Functions

Let’s introduce **functions** in programming, defining them as blocks of code designed to perform a specific task that might need to be repeated multiple times within a program. Using functions helps avoid **redundancy** (unnecessary repetition of code) and improves code **readability** and organisation3.... The `main` function is highlighted as the starting point of program execution and is itself a simple function.

A function is described as being similar to a black box that **takes some input, performs some work, and potentially returns some output**.
•A function definition specifies its **return type** (the type of value it will return, e.g., `int` for an integer, or `void` if it returns nothing), its **name**, parentheses `()` for parameters, and curly braces `{}` containing the **block of code** that defines the function's task.
•The code inside the curly braces constitutes the function's logic or **definition**.
•To execute the function's code, it must be **called** or **invoked** by writing its name followed by parentheses and a semicolon. A function defined but never called will not execute.
•The `return` keyword is used to send a value back from the function to the calling code. For `void` functions, an empty `return` statement can be used to simply return control. Any code written after a `return` statement inside a function will not be executed.

Functions can accept inputs through **parameters**, which are variables defined within the function's parentheses. When calling a function, the actual values passed into these parameters are called **arguments**. Arguments are the actual values (like numbers or variables) passed to the function, while parameters are essentially copies of these arguments within the function's scope. **Lerals** are technical terms for constant values like `1`, `10`, `15`, or characters, which remain unchanged.

Regarding how functions work in computer memory (specifically in the context of C++), two main types of memory are mentioned: **Stack** and **Heap**.
•Functions are generally stored in **Stack memory**.
•Each time a function is called, a **Stack Frame** or activation record is created on the Stack for that function.
•This Stack Frame stores the function's logic and all its related local variables.
•The current function being executed is always at the **top of the Stack**.
•When a function finishes execution (either by completing its code or hitting a `return` statement), its Stack Frame is removed from memory, and its local variables are no longer accessible. This explains why variables defined within one function cannot be directly accessed from another.

Pass by Value
•When primitive data types like `int`, `float`, `double`, `char`, or `bool` are passed as arguments to a function, a **copy** of the argument's value is created for the corresponding parameter in the function's Stack Frame.
•Therefore, any changes made to the parameter inside the function **do not affect the original variable** in the calling function.
•This is demonstrated with an example where changing a parameter `x` inside a function `changeX` does not alter the value of the original variable `x` defined in the `main` function after the function call.

Pass by Reference is briefly mentioned as an alternative method where the original entity (like its memory address) is passed, allowing changes within the function to affect the original variable.