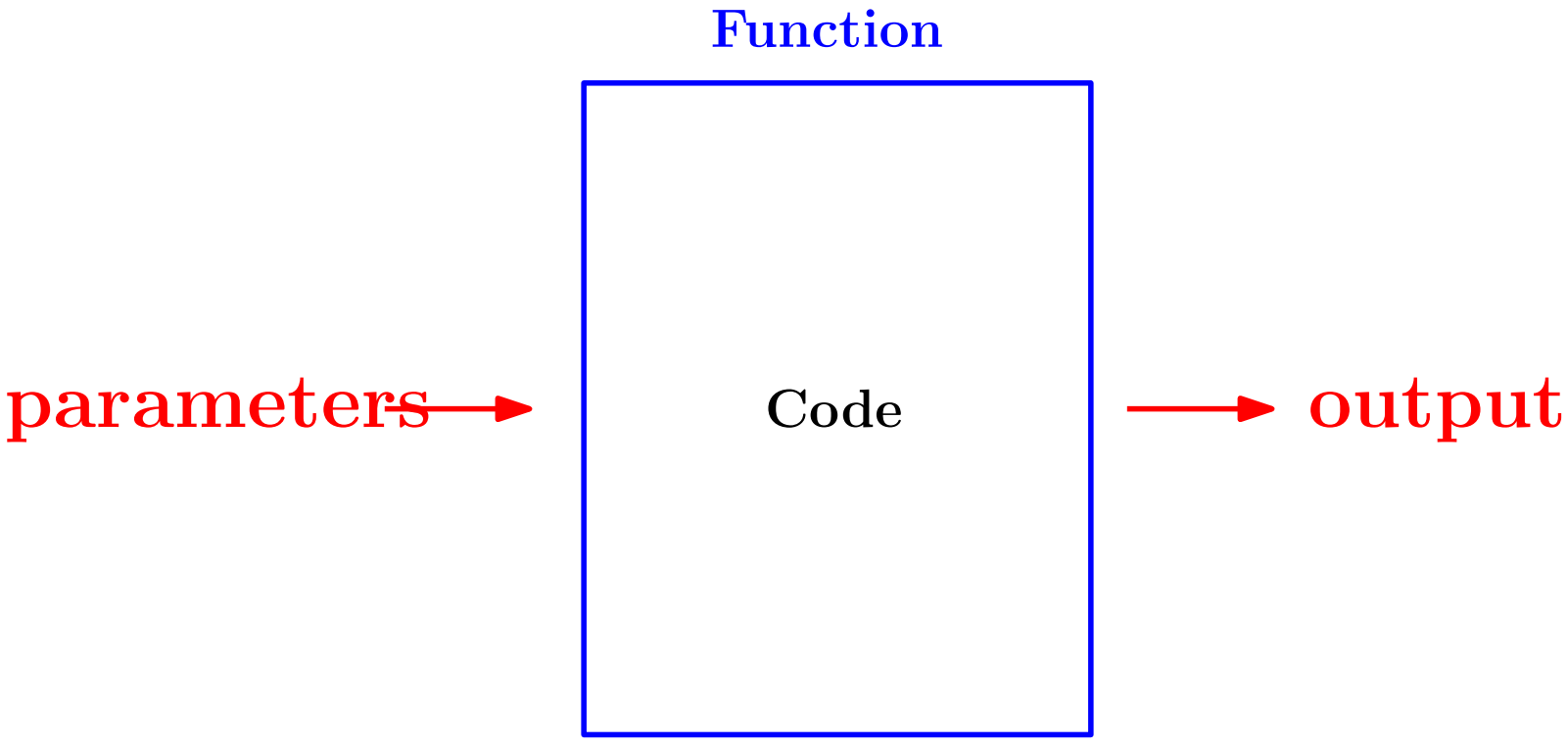
```
return-value-type  name(comma-separated-parameter-list) {
    declarations

    statements
}
```

Every function has:

1. a unique name
2. a list of input parameters
3. a return value

Function



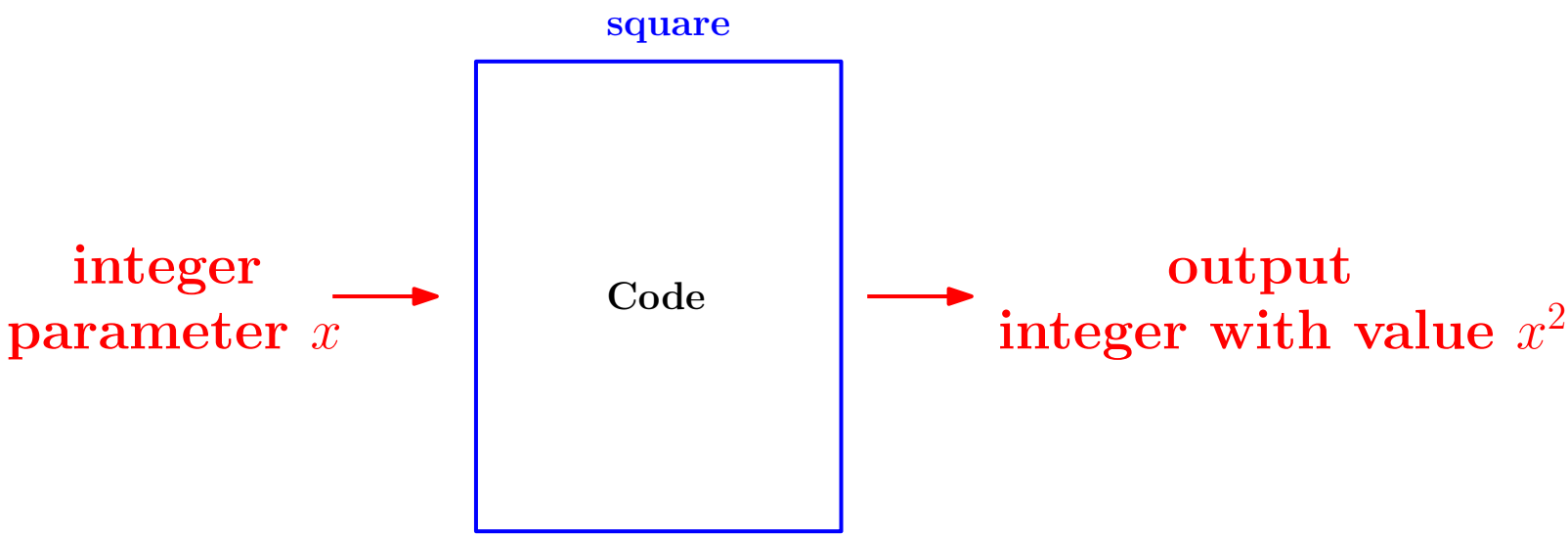
Function Definition Example

Below we can see how to define a function which calculates the square of an integer:

```
int square(int x) {
    return x * x;
}
```

- the function takes one parameter with name `x` and type integer and
- returns an integer

