# Chapter 11. Functions with a variable number of parameters. Overloaded functions. Prototypes and feature templates

## 11.1 Variable Number Functions

If the list of formal parameters of a function ends with an ellipsis, it means that when the function is called, you can specify a few more parameters in that place. Functions with a variable number of parameters allow you to pass an arbitrary number of arguments when calling a function. The structure of declaring a function with a variable number of parameters is shown in figure 11. 1.

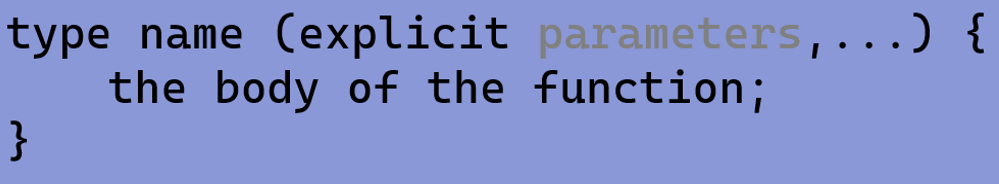


Figure 11.1 – Structure of the declaration and definition of a function with a variable number of parameters

To access optional parameters within the function, the library macros located in the header file *<сstdarg>*or *<stdarg.h> are used*.

To work with variable parameters, you use the *va\_list* type, which provides a mechanism for accessing variable-length arguments.

1. The va\_start macro is used to initialize *va\_list* before you start reading arguments.
2. The va\_arg macro is used to read arguments from *va\_list*. The type of the argument is determined by the type passed by the second argument.
3. The macro *va\_end* exits the *va\_list* when you finish reading the arguments.

An example of using a function with a variable number of parameters is shown in figure 11.2.

The main part of the code is to define the *average* function, which takes as its first argument a number of numbers followed by an arbitrary number of integer arguments [12]. Inside the function, *a variable argument mechanism* (va\_list, va\_start, va\_arg, va\_end) is used to process an arbitrary number of arguments and calculate their sum .

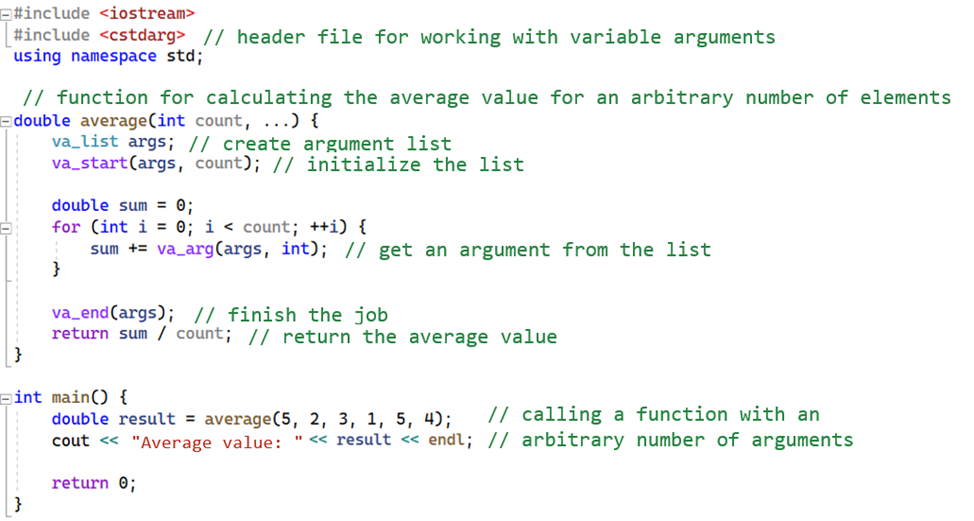


Figure 11.2 – Example of using a variable number of parameters function

## 11.2 Overloaded functions

The use of multiple functions with the same name but with different types of parameters is called function overloading. The compiler selects the most appropriate overloaded function based on the arguments passed. Figure 11.3 defines 3 functions with the same name, but with different types of parameters and return value.

