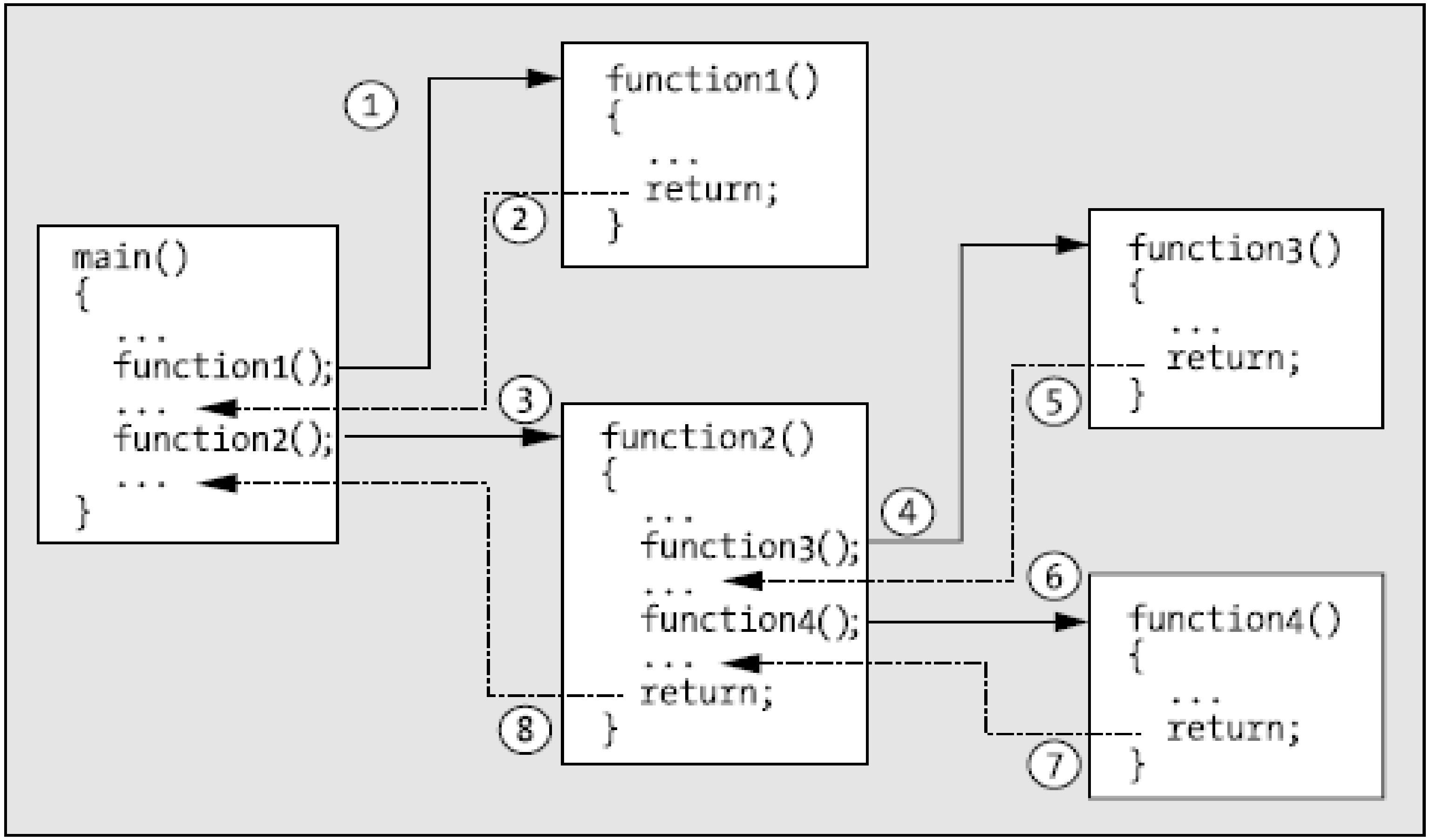
Functions (Subprograms) in C

- A subprogram (in C language is named function) is a self-contained unit of a program that evaluates a particular set of operations. For evaluation a function should be called from same place of the main( ) function or another function of the program. Only main( ) function is called by operating system.
- When a function is called, the code within the body of that function is executed, and when the function has finished executing, control returns to the point at which that function was called.

Execution of a program made up of several functions



Explanation for the previous scheme

- The program steps through the statements in sequence in the normal way until it comes across a call to a particular function. At that point, execution moves to the start of that function—that is, the first statement in the body of the function. Execution of the program continues through the function statements until it hits a return statement or reaches the closing brace marking the end of the function body. This signals that execution should go back to the point immediately after where the function was originally called.

