* **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.
* **Unspecified Number of Arguments:** To implement such interfaces, we have –
* Use a variadic template. This allows us to handle an arbitrary number of arbitrary types in a type-safe manner by writing a small template metaprogram that interprets the argument list to determine its meaning and take appropriate actions.
* Use an initialiser list as the argument type. This allows us to handle an arbitrary number of arguments of a single type in a type-safe manner.
* Terminate argument list with ellipses (…), which means “and may be some more arguments”. This allows us to handle an arbitrary number of almost arbitrary types by using some macros from <cstdarg>. This solution is not inherently type-safe.
* Only when the number of arguments and the type of arguments vary and a variadic template solution is deemed undesirable is the ellipses necessary.
* **Automatic Overload Resolution:**
* If we have separate print functions (with slightly different names for each) for each of the data types, we will have to remember several names and remember to use them correctly.
* This can be tedious, error-prone and defeats the attempt to do generic-programming.
* It also encourages the programmer to focus on relatively low-level type issues.
* **Pre- and Post-Conditions:**
* The writer of a function has several alternatives –
* Make sure that every input has a valid result, so that we don’t need a pre-condition.
* Assume that the pre-condition holds, i.e. rely on the caller to make mistakes.
* Check that the precondition holds and throw an exception if it does not.
* Check that the precondition holds and terminate the program if it does not.
* If a postcondition fails, there was either an unchecked precondition or a programming error.
* **Pointer to Function:**
* Like a data object, the code generated for a function body is stored somewhere in memory and hence has an associated address.
* We can have a pointer to a function just as we have a pointer to an object.
* However, a pointer to a function does not allow the code to be modified, because of various reasons –
* Some related to machine architecture, and
* Others to system design
* There are only two things one can do to a function –
* Call it, and
* Take its address
* **Macros:**
* The first rule about macros is: don’t use them unless you have to.
* Almost every macro demonstrates a flaw in the programming language or in the programmer.
* Macros manipulate character strings and know little about C++ syntax and nothing about C++ types and scope rules.
* Only the expanded form of a macro is seen by the compiler, so an error in a macro is reported when the macro is expanded, not when it is defined.
* Old C preprocessors are used to decode macros which may not understand the // (single line comment).
* Thus, for complicated macros, it is wise to use /\* \*/ (multi-level comments).
* The *auto*, *constexpr*, *const*, *decltype*, *enum*, *inline*, *lambda* expressions, *namespace*, and *template* mechanisms can be used as better-behaved alternatives to macros.