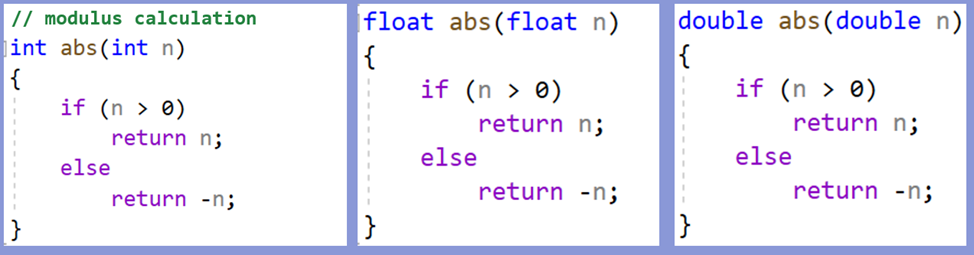


Figure 11.3 – Example of using overloaded functions

Rules for describing overloaded functions:

1. Overloaded functions must be in the same scope.
2. Functions are overloaded when the input and output parameter data types are different.
3. Functions cannot be overloaded if the description of their parameters differs only in the const modifier or the use of a reference (for example, int and const int or int and int&).

## 11.3 Function templates

Templates are programming language constructs that allow you to create universal code that can work with different types of data. They allow you to define a common logic or algorithm of work.  Function templates are introduced to automate the process of creating functions that can work with different types of data. Templates allow you to create families of functions, which provides a single logic (algorithm) when working with different types of data. The type of data that the template works with is passed as a parameter.

Template description rules:

1. The description of any template begins with the template keyword.

2. Template parameters (arguments) are listed in angle brackets <... >.

3. Template parameters denote data types, so each parameter must be preceded by a *typename* or *class keyword.*

4. There can be several template parameters.

Figure 11.4 shows an example of a boilerplate function that takes two parameters with different data types and outputs their size in bytes to the console.

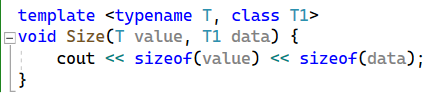


Figure 11.4 – Example of a template function

Figure 11.5 shows that instead of 3 overloaded functions that compute a modulus of values of different data types, you can use a single boilerplate function, which greatly simplifies the code [26].

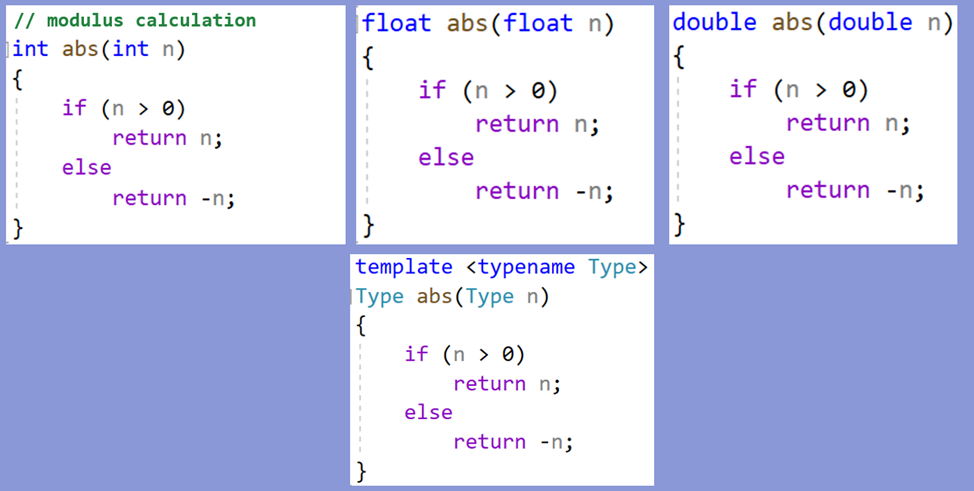


Figure 11.5 – Applying a template function

The main properties of the function template parameters are:

1. There can be several parameters in the list of template parameters, each preceded by the typename or class keyword. When you call a template function, you must make sure that the actual parameter types match the formal parameters.

In Figure 11.6, the first example shows a template function called "maximum" that takes two arguments, one of type "T" and one of type "T1". The function compares the values of the arguments and outputs the maximum value. There is no error here, because both the actual parameters – "a" and "b" – are of different types that correspond to different parameters of the template. That is, when creating a function from a template, the int type is set instead of the T parameter, and the float type is set instead of the T1 parameter.

