***C++ Basics (Chapter 2)***

**Functions –**

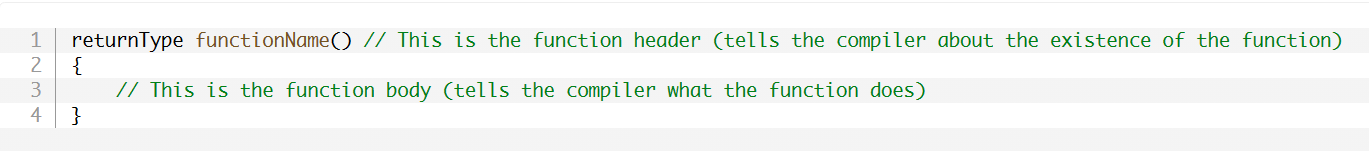
A **function**is a *reusable* sequence of statements designed to do a particular job.

Functions that you write yourself are called user-defined functions.

A **function call** is an expression that tells the CPU to interrupt the current function and execute another function to perform a task defined in the program. The *CPU* puts a ‘bookmark’ at the current point of execution, and **calls (**executes**)** the function named in the function call. When the *called* function ends, the CPU returns to the bookmarked point in the execution.

The function initiating the call is the **caller** and the function being called is the **called** (callee) function.

**User-defined function**



The first line (returnType functionName()) is called the **function header**, and it tells the compiler about the existence of a function, it’s name, and some other information (like return type and parameter type).

* The functionName is the name (identifier) of the *user-defined* function.
* The **parentheses** after the *identifier* tell the compiler we’re defining a function.

The curly braces and the statements inside are called the **function body**. This is where the statements go that determine the function’s purpose.

*Example –* Shows function *definition* and *call* –

A screenshot of a computer code

Description automatically generated

The program outputs –

A white background with black text

Description automatically generated

* The program begins execution at the top, the first line to be executed prints “Starting main()”.
* The second line calls the void function (doPrint()) defined above.

**Warning:** Don’t forget to include the parentheses () after the function’s name when making a function call.

Because a function call was made, execution of statements in *main* is suspended, and execution jumps to the top of the function doPrint(). The first (and only) line defined in doPrint prints “In doPrint()”. When doPrint terminates the program returns back to the bookmarked point in execution and prints the next line defined in *main* which prints “Ending main()”.

**Calling functions more than once –**

A screenshot of a computer program

Description automatically generated

The program produces the output –

A white background with black text

Description automatically generated

* doPrint is called twice by main, so it executes twice, and “in doPrint()” gets printed twice (once for each call).

**Functions that call other functions –**

Any function can call any other function. In the following program, function *main* calls function doA, which calls function doB.

A screenshot of a computer program

Description automatically generated

Output –

A white background with black text

Description automatically generated

**Nested functions are NOT supported –**

Unlike other programming languages, in C++, functions cannot be defined inside other functions.

The following program is not legal –

A screenshot of a computer program

Description automatically generated

A screenshot of a computer code

Description automatically generatedThe proper way to write it would be –

**Quiz.**

1. In a function definition, what are the curly braces and statements in-between called?

* The **function body**.

1. What does the following program print?

A screenshot of a computer program

Description automatically generated

The program prints –

**“Starting main()”  
“In doA()”  
“In doB()”  
“Ending main()”**

**X.**

“Starting main()”  
“In doA()”  
“In doB()” - **Read the code *carefully***  
“In doB()”  
“Ending main()”

**Function return values (value-returning functions)**

A screenshot of a computer code

Description automatically generated

The program is simple, it takes a value from a user then returns the value doubled.

The program does not need to be broken up (multiple functions) but if it were to be, how would you go about it?

*This is an attempt* (**incorrect**) –

A screenshot of a computer code

Description automatically generated

* The program doesn’t work because the value for num{} is never initialized after receiving input from user with the getValueFromUser() function.

The program needs some way for getValueFromUser to return the value back to *main* so that *main* can make user of the data.

**Return Values –**

To return a value back to the *caller,* two things are needed.