Square Function



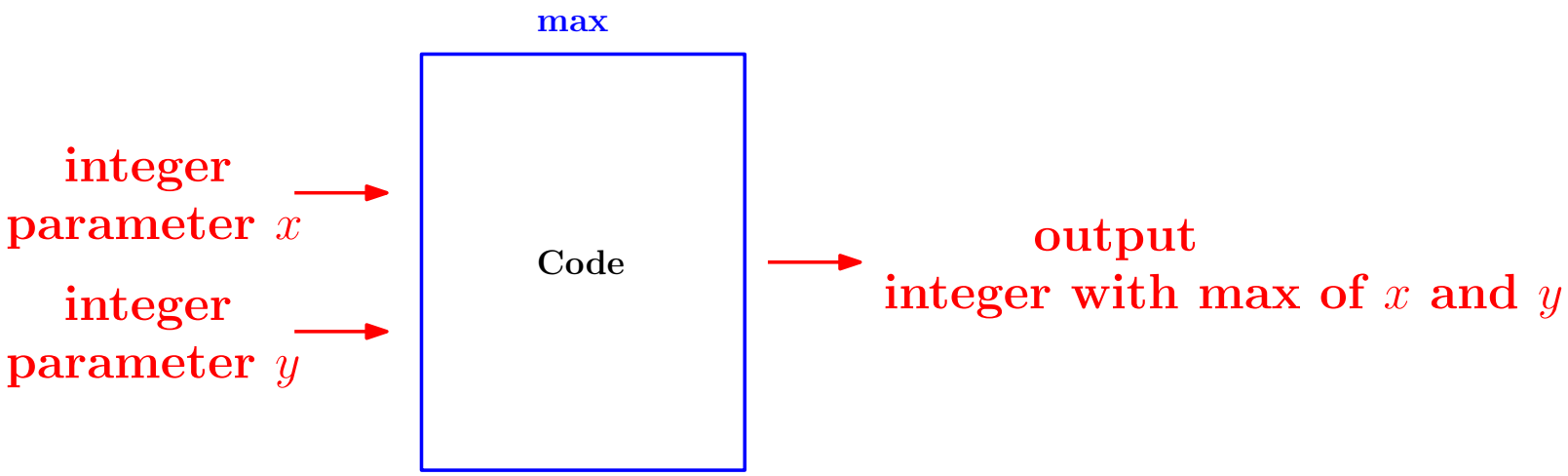
Function Definition Example

Let us also define a function which calculates the maximum of two integers:

```
int max(int x, int y) {
    if (x > y) {
        return x;
    } else {
        return y;
    }
}
```

- the function takes two integer parameters `x` and `y`
- and returns an integer

Function for Maximum of 2 Integers



Function Code

In a function we write code exactly like the way we write code in `main()`.

```c
#include <stdio.h>

int max(int a, int b, int c) {
    int max;

    max = a;
    if (b > max)
        max = b;
    if (c > max)
        max = c;
    return max;
}

int main() {
    printf("max of 1, 2 and 3 = %d\n", max(1,2,3));

    return 0;
}
```

Local Variables

```
int max(int a, int b, int c) {
    int max;

    max = a;
    if (b > max)
        max = b;
    if (c > max)
        max = c;

    return max;
}
```

The function above has 4 local variables:

- the parameters a, b and c
- the variable max

These variables are valid only inside the function.

Calling a Function

We call a function by using its name and providing expressions for its parameters inside parentheses.

```
int main() {
    int i, j, k;
    int m;

    printf("Give 3 values\n");
    scanf("%d%d%d", &i, &j, &k);

    m = max(i, j, k);
    printf("maximum = %d\n", m);

    return 0;
}
```

The return value must be either assigned to some variable or used directly inside another expressions, e.g. print it with `printf()`.

Return Type

When defining a function we must specify the return type.

```
int max(int x, int y) {
    return (x>y)?x:y;
}
```

What do we do if we do not want to return anything?

Return Type

C provides us with the reserved word `void` in order to specify that a function returns nothing.

```
void printstars(int n) {
    int i;

    for(i = 0; i < n; i++)
        putchar('*');

    return;    /* could be omitted */
}
```

The function above prints `n` asterisks and does not return anything.

Returning from a Function

The reserved word **return** terminates the execution of a function.

```
void printstars(int n) {
    int i;

    for(i = 0; i < n; i++)
        putchar('*');

    return;    /* FUNCTION END HERE */

    for(i = 0; i < n; i++)
        putchar('*');
}
```

The code after the **return** statement will not be executed.

Program Flow

The flow changes when a function is called.

- `f(a, b, c)`

1. first all expressions `a`, `b` and `c` are computed,
2. the computed values for `a`, `b` and `c` are copied to the parameters of `f`,
3. the program jumps to the first instruction of `f`,
4. the instructions in `f` are executed until we meet
   - `return`
   - end of function, meaning `}`,
5. the execution jumps back
6. the expression `f(a,b,c)` acquires the value that was returned by the function
7. the program continues by executing the next instruction

Calling a Function

```c
#include <stdio.h>

void printstars(int n) {
    int i;
    for(i = 0; i < n; i++)
        putchar('*');
}

int main() {
    int i;
    for(i = 1; i <= 5; i++) {
        printstars(i);
        printf("\n");
    }
    return 0;
}
```

When calling a function, the program jumps to the first instruction of the function.

Calling a Function

```c
#include <stdio.h>

void printstars(int n) {
    int i;
    for(i = 0; i < n; i++)
        putchar('*');
}

int main() {
    int i;
    for(i = 1; i <= 5; i++) {
        printstars(i);
        printf("\n");
    }
    return 0;
}
```

Before the first instruction is executed, the parameter `n` is initialized with the value we provided when calling the function.

