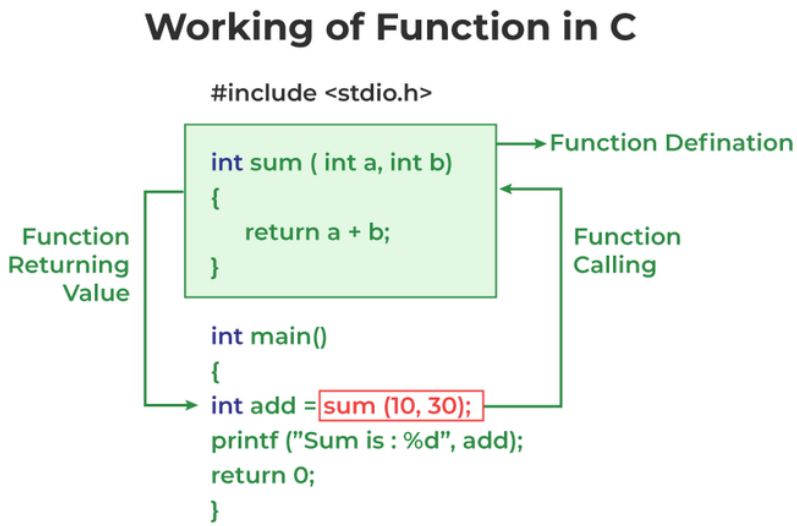
Functions

Pedro Ribeiro

DCC/FCUP

2024/2025



(based and/or partially inspired by Pedro Vasconcelos's slides for Imperative Programming)

Functions

- A function groups together a sequence of instructions with a name
- Each function can **receive arguments** and **return** a value
- Each function is a sub-program with its own statements and instructions

Why use functions?

- To divide the program into **separate components**
  - each function has a clearly identified goal
  - well-defined arguments and expected results
- Can be developed and studied independently
- Can be tested separately
- Can be reused in different programs

Example function

- Functions in C:
  `return_type function_name(arg1, arg2, ...) code_block`

- Example: a function that calculates the arithmetic mean of 2 values:

```
float mean(float a, float b) {
    float x = (a + b) / 2.0;
    return x;
}
```

- The function *identifier* is `mean`
- The `float` type before the identifier indicates the *type* of the result
- The parameters `a` and `b` are the two float values that must be supplied to execute the function
- The *body* of the function is enclosed in curly braces:
  - calculates the average value (using an auxiliary variable `x`)
  - the return statement terminates the function and returns the result to the context where the function was called

Invoking the function

- An expression: `identifier(arg1, arg2, ...)`

```c
int main(void) {
  ...
  int z = mean(x, y);
  ...
}
```

- Program execution starts with `main`
- `main` can call another function and so on
- Only functions called by `main` are executed (directly or indirectly)

- The arguments passed to functions can be any expressions of a valid type:

```c
z = mean(x*0.5, y+1);
```

- You can use the result immediately instead of storing it in a variable:

```c
printf("%f\n", mean(x*0.5, y+1));
```

Example - complete code

- `mean.c` (source code) - read 3 numbers and compute the averages 2 by 2.

```c
#include <stdio.h>

float mean(float a, float b) {
  float x = (a + b) / 2.0;
  return x;
}

int main(void) {
  float x, y, z;
  printf("Enter 3 numbers: ");
  scanf("%f %f %f", &x, &y, &z);
  printf("Means\n");
  printf("%.2f and %.2f: %.2f\n", x, y, mean(x,y));
  printf("%.2f and %.2f: %.2f\n", y, z, mean(y,z));
  printf("%.2f and %.2f: %.2f\n", x, z, mean(x,z));
  return 0;
}
```

```
Enter 3 numbers: 3.5 9.6 10.2
Means
3.50 and 9.60: 6.55
9.60 and 10.20: 9.90
3.50 and 10.20: 6.85
```

Declarations and definitions

- If we put the definition of the function before using it, we don't have to declare anything else
- However, if we use the function before the definition, we must put in a **prototype** declaration: `float average(float, float);`
- Example:

```c
float mean(float , float);       // prototype

int main(void) {
 ...
 printf(..., x, y, mean(x,y)); // use
 ...
 return 0;
}

float mean(float a, float b) { // definition
   x = (a + b) / 2.0;
   return x;
}
```

Functions with no return value

- Sometimes we may want to define functions that **don't** return a value
- We only run them for their *side effects*
  - e.g. printing messages on standard output
- In this case, the result type is `void`
- We *don't need* a return
- We *don't use* the result