First,

* Function must indicate what type of value it will return; this is the functions **return type.**
* Function ‘getValueFromUser’ has a return type *void* (meaning no value is returned), function ‘main’ has a return value of *int*.
  + Note: This doesn’t determine what *specific* value is returned – it only determines the *type* of value that will be returned by the function.

Second,

* We use a **return statement** to indicate the specific value being returned to the caller. The specific value returned is called the **return value.** When the *return statement* is executed the function exits immediately, and the return value is copied from the function to the *caller* (main() in this case).
* This process is called ***return by value.***

*Sample program that returns an integer value to a* ***caller*** *–*

*A screenshot of a computer

Description automatically generated*

The program prints –

A white rectangular object with a black border

Description automatically generated

**Fixing the previous problem –**

**A screenshot of a computer program

Description automatically generated**

The program initializes *int num*, it sees that there’s a function call to getValueFromUser(), it executes the function which requests the user to input a number and returns it back to the *main* function. The main function then receives the **return value** from getValueFromUser which is used as the initialization value for variable *num*.

**Revisiting main() –**

When the program is executed, the operating system makes a *function call* to main. Execution then jumps to the top of main. The statements in *main* are executed sequentially. Finally, main returns an integer value (usually 0) and the program terminate. The return value from *main* is called a **status code** (also sometimes, an *exit code,* or *return code).* It is used to indicate whether a program ran successfully or not.

**Best Practice:** Your main function should return the value 0 if the program ran normally, a non-zero value will result in *undefined behavior.*

**A value-returning function that does not return a value will produce undefined behavior –**

A function that returns a value is called a **value-returning function.** A function is value-returning if the return type is anything than *void.*

A value-returning function *must* return a value of that type (using a return statement), otherwise undefined behavior will result.

*Example program with undefined behavior –*

A screenshot of a computer code

Description automatically generated

* Note: A modern compiler should generate a warning because getValueFromUserUB is defined as returning an int type but has no return statement. Running this program will result in *undefined behaviour,* because getValueFromUserUB is a value-returning function with no *return statement.*

**Best practice:** Make sure non-void functions return a value in all cases. Failure to do so will result in *undefined behaviour.*

**Function main** – will *implicitly* return value 0 if no return statement is provided. That said, it is best practice to include a return statement to show your *intent* and keep consistency with the other functions (which will result in undefined behaviour without a return value).

**Functions can only return a *single value –***

A value-returning function can only return a single value each time it is called.

Note that the value returned does not need to be literal – it can be the result of a valid expression, including a variable or even a call to another function that returns a value. In the getValueFromUser example *above,* we returned a variable ‘input’, which held the user input.

**The function author can decide what the return value means –**

The meaning of a function is determined by the function’s author. Some functions are used to return a calculated value, some use return values as status codes (failure or success), other functions return nothing (void functions).

* Because of the wide variety of possibilities, it’s *best practice* to indicate what the return values mean:

A screenshot of a computer code

Description automatically generated

**Reusing functions –**

*Consider the following program –*

A screenshot of a computer code

Description automatically generated

While the program works, it’s a little redundant (repeats itself which is not good practice).

**DRY =** “Don’t Repeat Yourself’.

The program *improved –*

A computer screen with text on it

Description automatically generated

Outputs:

A black background with white text

Description automatically generated

In the program we call getValueFromUser twice, to initialize variables x and y. This saves us from duplicating code and reduces the odds of making a mistake.

This is the essence of *modular programming*; the ability to write a function, test it, ensure that it works, and then know that we can reuse it as many times as we want, and it will continue to work.

**Best Practice:** Follow DRY: “Don’t Repeat Yourself”. If you need to do something more than once, consider how to modify the code to remove as much redundancy as possible (the possibility of mistakes). Like all best practice, DRY is meant to be a *guideline,* not an absolute. DRY can harm overall comprehension when code is broken into pieces that are too small.

**Quiz.**

Inspect each of the following programs. Determine what the program will output, or whether the program will generate a compilation error.