Calling a Function

```c
#include <stdio.h>

void printstars(int n) {
    int i;
    for(i = 0; i < n; i++)
        putchar('*');
}

int main() {
    int i;
    for(i = 1; i <= 5; i++) {
        printstars(i);
        printf("\n");
    }
    return 0;
}
```

The execution of the function ends either when we call `return` or if the execution reaches the final statement.

Calling a Function

```c
#include <stdio.h>

void printstars(int n) {
    int i;
    for(i = 0; i < n; i++)
        putchar('*');
}

int main() {
    int i;
    for(i = 1; i <= 5; i++) {
        printstars(i);
        printf("\n");
    }
    return 0;
}
```

After the function ends, the program jumps back to the line where we called the function and continues normally.

Calling a Function from a Function

```c
#include <stdio.h>

void printchar(char c, int s) {
    int i;
    for(i = 0; i < s; i++)
        putchar(c);
}

void printstarspaces(int s, int n) {
    printchar(' ', s);
    printchar('*', n);
    printchar(' ', s);
}

int main() {
    int i, n = 9;
    for(i = 1; i <= n; i+=2) {
        printstarspaces((n-i)/2, i);
        printf("\n");
    }
    return 0;
}
```

Inside a function we can call other functions.

Calling a Function from a Function

```c
#include <stdio.h>

void printchar(char c, int s) {
    int i;
    for(i = 0; i < s; i++)
        putchar(c);
}

void printstarspaces(int s, int n) {
    printchar(' ', s);
    printchar('*', n);
    printchar(' ', s);
}

int main() {
    int i, n = 9;
    for(i = 1; i <= n; i+=2) {
        printstarspaces((n-i)/2, i);
        printf("\n");
    }
    return 0;
}
```

What does this code print?

Archetypes: Declarations and Definitions

```c
#include <stdio.h>

int max(int x, int y);  // Here we declare!

int main() {
    int a,b;
    printf("Enter two integers: ");
    scanf("%d%d", &a, &b);
    printf("Maximum is: %d\n", max(a,b));
    return 0;
}

int max(int x, int y) {  // Here we define!
    if (x > y)
        return x;
    else
        return y;
}
```

Archetypes: Declarations and Definitions

```c
#include <stdio.h>

int max(int x, int y);  // declaration

int max(int x, int y) { // definition
    if (x > y)
        return x;
    else
        return y;
}
```

Why to simple declare a function?

- we can call a function which we have not defined

Archetypes: Declarations and Definitions

```c
#include <stdio.h>

int max(int x, int y);  // declaration

int max(int x, int y) { // definition
    if (x > y)
        return x;
    else
        return y;
}
```

Why to simple declare a function?

- we can call a function which we have not defined
- maybe the function is defined in another file

Archetypes: Declarations and Definitions

```c
#include <stdio.h>

int max(int x, int y);  // declaration

int max(int x, int y) { // definition
    if (x > y)
        return x;
    else
        return y;
}
```

Why to simple declare a function?

- we can call a function which we have not defined
- maybe the function is defined in another file
- maybe the function is already compiled and available in machine code

Archetypes: Declarations and Definitions

```c
#include <stdio.h>

int max(int x, int y);  // declaration

int max(int x, int y) { // definition
    if (x > y)
        return x;
    else
        return y;
}
```

Why to simple declare a function?

- we can call a function which we have not defined
- maybe the function is defined in another file
- maybe the function is already compiled and available in machine code
- the compiler has enough information to check that our use of the function is correct

Header Files

From day one we instruct the pre-processor to read file "stdio.h".

```c
#include <stdio.h>

int main() {
    /* code goes here */
}
```

This file is located in a well defined location in the system (possibly different for different OSs or compilers) and among others includes declarations for input/output related functions.

Header Files

What is a library?

- collection of functions and data types
- usually performs some specialized function
- usually already compiled code
- provides header files in order to easily know that is inside the library (function names and types, data types, etc.)
- documentation

The standard library of C already contains several header files.

Basic Header Files

<stdio.h>	functions for input/output
<stdlib.h>	functions for converting numbers between text and numbers and vice-versa, memory allocation, pseudo-random number generation, etc.
<math.h>	mathematical functions
<assert.h>	diagnostics for program checking and validation
<ctype.h>	check properties of characters, convert between small and upper case, etc.

Mathematical Functions and the Type `double`

The type `double`, just like `float`, is an approximate representation of real numbers.