The second example implements the same function, but takes two arguments of the same type "T". This is where an error occurs. When the function is called, two parameters of different types are passed: "a" is of type "int", and "b" is of type "float". The actual parameters do not match the formal parameters of the template, which leads to a compilation failure because it is not clear which type should be set instead of T.

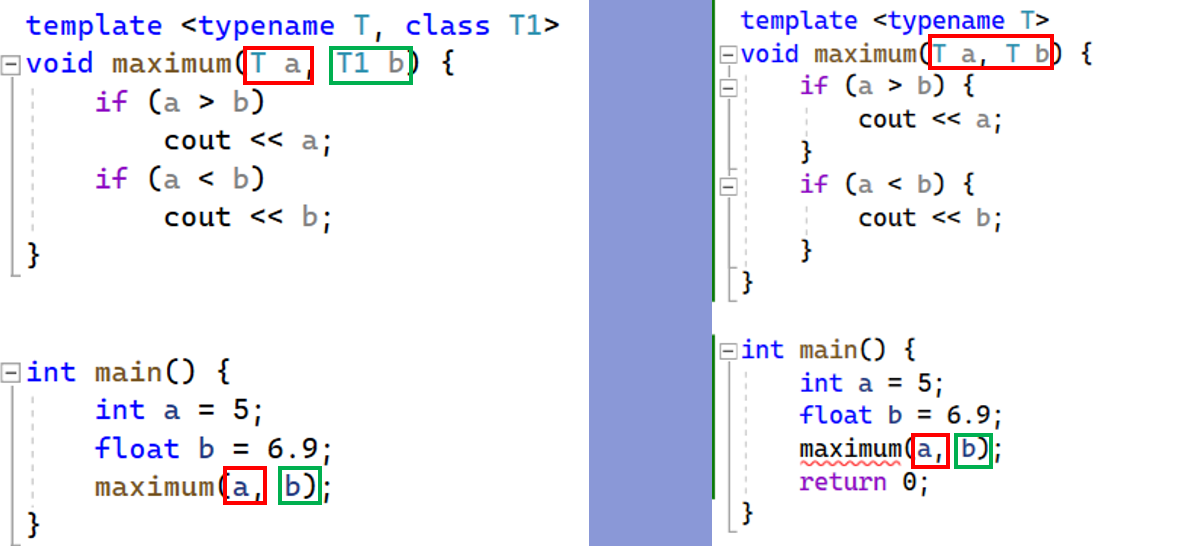


Figure 11.6 – Example of correct and incorrect use of a template function

1. The name of a template parameter has all the rights of the type name in the template-defined function. This means that:

The template parameter name can be used as the type name to define function parameters;

The name of the template parameter can be used to determine the type of value that the function returns;

The name of a template parameter can be used to define variables, constants, or other objects that are localized in the body of the function.

In the example in figure 11.7, the function defines formal parameters "a" and "b" with type "T", which is the name of the template parameter. A variable "c" of type "T" is declared inside the function [31]. Then the values "a" and "b" are compared and the variable "c" is assigned the larger of them. The return value of the function is also of type "T", which is determined by the template parameter. The function returns the value of the variable "c", which is the maximum of the two arguments.