1. A screenshot of a computer code

   Description automatically generated

The program will output: “**16”.**

1. A screenshot of a computer code

   Description automatically generated

This program will generate a compilation error because a function cannot be defined inside another function. (**Nested** **functions).**

1. A screenshot of a computer code

   Description automatically generated

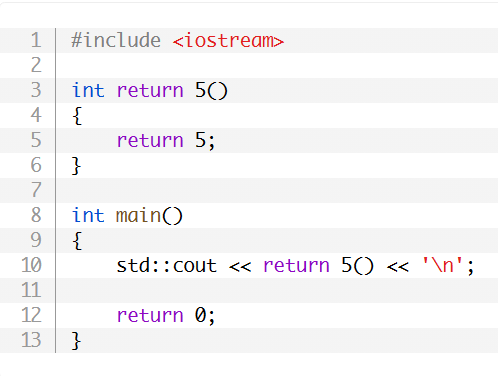
The program will compile correctly but there will be no output.

1. A screenshot of a computer code

   Description automatically generated

Incorrect, there can only be one return statement per function. Because:

(The program prints 5 twice. Both times when getNumbers is called, the value 5 is returned, when the return 5 statement is executed, the function immediately exits, so the return 7 statement is never executed.)

1. 

The function call is incorrect (X. The function has an invalid name. **READ THE QUESTION).**

1. A screenshot of a computer code

   Description automatically generated

The program doesn’t compile because the function call is incorrect (missing parentheses).

1. What does “DRY” stand for, and why is it a useful practice to follow?

DRY stands for “Don’t Repeat Yourself”. It is a practice that involves writing your code so that it reduces redundancy. This makes the program more *concise, less error prone, and more maintainable.*

**Void Functions (***non-value returning functions) –*

*Void return values:* Functions are not required to return a value back to the caller. To define a function that does not return a value, the return type **void** is used. For example:

A screenshot of a computer code

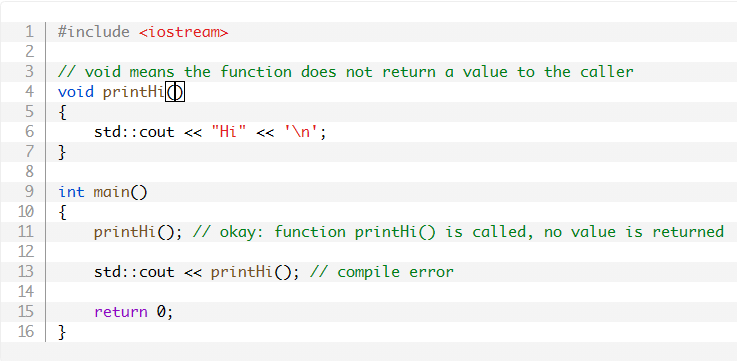
Description automatically generated

In the example provided, printHi is called in *main,* it performs its function (Prints “Hi”), since it’s a void there’s no return statement so the program finishes executing *main* and returns 0.

* *A void function* will automatically return to the caller at the end of the function. No return statement is needed.

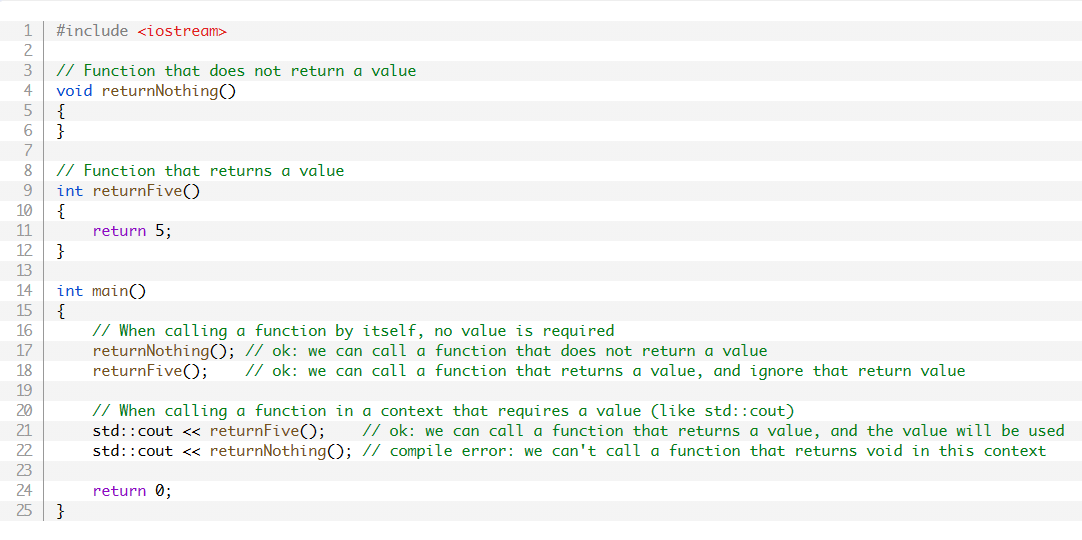
**Best Practice:** Do not put a return statement at the end of a void function.