It has bigger precision and can thus represent more numbers.

Mathematica Functions and the Type `double`

The C library declares in `<math.h>` various mathematical functions, such as

```
double sqrt(double x);

double exp(double x);

double log(double x);

double pow(double x, double y);

double sin(double x);
```

and others.

Many of them also have a `float` equivalent.

Calling Functions and Parameters

Programming languages provide 2 standard ways of calling a function:

1. call-by-value
2. call-by-reference

Call-by-value

- The call-by-value method copies the values of the actual parameters into the function's formal parameters.
- The parameters are stored in different memory locations.
- Any changes made inside the functions are not reflected in the actual parameters of the caller.

Call by reference method copies the address of an argument into the formal parameter.

- Changes made in the parameter inside the function alter the passing argument.
- The C language does **not** support call-by-reference, but we can simulate it.

We will see details later on.

Call-by-value

```c
#include <stdio.h>

void increment(int x) {
    x++;
}

int main() {
    int y = 5;

    printf("y = %d\n", y);
    increment(y);
    printf("y = %d\n", y);
}
```

Call-by-value

```c
#include <stdio.h>

void increment(int x) {
    x++;
}

int main() {
    int y = 5;

    printf("y = %d\n", y);
    increment(y);
    printf("y = %d\n", y);
}
```

Both `printf()` calls print the value 5 since function `increment()` works on a different variable `y` than function `main()`.

Call-by-reference

The C language does not support call-by-reference, but can simulate it.

We do not yet have the necessary knowledge to see the details.

But we have seen it in action already when we called the function `scanf()`:

```
#include <stdio.h>

int main() {
    int x;

    printf("Give a number\n");
    scanf("%d", &x);

    /* ... */
}
```

Variable Scoping

A variable is known inside the block that is declared.

```
#include <stdio.h>

void fun() {
    int x = 4;
    /* a is NOT present here */
}

void more-fun() {
    int a = 5;
    /* x is not present here */
}

int main() {
    /* a or x are NOT present here */

    /* ... */
}
```

A variable has scope until the end of the block where it is declared(\{ \}).

Variable Scoping Rules

Function parameters are local variables and thus are accessable only inside the function.

```c
#include <stdio.h>

int min(int x, int y) {
    /* a or b are NOT present here */
    return (x<y)?x:y;
}

int max(int a, int b) {
    /* x or y are NOT present here */
    return (a>b)?a:b;
}

int main() {
    /* a,b,x or y are not present here */

    /* ... */
}
```

A variable has scope until the end of the block where it is declared(\{ \}).

Variable Scoping Rules

Local variables with the same name are different variables.

```c
#include <stdio.h>

int fun(int x) {
    int a = 1;

    /* a is different from a of morefun */
}

int morefun(int y) {
    int a = 2;

    /* a is different from a of fun */
}

int main() {
    /* no variable a is present here */
}
```

Variable Scoping Rules

We can also declare variables outside of a function (global variable).

```c
#include <stdio.h>

void fun() {
    /* a is NOT present here */
}

int a = 5;

void more-fun() {
    /* a IS present here */
}

int main() {
    /* a IS present here */
    /* some code */
}
```

A global variable is known from the point of its declaration and below.

Variable Scoping Rules

In case we declare a local variable with the same name as a global variable, the local variable hides the global one.

```c
#include <stdio.h>

int a = 5;

void fun() {
    int a = 200;
    /* a here is 200 and different from global a */
    a++;
}

void morefun() {
    /* a here has value 5 */
    a++;
}

int main() {
    fun();
    morefun();
    /* a here has value 6 */
}
```

Variable Scoping Rules

In case we declare two local variables with the same name, the inner one hides the outer one.

```c
#include <stdio.h>

int main() {
    int i, a;

    i = 0;
    a = 0;

    while(i++ < 10) {
        int a;
        a = 5;
        printf("%d\n", a); /* a here is the one inside the while */
    }

    printf("%d\n", a); /* a here is the one inside main */

    return 0;
}
```

Static Variables

reserved word `static`

We can declare a variable as `static`.

```
static int x;
static float f;
```

The result depends on the scope of the declaration.

