Computer Programming

Functions. Arrays

Robert Varga

Technical University of Cluj-Napoca
Computer Science Department

Course 4

Contents

Contents

Functions - introduction

- a function is a self-contained unit of a program that receives input data through parameters, performs calculations, and returns results
- it is a miniature program
- it is good practice to write functions without secondary effects = changes to external variables
- they should also be deterministic = produce the same results for the same input
- enforcing these recommendations leads to functional programming
- the proper application of functions ensures code reusability, abstraction, and readability

Function declaration

- similarly to variables, all functions must be declared
- the declaration specifies, in this order:
  - return type, function name, list of input parameters with their types
- the declaration of a function is called its **header** or **prototype**
- the declaration can be followed by a semicolon
  - it just defines the function interface and no functionality
- or it can be followed by the function definition
  - specifying the exact operations the function should perform

Function declaration - syntax

```
type function_name(type1 in_param1, type2 in_param2, ...)
```

- similar to how we use functions in mathematics: $y = f(x)$
- the type of the value which is returned from the function is specified first
- it is followed the name of the function
- the formal input parameters are given next inside parentheses

Function declaration - return type

- functions may return any primitive C type
- pointers and structures can also be returned (more details later)
- if no value is returned, the return type should be set to *void*
  - example: printing only
- any value that is returned from the function using the return statement is *converted implicitly* to the return type of the function
- should be specified, if not given, *int* is assumed

Function declaration - name

- the function name should adhere to the same rules as the ones required for naming variables (do not start with numbers, single word, case sensitive)
- multiple words can be linked with underscore or camelCase
- should be suggestive but not long
- it is recommended to contruct it using a verb and an object in order to describe an action (print_digits, convert_binary)

Function declaration - formal parameters

- **parameters** specify the format of the input data
- it is a list of types and names (names are optional)
- syntax is similar to variable declaration
  - however, all types should be given explicitly (cannot be grouped)
  - correct: `f(int a, int b);`
  - also correct: `f(int, int);` - only types are required
  - incorrect: `f(int a, b)` - compilation error
- they are called formal parameters because they only describe the data type and do not hold actual data

Function definition

- it can be done right after function declaration (most common)
  - in this case, the statements comprising the function body are given right after the header inside curly brackets {}
- it can be done later (even in a different source file)
  - in this case the declaration ends with a semicolon and the function definition repeats the function header with names for the formal input parameters
- the return statement inside a function ends its execution
- for non-void functions
  - return also sends a result value to the function caller
  - all execution branches should have return statements, and preferably they should be placed at the end
- for void functions:
  - return statement is optional
  - the function execution ends after the last statement

Function call

- once a function is declared it can be used
- using a function for its result or its secondary effects is termed **calling the function**
- this is achieved by writing the function name and giving it the required input arguments
- the **arguments** are the actual values sent to the function = actual parameters
- the function call can be used in an expression and will be evaluated during run-time
- during program execution, the function definition must be available

Program 4.1 - simple example

```c
#include <stdio.h>
int last_bit(int x){
  if (x&1)
    return 1;
  else
    return 0;
}

int main(){
  int b = last_bit(5);
  printf("%d\n", b);
  int x = 1024;
  printf("%d\n", last_bit(x));
  return 0;
}
```

- the function is declared and defined right after
- it returns an int value and accepts as input parameter an int value
- both branches have a return statement
- function call with a literal (constant number)
- function call with a variable as argument
- calls can be used as expressions

Program 4.2 - advanced example

```
#include <stdio.h>
void reverse_binary(int);

int main(){
  reverse_binary(123);
  return 0;
}

void reverse_binary(int x){
  while(x){
    printf("%d", x&1);
    x >>= 1;
  }
  return;
  puts("after return");
}
```

- the function is declared before *main*
- *void* return type, used for printing
- expects a single int value
- function call, no return value to be saved
- function definition happens later
- return statement ends the function, nothing after it will be executed

Passing by value

- function arguments are always **passed by value** in C
- this means only the contents of variables are passed
- perhaps a more descriptive term would be: passing after copying
- if variables are sent to the function, their values will not change
- the function acts as an independent program with its own variables that do not affect the outside world
- it is possible to obtain a different behavior using another mechanism

Program 4.3 - incorrect interchange

```c
#include <stdio.h>
void interchange(int a, int b){
  int t = a;
  a = b;
  b = t;
  printf("in func a = %d, b = %d\n",
    a, b);
}

int main(){
  int a = 1, b = 2;
  printf("before a = %d, b = %d\n",
    a, b);
  interchange(a, b);
  printf("after  a = %d, b = %d\n",
    a, b);
  return 0;
}
```

- the input arguments are sent by value
- *a* and *b* inside the function are different variables from the ones from *main*
- they just hold the values of *a* and *b* initially
- no changes occur in the context of the main function