Void functions **can’t** be used in *expressions* that require a value.



The first call to printHi() is called in a way that doesn’t require a value, so it’s fine.

The second call to printHi() won’t compile because statement is asking to return the value of printHi to ‘std::cout’ which won’t work because the void *printHi()* does not a return value.



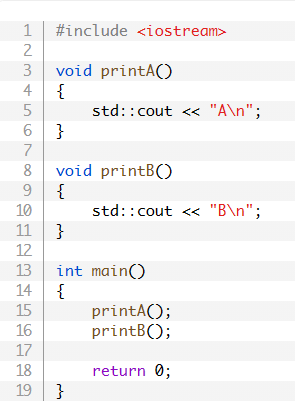
*Returning a value from a void function is a compile* *error.*

A close-up of a computer code

Description automatically generated

**Quiz.**

**Inspect the programs and state the output, or whether they will not compile.**

1. ****

This outputs:

A  
B

1. **A screenshot of a computer code

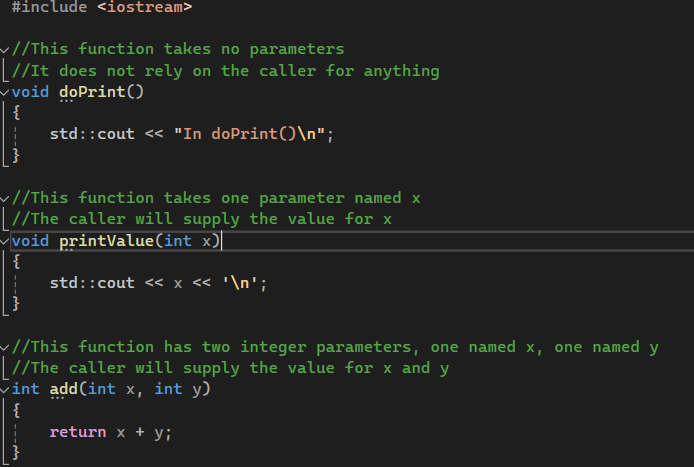
   Description automatically generated**

This won’t compile since the printA is being called to return a value for cout, which it can’t do because it’s a **void function.**

**Function Parameters and Arguments –**

A *function parameter* is a variable used in the header of a function (definition of the function that includes identifier, type, etc). **Parameters** work almost identically to variables defined inside the function, but with one difference; they are initialized with a value provided by the caller of the function. *Parameters* are defined in the function header by placing them in the parenthesis after the function name, with multiple parameters being separated by commas.

Some examples:



An **argument** is a value that is passed from the *caller* to the function (callee).