Global Static Variables

A global variable can be declared as **static**.

```c
#include <stdio.h>

static int x;

int main() {
    /* ... */
}
```

Which means that the variable scope will be limited to the actual file in which it is defined and cannot be accessed from other files.

We never talked about multiple files. It is in general possible to define a variable inside a file in C and access it from other files.

Local Static Variables

They still have local scope but have **static** storage. After they get initialized, they remain in memory for the whole duration of the program.

```c
#include <stdio.h>

void foo() {
    static int x = 0;

    printf("function foo has been called %d time(s)\n", ++x);
}

int main() {
    int i;
    for(i = 0; i < 100; i++) {
        foo();
    }
    return 0;
}
```

Every time we call function `foo` the same variable `x` is used which retains its value from the previous calls. However, its scope is local which means that it is accessible only inside the function.

Recursion

Recursion is when a function calls itself.

It is sometimes much easier to implement something using recursion.

Calculating Factorial

The factorial is recursively defined as:

$$n! = \begin{cases} 1 & \text{if } n = 0, \\ n \cdot (n-1)! & \text{if } n > 0. \end{cases}$$

We can directly write in C:

```
int factorial(int n) {
    if (n <= 0)
        return 1;
    return n * factorial(n-1);
}
```

Fibonacci Series

The Fibonacci series

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots$$

starts with 0 and 1 and has the property that each subsequent number is the sum of its two predecessors in the series

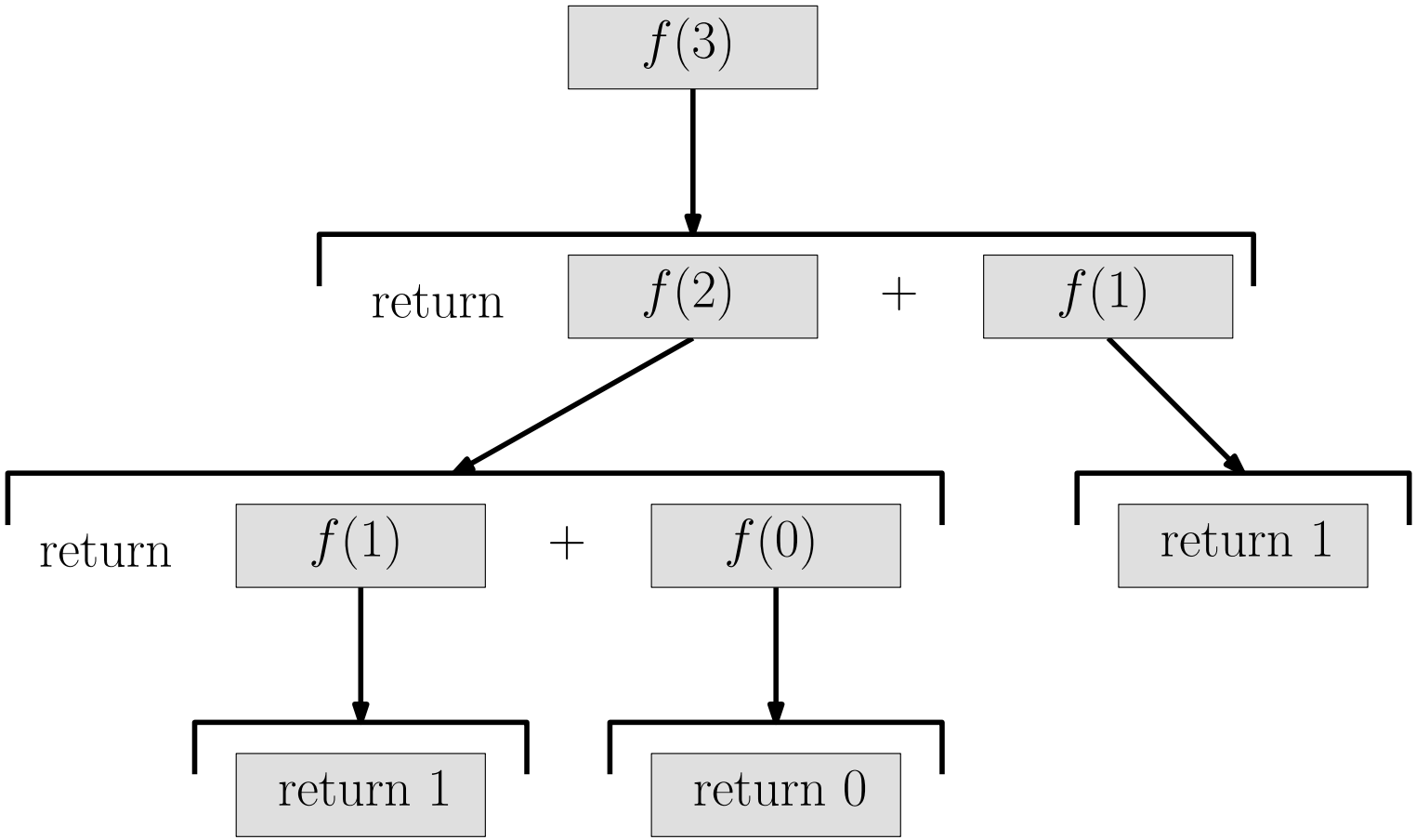
```c
#include <stdio.h>

long fibonacci(long n) {
    if (n == 0 || n == 1)
        return n;
    return fibonacci(n-1) + fibonacci(n-2);
}

int main() {
   long result, number;

   printf("Enter an integer: ");
   scanf("%ld", &number);
   result = fibonacci(number);
   printf("Fibonacci( %ld ) = %ld\n", number, result);
   return 0;
}
```