Function fundamentals

When you create a function, you need to specify the **function header** as the first line of the function definition, followed by the executable code for the function enclosed between braces. The block of code between braces following the function header is called the **function body**.

- The function header defines the type for the value that the function returns (is shorter called type of the function), name of the function, the function parameters (in other words, what types and values can be passed to the function when it's called).
- The function body determines what operations the function performs on the values that are passed to it.

Function fundamentals

When using functions we make a distinction between a function **declaration** and a function **definition**.

- A **function declaration** is a statement that defines the essential characteristics of a function. It defines its name, its return value type, and the type of each of its parameters. A function declaration is also called a **function prototype**, because it provides all the external specifications for the function. A function prototype enables the compiler to generate the appropriate instructions at each point where you use the function and to check that you use it correctly in each case. When you include a header file in a program, the compiler adds the function prototypes for library functions to the program.
- A function declaration (a function prototype) looks like this:

return_type function_name(list of types and names of parameters separated by commas);

(note the semicolon at the end of a function declaration)

Function fundamentals

When you **define** a function, you need to specify the **function header** as the first line of the function **definition**, followed by the executable code for the function enclosed between braces. The block of code between braces following the function header is called the **function body**.

- The function header defines the name of the function, the function parameters (what types of values and there names passed to the function when it's called), and the type for the value that the function returns.
- The function body determines what calculations the function performs on the values that are passed to it.

The general form of a function **definition** is:

return_type function_name(list of types and names of or parameters separated by commas)

{ body of the function;}

Function parameters

- Function parameters are defined within the function header round breckets and are placeholders for the arguments that need to be specified when the function is called. The parameters for a function are a list of parameter names and their types, and successive parameters are separated by commas. The entire list of parameters is enclosed between the parentheses that follow the function name.

Function parameters

- Parameters provide the means by which you pass information *from* the calling function *into* the function that is called. These parameter names are local to the function, and the values that are assigned to them when the function is called are referred to as arguments. The computation in the body of the function is then written using these parameter names, which will have the values of the arguments when the function executes.

Interchanging data between functions - sending data

- Function call within the program is referenced by its name.
- If some transfer information is needed to be passed, as it usually happen, this information is passed as arguments to a function, by including these arguments between parentheses following the function name.

Interchanging data between functions – receiving data

- The information can be received in 2 ways:
- By function arguments in parentheses. You provide an address of a variable through an argument to a function, and the function places a value in that address.
- By returning value (only one).

Function parameters – passing mechanisms

1) **Pass by value mechanism**: by passing the variables to the function, the function actually operate on **copies** of these variables. This means that the variables passed themselves are not affected by the function to which they are passed. The only way to modify the argument passed to a function is to pass it by address.

Function parameters – passing mechanisms

Example:

```
void Factorial (unsigned int a)
{
  unsigned int i;
  for (i=a; i>=1; i--)
  a=a*i;
  return ;
}
```

In this case the value of '**a**' is not modified, although we apparently assign to '**a**' new values.

Function parameters – passing mechanisms

2) **Passing by address** mechanism: passing the address of a variable we still pass a copy of that address to the function. The address itself is not modified by the function. The point is that the address (pointer) passed points to the location of the variable. By **dereferencing** the name of this parameter inside the function body, we can modify the contents of the variable, so that they are seen outside the function body.

Function parameters – passing mechanisms

Example:

```
void Factorial (unsigned int* pa)
{
  unsigned int i;
  for (i=*pa; i>=1; i--)
 *pa = *pa *i;
  return ;
}
```

Function parameters – passing mechanisms

In this case the value of the variable pointed to by the pointer **pa** is modified.

If **pa** points to variable **'a'** defined inside the calling function, **'a'** , after the call to this function, will be equal to the factorial of its previous value (provided no overflow occurred during multiplication).

Note the dereferencing mechanism used (preceding the pointer variable by the unary operator '*').

Function basics- return type

The return type specifies the type of the value returned by the function. If the function is used in an expression or as the right side of an assignment statement, the return value supplied by the function will effectively be substituted for the function in its position. The type of value to be returned by a function can be specified as any of the legal types in C, including pointers. The type can also be specified as void, meaning that no value is returned. A function with a void return type can’t be used in an expression or anywhere in an assignment statement.

Function basics- function variables

Variables used in a function can be of 3 types:

1. Local variables – these variables are declared within the function body and can be referenced only during function execution. They are automatically destroyed as the function execution ends.
2. Static variables – a variable declared as static within a function is similar to local variables in the meaning that they also can be accessed only during function execution. The difference is that they are not destroyed as the function execution ends. Instead their value is preserved during function calls. If a variable is initialized when it is defined the initialization process will happen only at the first function call and will not take place in subsequent function calls.
3. Global variables – these are variables that can be used anywhere in the program. The are defined outside of any function body.