* **Function Declarations –**
* A function declaration may contain argument names.
* But unless the declaration is also a function definition, the compiler ignores these names.
* **Why Functions?**
* The most basic advice is to keep a function of a size so that one can look at it in total on a screen.
* Many programmers put a limit of about 40 lines in a function.
* But for the founder of C++, the average size is 7 lines.
* **Parts of a function declaration –**
* The *name* of the function. Required.
* The *argument list* which may be empty. Required.
* The *return type*, which may be void and may be a prefix or suffix (by using auto). Required.
* *inline*, indicating a desire to have function calls implemented by inlining the function body.
* *constexpr*, indicating that it should be possible to evaluate the function at compile time if given constant expressions as arguments.
* *noexcept*, indicating that the function may throw an exception.
* A *linkage specification*, for example, *static*.
* *[[noreturn]]*, indicating that the function will not return using the normal call/ return mechanism.
* **Parts of a member function can additionally be specified as –**
* *virtual*, indicating that it may be overridden in a derived class.
* *override*, indicating that it must be overriding a virtual function from a base class.
* *final*, indicating that it cannot be overridden in a derived class.
* *static*, indicating that it is not associated with a particular object.
* *const*, indicating that it may not modify its object.
* **Function definitions –**
* A function can be defined in two ways –



or



* Naming arguments in declarations that are not definitions is optional and commonly used to simplify documentation.
* **Things other than functions that we can call –**
* *Constructors:* Technically not functions. They don’t return a value. Can initialise bases and members. Can’t have their address taken.
* *Destructors:* Can’t be overloaded. Can’t have their address taken.
* *Function Objects:* Not functions, but objects. Can’t be overloaded. Their ***operator()***s are functions.
* *Lambda Expressions:* Shorthand for defining function objects.
* **Returning Values –**
* A value must be returned from a function that is not declared *void*.
* Conversely, a value can’t be returned from a *void* function.
* A function that calls itself is said to be recursive.
* A return statement is one of the following ways of returning a statement –
* Executing a return statement.
* Simply reaching the end of a *void* function body.
* Simply reaching the end of *main()* function body.
* Throwing an exception that is not caught locally.
* Terminating because an exception was thrown and not caught locally in a *noexcept* function.
* Directly or indirectly invoking a system function that does not return.
* A function is marked as *[[noreturn]]* if it is not returned normally, i.e. through a *return* or simply reaching the end.
* What happens when the function returns inspite of a *[[noreturn]]* attribute is undefined.
* **Local Variables –**
* A local variable or constant is initialised when the thread of execution reaches its definition.
* A *static* local variable allows the function to preserve information between calls without introducing a global variable that might be accessed and corrupted by other functions.
* The effect of initialising a local *static* is undefined.
* **Reference Arguments:** Rules of thumb to choose among the way of passing arguments –
* Use pass-by-value for small objects.
* Use pass-by-const-reference to pass large values that don’t need to modify.
* Return a result as a return value rather than modifying an object through argument.
* Use rvalue references to implement move and forwarding.
* Pass a pointer if “no object” is a valid alternative. Represent “no object" by nullptr.
* Pass use-by-reference only if you have to.