Fibonacci Series



Iteration with Recursion

The fact that call-by-value is used when calling functions, allows us to perform iterations using recursion.

```c
#include <stdio.h>

void backwards(int x) {
    if (x < 0)
        return;
    printf("%d ", x);
    backwards(x-1);
}

int main() {
    backwards(100);
    return 0;
}
```

Iteration with Recursion

The fact that call-by-value is used when calling functions, allows us to perform iterations using recursion.

```c
#include <stdio.h>

void backwards(int x) {
    if (x < 0)
        return;
    printf("%d ", x);
    backwards(x-1);
}

int main() {
    backwards(100);
    return 0;
}
```

The program above prints

```
100 99 98 ... 2 1 0
```

Iteration with Recursion

The fact that call-by-value is used when calling functions, allows us to perform iterations using recursion.

```c
#include <stdio.h>

void backwards(int x) {
    if (x < 0)
        return;
    printf("%d ", x);
    backwards(x-1);
    printf("%d ", x);
}

int main() {
    backwards(100);
    return 0;
}
```

The program above prints

```
100 99 98 ... 2 1 0 0 1 2 ... 98 99 100
```

Iteration with Recursion

Consider the following code which calculates the sum from 0 to n.

```c
#include <stdio.h>

int main() {
    int i, n, sum;

    scanf("%d", &n);

    for(i = 0, sum = 0; i <= n; i++) {
        sum += i;
    }

    printf("sum = %d\n", sum);
    return 0;
}
```

Can we implement the same program using a recursive function but no loop?

Iteration with Recursion

The followind code calculates the sum from 0 to n using a recursive function.

```c
#include <stdio.h>

int sum(int i) {
    if (i == 0)
        return i;
    return sum(i-1) + i;
}

int main() {
    int n;

    scanf("%d", &n);

    printf("sum = %d\n", sum(n));
    return 0;
}
```

One more recursion example

```c
#include <stdio.h>

void recursive_function(int n) {
    if (n <= 0) {
        printf("Called with non-positive n, stopping..\n");
        return;
    }
    printf("Called with n = %d\n", n);
    printf("Calling myself with n = %d\n", n-1);
    recursive_function(n-1);
    printf("Continuing call with n = %d\n", n);
    printf("Call with n = %d finished, stopping..\n", n);
}

int main() {
    recursive_function(5);
    return 0;
}
```

One more recursion example

```
Called with n = 5
Calling myself with n = 4
Called with n = 4
Calling myself with n = 3
Called with n = 3
Calling myself with n = 2
Called with n = 2
Calling myself with n = 1
Called with n = 1
Calling myself with n = 0
Called with non-positive n, stopping..
Continuing call with n = 1
Call with n = 1 finished, stopping..
Continuing call with n = 2
Call with n = 2 finished, stopping..
Continuing call with n = 3
Call with n = 3 finished, stopping..
Continuing call with n = 4
Call with n = 4 finished, stopping..
Continuing call with n = 5
Call with n = 5 finished, stopping..
```

Why are we seeing the above output?