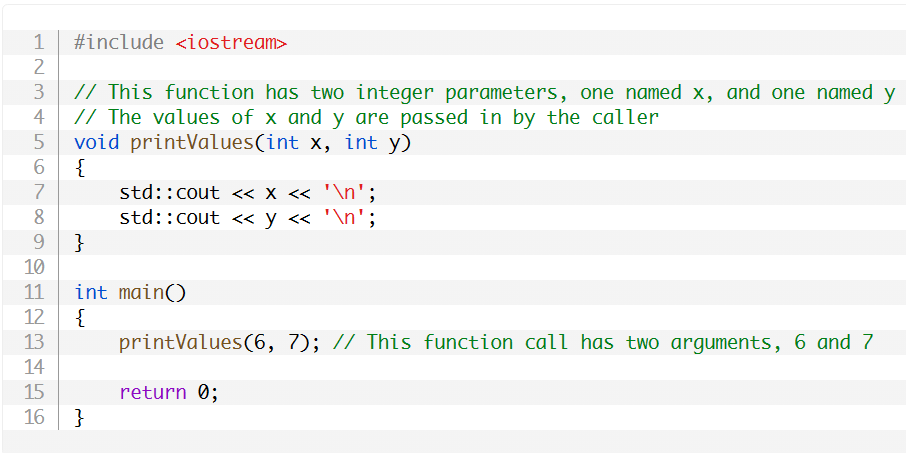
doPrint(); // this call has no arguments

printValue(6); // 6 is the argument passed to function printValue()

add(2, 3); // 2 and 3 are the arguments passed to function add()

**How parameters and arguments work together –**

When a function is called, all the *parameters* of the function are created as variables, and the value of each argument is copied into the matching parameter (using copy initialization). This process is called **pass by value**. Function parameters that utilize pass by value are called **value parameters.**



Note: the number of arguments must *generally* match the number of function parameters, or the compiler will throw an error.

**How parameters and return values work together –**

By using parameters and a return value, you can create functions that data as input, do some calculation with it, and return the value to the caller.

An example:

A screenshot of a computer

Description automatically generated

* Execution starts at the top of *main.* When ‘add(4,5)’ is evaluated, function *add* is called, parameters x and y are *initialized*.
* The return statement in function add evaluates x + y to produce the value, which is then return back to *main.* This value of 9 is then sent to std::cout to be printed on the console.

Output –

A number on a white background

Description automatically generated

A screenshot of a computer program

Description automatically generated

**Unreferenced Parameters –**

In certain cases, you will encounter parameters that are not used in the body of the function. These are called **unreferenced parameters.**

This can happen when a function parameter was used but is no longer needed.

A screenshot of a computer

Description automatically generated

* If the unused function parameter was simply removed, then any existing call to that function would break (because the function call would be supplying more arguments than the function can handle).

In a function definition, the name of a *parameter* is optional. Therefore, in cases where a parameter needs to exist but *is not used*, you can simply omit the name. A parameter without a name is called an **unnamed parameter.**

A screenshot of a computer

Description automatically generated

**Quiz.**

1. A screenshot of a computer code

   Description automatically generated

What is wrong with the program?

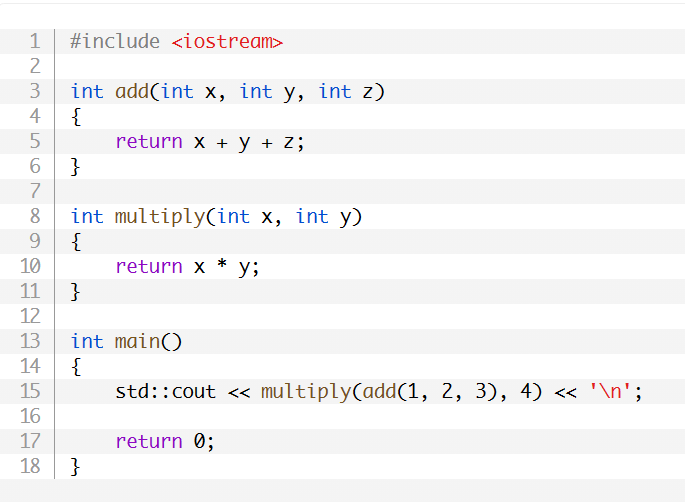
* Function *multiply* is a void return type so it can’t use a return statement. The function type should be an *int.*

1. A screenshot of a computer code

   Description automatically generated

What **two** things are wrong with the program?