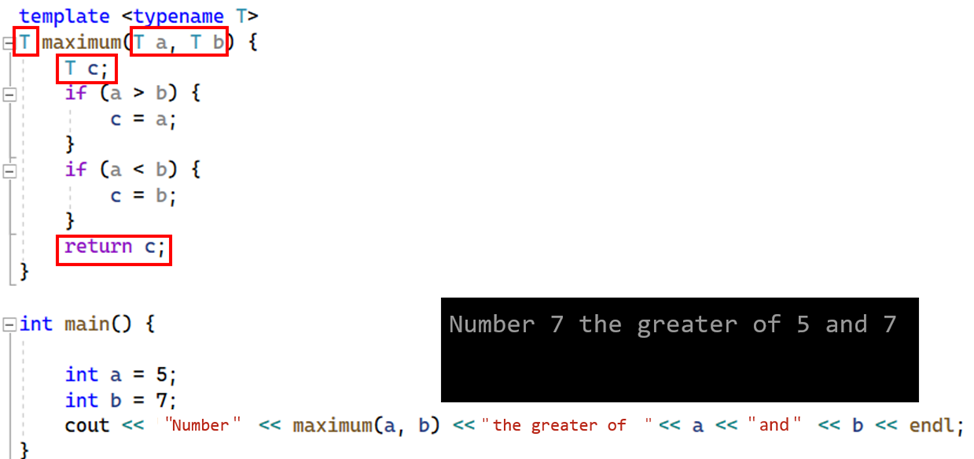


Figure 11.7 – Example of working with a template function

1. A parameterized function can have as many non-parameterized formal parameters as you like. In the example in figure 11.8, the template function "function4" takes three formal parameters: "a" of type "T1", "b" of type "int", and "c" of type "bool". Note that the parameter types "b" and "c" are not template parameters, they have standard data types.



Figure 11.8 – Example of standard parameters in a template function

1. Template parameter names must not be the same as function parameter names. In the example in figure 11.9, the template function takes two parameters of type "T". However, there is a problem here: the name "T" has both a template parameter and a function parameter. This leads to confusion and errors in the code. To avoid such situations, it is important to choose different names for the template parameters and formal parameters of the function.



Figure 11.9 – Example of incorrect use of parameters in a template function

When the compiler sees the template keyword and the function definition that follows it, the compiler "remembers" the template for future use. This means that the compiler recognizes that the function is template and needs to be handled differently.

Code generation does not occur until the function is actually called during the execution of the program. The compiler does not generate specialized code for each possible data type in advance, but deferres this task until the function with specific data types is actually called.

The first time you call a function with specific parameter types, the compiler will build a function with parameters of that type. For example, if there is a template function with a parameter of type T, and it is called for parameters of type int, the compiler will create a specialized function where it substitutes an integer data type instead of T. This process is called template instantiation.

## 11.4 Function Prototype

Prototypes allow you to declare functions before defining them. This simplifies the organization of the code and contributes to the readability of the program. Typically, prototypes are placed at the beginning of a program or in header files (.h). In the example in figure 11.10, you can see the prototype of the "summ" function, which outputs the result of the sum of two integer variables, followed by a function call, and only then the definition, and the compiler does not throw an error. If a function is defined after it is called, and there is no prototype, then the program would not run. It is also worth noting that it is not necessary to specify parameter names in the prototype, you can specify only types, but the sequence of types must coincide with the sequence of types in the definition.

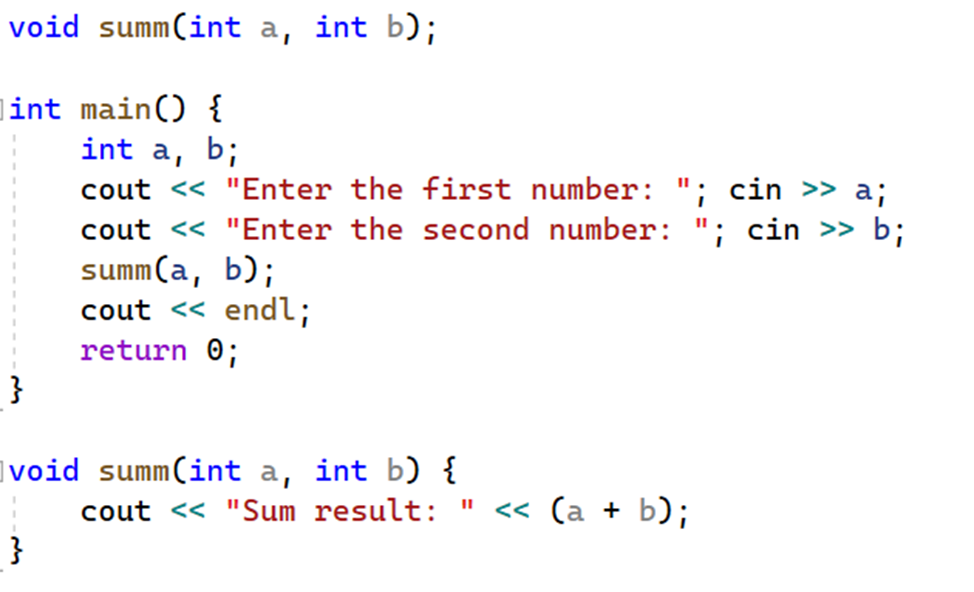


Figure 11.10 – Example of using a function prototype