Example functions from the library

`int rand();` from stdlib.h

- returns a pseudo-random integer between 0 and RAND_MAX

`double pow(double base, double exponent);` from math.h

- returns $base^{exponent}$
- works with floating point types = imprecise

`float atan2f(float y, float x);` from math.h

- returns the arctangent (angle) of the fraction $\frac{y}{x}$ in the correct quadrant, in radians $\in [-\pi, \pi]$
- always consult the documentation when employing functions from the library

Program 4.4 - incorrect pow

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

int main(){
  long long x = pow(10, 18) - 100;
  printf("%lld\n", x);
  return 0;
}
```

- the pow function works with floating point numbers
- if exact values are required this should be avoided
- the integer arguments 10 and 18 are converted to double
- the result is a double with 15 digit precision = last 3 digits may be wrong
- conversion to long long happens only after calculations are done

Call stack

- it is a memory zone for storing local variables, function arguments, and function return addresses
- it is a stack because values are put on the top of stack, and they are also extracted form the top
- this policy is called Last In First Out (LIFO)
- the size of the stack by default is of the order of MBs
- programmers must not exceed the stack limit
  - use local variables of reduced dimensions
  - limit the number of recursive function calls
- stack limit can be changed, but this is not recommended

Program 4.5 - stack overflow

```c
int f(int n){
  if (n == 0)
    return 0;  //C
  return n + f(n-1);  //B
}

int main(){
  int n = 1e6;
  f(n);  //A
  return 0;
}
```

- recursive function = it calls itself
- line C: halting condition
- line B: line where control should return to after function execution
- line A: source of recursive function calls
- the double literal $1e6$ is in scientific notation
- it is equal to 10^6, conversion to int happens implicitly

Stack snapshot

The stack when the halting condition is reached starting from n = 3:

f	n = 0	line C
f	n = 1	line B
f	n = 2	line B
f	n = 3	line B
main	n = 3	line A

- there are multiple instances of the *f* function on the call stack
- the most recent call is on top
- the calls further down depend on and wait for the result from the upper calls
- each call has its own value for *n* and the line where it should return to after the call ends

Study problem - Greatest Common Divisor

- you are given two integers $a, b \leq 10^9$
- determine their greatest common divisor
- solve the special cases:
  - a and b can be 0
  - common when working with fractions
  - in fact, the method we will derive works on negative numbers and on even larger numbers
- we will look at Euclid's algorithm
- it is efficient, compact, elegant and handles all special cases

Study problem - Greatest Common Divisor

We will exploit the following property to propose an algorithm:

Lemma

If a is a multiple of d and b is a multiple of d, then a-b is a multiple of d.

while $a \neq b$ **do**
  if $a > b$ **then**
    $a = a - b$
  else
    $b = b - a$
  end if
end while
$d = a$

Study problem - Greatest Common Divisor

There are two main issues with the previous approach:

- it is slow if a is large and b is relatively small
- it does not work if the smaller number is 0

If the smaller number is positive the algorithm halts.

- this is because at each step, the larger number is decreased by at least 1
- at every step both a and b are divisible by their gcd
- in the worst case, gcd $= 1$, a and b are reduced until they both become 1

Study problem - Greatest Common Divisor

Recall a relevant theorem for calculating the remainder after division:

Theorem

If a is an integer and d is natural number, then there exist unique integers q and r, $0 \leq r < d$ *such that:*

$$a = qd + r$$

If g is the greatest common divisor of a and b, then g also divides the remainder r of a divided by $b = a\%b$.
This is because if g divides both a and b, then it divides $a - b$, $a - 2b$, ..., $a - qb = r$, by applying the Lemma multiple times, until we obtain a number r less than b.

Code snippet - Greatest Common Divisor

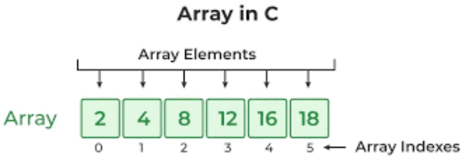
```
int gcd(int a, int b){
  while(b){
    int r = a%b;
    a = b;
    b = r;
  }
  return a;
}
```

- if b is zero the answer is a
- if $a < b$, then a and b are swapped
- otherwise, find the remainder r
- the *gcd* divides this remainder
- continue with *(b, r)* instead of *(a, b)*
- number of steps is approximately $log_{\varphi}(a+b)$, with $\varphi = \frac{1+\sqrt{5}}{2}$

Contents

Arrays - Introduction

- an array consists of multiple variables which occupy a contiguous memory location
- sequences appear naturally in problems
- for example, the first n even numbers
- allows programmatic access to a specific element = the position can be calculated



Array declaration

- arrays are declared by specifying the type, the name and the number of elements in square brackets

  `element_type array_name[nr_elements];`

- the type of each element can be any primitive C type or more advanced
- the number of elements can be a variable, but it will be fixed at the moment of declaration
- multidimensional arrays can declared by giving the number of elements for each dimension

