<u>Subprograms Summary</u>

- **Subprogram** definition:
    - Subprogram header
        - Kind
            - Function
            - Procedure (no return type)
        - Name (identifier)
        - Protocol
            - Parameter profile
                - Number
                - Order
                - Type
            - Return type
    - Subprogram body: specifies a sequence of statement which are executed in order.
- Subprogram call is the explicit request to execute its body.
- Formal parameters = parameters (called subprogram).
    - double Fun(int a, double b) {}
- Actual parameters = arguments (calling subprogram).
    - double x = Fun(9,7.2);
- **Control** is passed to the entry point (usually the first statement) of the called subprogram, then returns to the caller when the called subprogram execution terminates.
- **Positional parameters:** The correspondence between <u>actual</u> and <u>formal</u> parameters is usually done by position.
- **Keyword parameters:** Specifies the name of the <u>formal</u> parameter to be bound with the <u>actual</u> parameter. (readability$\uparrow$)
    - So, the programmer does not need to remember the order of <u>formal</u> parameters.
- **Default values:** Can be associated to <u>formal</u> parameters which are used whenever the subprogram call does not specify the corresponding actual parameters.
- **params** modifier, where the caller sends either array or list of expressions (C#).
- **Subprogram declaration** (called prototype in C++)**:** provides the subprogram <u>header</u> but does not include its <u>body</u>.
- C and C++ require all subprograms to be either <u>defined</u> or <u>declared</u> before they are called and inside the same translation unit where they are called (to perform static type checking).
- Source file ----- preprocessor-----> translation unit.
    - expanding all macros (instructions starting with #), usually by simple text substitution.

- Translation unit ----- compiler-----> object file.
    - The compiler requires that all functions be either defined or declared before its first call in the same translation unit , to perform **static type checking** to validate **type compatibility** between actual and formal parameters.
- Object files ----- linker-----> executable image (load module) , which is a machine-readable executable file or library.
    - Linker is responsible for resolving all function calls by calculating the correct addresses and placing them into the corresponding call statements inside the executable. To be able to do so, the definition of each function must exist exactly in one translation unit.
- The above compilation and linking mechanisms **reduce compilation time** by avoiding recompilation of source files which are not changed, including source files for built-in C++ libraries. Only the changed source files of a project need to be recompiled.
- In C++, there is no restriction on the number of **function declarations**.
- Each **non-inline function** must be defined **exactly once** across all files.
- **Classes** and **inline functions** must be defined **at most once per translation unit**, such that at least one definition exists for each entity across all files, and all definitions for the same entity are identical.
- **Inline functions**: the compiler tries to replace calls to inline functions by the code of the function body itself, which may be useful for optimization only if the number of statements in the body is small.
- It is just declared (not defined) in the new file using the **extern** modifier before accessing it, because each variable must be defined exactly once across all files (global variables).
- **Static class data members** are considered declared but not defined if their declarations appear inside their class definitions. This is because classes can be defined several times in different translation units, but variables cannot.
- **Formal parameters:**
    - **In mode**: Formal parameters receive data from the corresponding actual parameters.
        - Pass by value: The value of actual parameter is used to initialize the corresponding formal parameter by copying.
        - Pass by read-only reference: Provides read-only access path to the actual parameter.
    - **Out mode**: Formal parameters transmit data to the corresponding actual parameters (In C# not C++).
        - Pass by result: No value is transmitted to the formal parameter, which acts as local variable whose value is transmitted back to the actual parameter by copying just before control is transferred back to the caller.
    - **In-out mode**: Formal parameters receive data from and transmit data to the corresponding actual parameters.
        - Pass by value result.

        - Pass by reference: Provides access path to the <u>actual parameter</u>.
        - Pass by name: The <u>actual parameter</u> is textually substituted for the corresponding <u>formal parameter</u> (compile-time only by C++).
- **<u>Implementing subprogram calls:</u>**
    - Each subprogram has the following simplified typical activation record:
        - Return variable
        - Local variables
        - Parameter variables
        - Return address
    - Each call to function starts by pushing to the <u>run-time stack</u> an <u>activation record instance</u> of the activation record of the function.
    - The run-time stack of a specific program is part of the <u>main memory</u> assigned by the <u>operating system</u> to this program and can be used to allocate its <u>stack-dynamic variables</u>.
    - <u>Activation record instance (ARI):</u> is a specific instance of the activation record with specific allocated variables.
    - The <u>return address</u> is the address of the instruction that follows the function call instruction.
    - Since the size of an <u>activation record instance</u> for a specific function is known before the function call (it is known at <u>compile time</u>), only one memory allocation is needed to allocate the whole activation record instance which is efficient.
    - While executing a specific function call, its associated set of variables always exist at the activation record instance at the <u>top</u> of the stack.
    - If the subprogram contains inner blocks, the compiler chooses one of the following two ways to implement its calls:
        - Each inner block is treated as a call to a subprogram with no parameters. In this case, the activation record of the subprogram does not contain any variable local to an inner block. Each inner block has its own activation record.
        - The activation record of the subprograms contains local variables which are not local to inner blocks, and it contains space sufficient to hold the maximum amount of storage for inner block variables at any time during the subprogram execution.
- **<u>Simulating recursion:</u>**
    - Simulating without recursion because of one of the following reasons:
        - Decrease the usage of the run-time stack assigned by the operating system.
        - Run the program on embedded system environment which does not support recursion.
        - Use a programming language which does not support recursion.
        - Need to avoid using the run-time stack to track memory usage in limited-memory environment. Make a compiler simulation.