

Functions

Functions in C

Function. A **function** in a programming language is a specially created, *named* block of code.

Typically, a **function** is a block of code that is written to achieve a specific *single goal or objective* or to *perform a specific computation/ compute and report a specific value*.

Functions have the following properties:

- Function declaration.
- Function definition.
- Return type/ return value.
- Input parameters/arguments.
- Local variables.
- Code.

In C, functions can be

- part of the standard C library.
- part of an existing C library.
- user-defined.

We have used numerous functions from the standard C library (`printf()`, `scanf()`, `sqrt()`, `sin()`, `rand()`, etc...).

We have also seen one user-defined function: `in main()`.

Functions in C. In C every piece of code **must be part of a function**. Thus, C programs are **collections of functions**. The simplest C program consists of a single function.

main() as a function. Function `main()` (in `main()` in ANSI C) is a **user-defined C function** with special meaning. C compilers consider this function to be **the program**:

- Any C program is compiled to start its execution at the beginning of the `main()` function.
- The execution of any C program stops when either the control reaches the end of `main()` (non-ANSI C) or a `return` statement found in the body of `main()` is executed (both ANSI and non-ANSI C).

User-defined functions: declarations and definitions

Function declaration. A statement in a C program that tells the compiler that the program will contain a specific function. A **function declration** in C provides:

- name,
- input parameters,
- return type

of the function, but **does not specify its code**.

Function definition. A part of a C program that contains all code for a specific function. **Function definitions** in C specify all of the following:

- name,
- input parameters,
- return type,
- local variables,
- code

Function declarations and definitions: Syntax

Function delcarations . General syntax is:

```
<ReturnType> <FunctionName>([<Arguments>]);
```

Here,

`<ReturnType>` is the return type of the function;
`<FunctionName>` is the name of the function;
`<Arguments>` is the *optional* list of arguments.

(square brackets around `<Arguments>` mean "optional")

Function Names follow the same rules as variables: they must be a valid C identifier.

Return type is any C type, e.g., `int`, `char` or `double`.

An additional type, `void` represents a function with no output value.

Function arguments. `<Arguments>` is a comma-separated list of `Type Name` pairs (e.g., `int i`, `double radius`, etc...).

Examples of function declarations.

- Void, no arguments:

```
void printMessage();
void initialize();
```

- Void, arguments:

```
void printMessage(int stateDecision);
void printMax(int x, int y);
```

- non-void, no arguments:

```
float getNumber();
int getDaysLeft();
```

- non-void, arguments:

```
int findMax(int x, int y);
char convert2Char(int x);
float average(float x, float y, float z);
```

Function definitions syntax. A function definition is a **complete description of a function**. Each user-defined function in a C program must be defined. The syntax is as follows:

```
<ReturnType> <FunctionName>([<Arguments>]) {
    <Declarations>

    <Statements>
}
```

The first line of a function definition repeats the function declaration of the function. But function definitions use code blocks (enclosed in "{", "}") while declarations do not.

The <Declarations> and the <Statements> jointly are called the **body** of a function.

Return type and return values

Return type of a function. Functions represent **computations**. To ensure that the results of these computations can be used in the rest of the program, functions **return values**.

Each function may return values of a single type, called the **return type** of the function.

In C functions that do not return any values have **return type void**.

The **return value** of the function is supplied by the

```
return <Expression>;
```

statements found in the body of the function.

Note: A function may contain multiple **return** statements, **but only one of them** will be executed each time the function is executed.

Examples. Here are a couple of examples of functions that take no parameters but return values:

```
float getNumber() {
    float in;                /* declare a local variable */

    printf("Enter a number:"); /* print a prompt */
    scanf("%f",&in);          /* read a value from keyboard */
    return in;               /* return it */
}

int getDaysLeft() {
    return 30 - localtime(time(0)).tm_mday; /* how many days are left till the end of the month? */
}

#define PI 3.1415926
float sqrtPi() {
    return sqrt(PI); /* return square root of PI */
}

int randomColorIntensity() {
    int i;
    i = rand();          /* get a random number */
    i = i % 2;          /* turn it into 0 or 1 */
    if (i) {
        return 255;     /* if i is 1, return high intensity */
    }
    else {
        return 0;       /* if i is 0, return low intensity */
    }
}
```

Here are some examples of void functions.

```
void printMessage() {
    printf("Hello! My name is Inigo Montoya!\n");
    printf("You killed my father.\n");
    printf("Prepare to die.");

    return;
}

void playGame() {

    int x, y;
    x = 0;
    do {
        printf("Pick a number (1 -- 10):"); /* read a number from 1 to 10 */
        scanf("%d",&x);
    } while (x < 1 || x > 10);

    y = rand() % 10 + 1; /* randomly generate a number from 1 to 10 */

    if (x == y) { /* numbers equal: user won */
        printf("You WIN!");
    }
    else { /* numbers not equal: user lost */
        printf("You LOSE!");
    }

    return;
}
```

Parameters (Arguments)

In C many functions behave in a way similar to **mathematical functions**: they compute something for a given value or a set of values. E.g., `sin()`, `cos()`, `sqrt()` compute the sine, cosine and the square root of a given floating point number.