Functions with no return value - example

- `tminus.c` (source code) - show messages with time left

```c
#include <stdio.h>

void print_time(int n) {
  printf("T minus %d and counting\n", n);
}

int main(void) {
  print_time(3);
  print_time(2);
  print_time(1);

  return 0;
}
```

```
T minus 3 and counting
T minus 2 and counting
T minus 1 and counting
```

Return statement

- A function with a type other than `void` **must use the return statement** to specify the result
- The general form is: `return expression;`
- There is no need for parentheses around the expression
- Sometimes the expression is a constant or variable, but it can be a complex expression:
  - `return 0;`
  - `return n;`
  - `return (x + y) / 2.0;`

Return statement

- We can use `return` to end the execution of the function in the middle of the body.
- Example:

```
int max(int a, int b) {
  if(a >= b)
    return a; // ends immediately

  // if execution reaches this point,
  // then a < b; so the maximum is b
  return b;
}
```

- Using multiple returns can make it difficult to understand the flow of execution
  Recommendation: only use to terminate a function in special cases (e.g. error)

Return statement

- We can also use `return` in functions that don't return results (`void`)
- In this case `return` only serves to end the execution of the function
- We omit the expression
- Example:

```
void print_time(int n) {
  if (n < 0)
    return; // terminate immediately
  printf("T minus %d and counting\n", n);
}
```

Passing arguments

- C function arguments are **passed by value**
  - each expression is evaluated and its value is copied to a parameter local to the function
- Function parameters therefore behave like *temporary* variables
  - changes to the arguments are **not** visible after the function returns
- Example:

```c
// maximum of 2 values (modifies the first argument)
int max(int a, int b) {
  if (b > a)
    a = b;
  return a;
}

int main(void) {
  int x = 1, y = 2;
  printf("%d\n", max(x,y)); // print 2
  printf("%d %d\n", x, y);  // print 1, 2
  return 0;
}
```

Passing arguments

- Another example:

```c
// Try swapping the values of a, b
// (doesn't work because a,b are temporary)
void swap(int a, int b) {
  int t;
  t = a;
  a = b;
  b = t;
}

int main(void) {
  int x = 1, y = 2;
  swap(x, y);
  printf("%d %d\n", x, y); // print 1, 2
  return 0;
}
```

Functional decomposition

- Functions allow problems to be **broken down into simpler parts**
- Objective: **combine** the parts to solve the original problem
  - analogy: lego pieces
- This **"divide and conquer"** methodology makes it possible to build programs that are both elegant and efficient

Example: printing numbers

- Write a program that prints the digits of an integer number *(from the least significant to the most significant)*
- Examples of execution:

```
12
two
one
```

```
473
three
seven
four
```

```
9008
eight
zero
zero
nine
```

Sub-problem

- Given the value of a digit from 0-9, print the corresponding text in english:

0	"zero"
1	"one"
2	"two"
3	"three"
4	"four"
5	"five"
6	"six"
7	"seven"
8	"eight"
9	"nine"

Auxiliary function

- Let's define an auxiliary function: `void print_digit(int d);`
  - the argument is an integer (the value of the digit)
  - the function prints the english text and does not return anything

- The function definition is long but simple: a sequence of cascading if conditions.

```
void print_digit(int d) {
  if (d == 0) printf("zero");
  else if (d == 1) printf("one");
  else if (d == 2) printf("two");
  else if (d == 3) printf("three");
  else if (d == 4) printf("four");
  else if (d == 5) printf("five");
  else if (d == 6) printf("six");
  else if (d == 7) printf("seven");
  else if (d == 8) printf("eigth");
  else if (d == 9) printf("nine");
  else printf("invalid digit!");
}
```

Note: on future lessons we will learn alternatives to the "cascade" of ifs

Auxiliary function

- Let's now define an auxiliary function: `void digits(int n);`
  - the argument is the number to show the digits
  - it should call `print_digit(d)` for each digit *d* of *n*

- Idea: use a **cycle** to *iterate* through each digit

```
void digits(int n) {
  do {
    print_digit(n % 10);
    n /= 10;
  } while (n>0);
}
```

Main Function

- The main program is now quite simple: it just reads a number and calls the corresponding function:

```c
int main(void) {
  int n;

  scanf("%d", &n);
  digits(n);

  return 0;
}
```

- **Decomposition** into functions has made it possible to:
  - write a program in simpler parts
  - consider part of the problem at a time
  - combine the solutions to solve the original problem