```
int a[5];
int n = 4;
int matrix[n][n];
```

Element access

- an array with *n* elements reserves space for *n* variables of the specified type
- these will occupy a contiguous memory zone (without breaks)
- an element from the array can be accessed for both reading and writing by:

  `array_name[element_pos]`
- arrays use 0-indexing for a good reason
  - the first element is located at position 0
  - the last element is located at position n-1
- Caution!
  - the n-th position is not part of the array
  - accessing the n-th position is a mistake and can lead to run-time error (undeterministic behavior)

Array initialization

- arrays can be initialized right at their declaration using a syntax with braces {}
- elements are enumerated, starting from the first (0th)
- without initialization, the elements of the array are random values
- if an array is initialized with values, its dimension can be deduced automatically and it can be omitted from the declaration
- if the size of the array is specified and initialization is also done, the unspecified elements are initialized with 0

```
int a[5] = {1, 2, 0};
// a[0] = 1, a[1] = 2, a[2] = a[3] = a[4] = 0
double b[] = {1.2, -0.5};
// the number of elements is deduced to be 2
```

Array initialization - advanced

- elements from specific positions can be initialized with the [] syntax
- this is especially useful for sparse arrays
- just provide values for specific positions:

```
int a[15] = {[2] = 29, [9] = 7, [14] = 28};
```

- in any order:

```
int a[15] = {[14] = 28, [2] = 29, [9] = 7};
```

- can be mixed with classical initialization:

```
int a[15] = {0, 1, 2, [4] = 4, [3] = 3};
```

- if the dimension of the array is given, all unspecified positions are filled with 0

```
int I[][] = {[0][0] = 1, [1][1] = 1};
```

Arrays as function parameters

- arrays can be used as input parameters to functions
- in the following, `name` refers to the name of the array
- for one-dimensional arrays
  - in the function prototype: `type name[]` or `type name[dim]`
  - in function call: `name`
  - if the dimension is a variable, it must be sent before the array
- for multi-dimensional arrays
  - in the function prototype: `type name[][dim2]`
  - the first dimension is optional
  - in function call: `name`
  - if the dimensions are variable, they must be sent before the array

Program 4.6 - Arrays example

```
#include <stdio.h>

int main()
{
  int n;
  printf("n = ");
  scanf("%d", &n);
  int a[n];
  for(int i=0; i<n; i++){
    printf("a[%d] = ", i);
    scanf("%d", &a[i]);
  }
  for(int i=0; i<n; i++)
    a[i] = 2*a[i];
  for(int i=0; i<n; i++)
    printf("a[%d] = %d\n", i, a[i]);
  return 0;
}
```

- the number elements should be known before declaring the array
- during reading, array elements are preceded by the & operator
- elements can be read and written
- arrays in C are 0-indexed

Array dimension

- there is an instruction called sizeof for determining the size of any variable or type in bytes
- it can behave as a function, which receives a variable or a type as input
- for arrays it returns the size of the whole array
- to find the number of elements, the array size should be divided by the size of a single element

```
int x = 5;
printf("%u %u", sizeof(x), sizeof(int));
int a[5];
printf("%u %u", sizeof(a), sizeof(a)/sizeof(a[0]));
```

Program 4.7 - 2D arrays example

```c
#include <stdio.h>

void read_mat(int n, float a[][n]){
  for(int i=0; i<n; i++){
    for(int j=0; j<n; j++){
      printf("a[%d][%d] = ", i, j);
      scanf("%f", &a[i][j]);
    }
  }
}

void diag(int n, float a[][n], float d[]){
  printf("%d\n", sizeof(d));
  for(int i=0; i<n; i++)
    d[i] = a[i][i];

}
```

- read_mat has as input the variable dimension *n* and the matrix *a*
- the first dimension can be omitted when specifying parameters
- sizeof inside a function on an array doesn't work as expected

Program 4.7 - 2D arrays example - main

```
int main(){
  int n=2;
  float a[n][n];
  float d[n];
  read_mat(n, a);
  diag(n, a, d);
  for(int i=0; i<n; i++)
    printf("%f ", d[i]);
  return 0;
}
```

- array dimension must be known before declaring it
- single or multidimensional arrays are sent to functions just by their names

Program 4.8 - Stack overflow 2

```c
int f(int n){
  int a[n];
}

int main(){
  f(1000000);     //A
  return 0;
}
```

- arrays can get very large
- *a* is declared inside the function *f*
- the required space is reserved on the stack
- will likely result in run-time error, depending on the size of stack

Program 4.9 - Interchange

```
#include <stdio.h>
void interchange(int ab[]){
  int t = ab[0];
  ab[0] = ab[1];
  ab[1] = t;
  printf("in func a = %d, b = %d\n",
    ab[0], ab[1]);
}
int main(){
  int ab[] = {1, 2};
  printf("before a = %d, b = %d\n",
    ab[0], ab[1]);
  interchange(ab);
  printf("after  a = %d, b = %d\n",
    ab[0], ab[1]);
  return 0;
}
```

- why does the swap work?