The values presented to the function to perform computations with are called **function arguments** or **parameters**.

Function declarations and **function definitions** must specify **all** parameters for a function.

A simple parameter declaration is

```
<Type> <Name>
```

For example, consider a function `int findMax()`. We want to design it to pick the larger of the two values and return it. Thus, this function will have *two input parameters*:

- Both parameters must be `int` values (because the return type of the function is `int`).
- **We can give these parameters ANY names.**

Thus, the function declaration is (as described above):

```
int findMax(int x, int y);
```

The following declaration is equivalent:

```
int findMax(int first, int second);
```

Use of arguments in functions

Function argument names defined as shown above can be used in the functions as **variables**. They can be used in comparisons, arithmetic and other expressions (computations), **and** they can be assigned new values.

However, the new values assigned are only valid for the duration of the code of the function.

Examples

```
int findMax(int x, int y) {
    if (x >= y) {return x;} /* compare two integers */
    else      {return y;} /* return the larger of the two */
}

float average(float x, float y, float z) {
    return (x+y+z)/3; /* compute the average of three numbers */
}
```

Local Variables

In addition to **input parameters** each function may **declare any variables** that it needs in order to perform proper computations.

Local variable. In C, any variable declared in the body of a function is called a **local variable**. It is **local** to the function in which it has been declared.

Variable scope. In a programming language, **scope of a variable** is the part of the program (i.e., the set of statements) in which a given variable *can* be used.

Variable scope determines the **lifetime** of a variable in the program. From a **syntax** perspective, **variable scope** identifies in which lines of code you can include the variable, and in which — you may not.

In C, the **scope of each local variable** is the **body** of the function in which it is declared.

Examples. Consider the following two functions defined in the same program:

```
int findMax(int x, int y) {

    int r;                /* this is a modification of the findMax() function above */
    if (x > y) {r = x;}    /* this version uses a local variable r                */
    else {r = y;}         /* to store the larger of the two numbers                */

    return r;
}

int findMin(int x, int y) {
    int r;
    if (x > y) {r = y;}    /* this is a similar function returning the smaller of two numbers */
    else {r = x;}

    return r;
}
```

Both functions use the **same names** for the input parameters (x,y) and for the local variable (r) used to store result.

We are allowed to create variables with the same name in different functions because of the scope rules for local variables in C programs!

Function Calls

A **function declaration** is simply an indication to the C compiler that the user wants to create a new function.

A **function definition** is the code of the new function.

A **function call** is the means of making the code of functions **execute**.

A **function call** is an expression or statement with the following syntax:

```
<FunctionName>(<Arg1>,<Arg2>,...,<ArgK>)
```

where <FunctionName> is the name of the function, and <Arg1>, ... <ArgK> are C

Putting it all together

A **C program** is a collection of preprocessor directives, function declarations and function definitions.

If a C program is stored in a single file (like we do so far), use the following organization:

```
/* Comments */

<Preprocessor Directives>

<Function Declarations>

int main() {
  <Declarations>

  <Statements>
}

<Function Definitions>
```

<Function Declarations> block shall contain declarations of **all** user-defined functions in the program except for `int main()`.

Example. Here is a simple example of a program that uses multiple functions.

```
/* CPE 101 program with functions */
#include <stdio.h>

int findMax(int x, int y); /* findMax(): returns the max of two numbers */
int findMin(int x, int y); /* findMin(): returns the min of two numbers */
void printMessage();      /* printMessage(): outputs a greeting      */
void announce(int x);     /* announce(): announces a winning number */

int main() {

  int a,b,c,d;
  int x, y;

  printMessage();
  scanf("%d%d%d%d", &a,&b,&c,&d);
  x = findMax(a,b);
  x = findMax(x,c);
  x = findMax(x,d);

  y = findMin(a,b);
  y = findMin(y,c);
  y = findMin(y,d);

  if ((x - y) > 10) {
    announce(x);
  }
  else {
    announce(y);
  }
  return 0;
}

int findMax(int x, int y) {

  int r;
  if (x > y) {r = x;} /* this is a modification of the findMax() function above */
  else {r = y;}      /* this version uses a local variable r */
                    /* to store the larger of the two numbers */
}
```

```

    return r;
}

int findMin(int x, int y) {
    int r;
    if (x > y) {r = y;}    /* this is a similar function returning the smaller of two numbers */
    else    {r = x;}

    return r;
}

void printMessage() {    /* printMessage(): outputs a greeting    */
    printf("Welcome to the battle of the numbers!\n");
    printf("Please enter four integer numbers:");
    return;
}

void announce(int x){    /* announce(): announces a winning number    */
    printf(".... And the winner is ....\n");
    printf(".....\n");
    printf(".....%d !!!!\n",x);
}

```