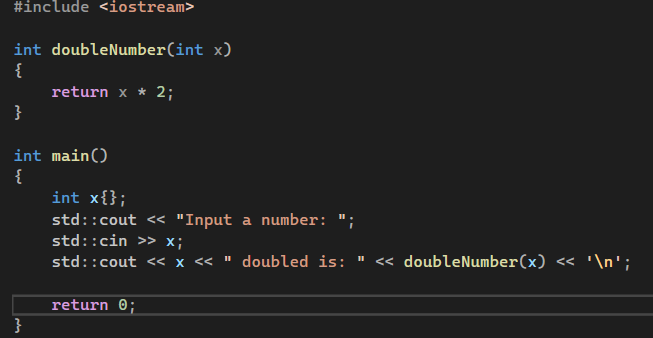
* Main is calling multiply but only passing *one* argument, when it needs to pass 2.
* Multiply function does not have a return statement.

1. 

What does this program print?

* It prints ‘24’, multiply calls add to provide the value for ‘x’, which is 6, 6 is multiplied by 4 giving 24.

1. Write a function named doubleNumber() that takes one integer parameter, the function should return double the value of the parameter.
2. Write a complete program that reads an integer in from the user, doubles it using the doubleNumber() function and then prints the doubled value out.



**Local Scope –**

*Variables* defined within the body of a function are called **local variables**.

Int add(int x, int y)  
{

Int z{x + y}; // z is a local variable.

Return z;  
}

***Function parameters*** are also considered local variables.

Int add(int x, int y) // function parameters x and y are also local variables  
{

Int z{x + y}; // z is a local variable.

Return z;  
}

**Local Variable Lifetime –**

* Function *parameters* are created and initialized when the function is entered.
* Local Variables are created at initialized at the point of definition.

Int add(int x, int y) // x and y created and initialized here

{

Int z{x + y}; // z created and initialized here

Return z;  
}

An object’s **lifetime** is defined to be the time between its creation and destruction. Note that variable creation and destruction happen when the program is *running,* not at compilation. Therefore, lifetime is a runtime property.

An example demonstrating lifetime (x):

A screenshot of a computer program

Description automatically generated

**Local Scope (In scope, out of scope) –**

An identifier’s scope determines *where* the identifier can be seen and used within the source code. When an identifier can be seen, we say it is **in scope.** When an identifier cannot be seen, we cannot use it, so we call it **out of scope.** Scope is a *compile-time* property, and trying to use it when its *out of scope* will result in a compile error.

A variable’s scope begins at the point of definition and stops at the end of the curly braces in which it is defined (or for function parameters, at the end of the function). This ensures variables cannot be used before the point of definition. Local variables defined in one function are not in scope in other functions that are called.

**Why functions are useful –**

Organisation – Complex programs get increasingly more difficult to manage. A function is almost like a *mini program* that’s written separately from the main function. This allows for reduced complexity, more manageable chunks of code.

Reusability – It avoids duplicated code (“Don’t Repeat Yourself”) and minimises probability of copy/paste errors. Functions can also be *shared* with other programs, reducing the amount of code required to be written from scratch.

Testing – Because functions are self-contained, once we’ve tested a function’s usability we don’t need to test it again. This reduces the amount of code we must test making it easier to debug.

Extensibility – When we need to extend a program to handle a new *case*, we can change the function in one place and have the change take effect everywhere the function is called.

Abstraction – To use a function, you only need its name, inputs, outputs, and its location. This lowers the knowledge required to use other people’s code.

**Using functions effectively –**

* Groups of statements that appear more than once should generally be made into a function. (For example. If user input is read multiple times in the *same way,* it’s a great candidate for a function. If an output is displayed in the *same way* multiple times, it should be a function).
* Code that has a well-defined set of inputs and outputs; if we have a list of items that we want to sort, the code to do the sorting would make a great function (even if it’s only done once). Another example is code that simulates the roll of a 6-sided dice. The current program might only use that in one place, but if you turn it into a function, it’s ready to be reused if you later extend the program or a future program.
* A function should generally perform only **one** task.
* When a function becomes too complicated, it can be split into multiple sub-functions. This is called **refactoring.**

**Forward declarations and definitions –**

A screenshot of a computer code

Description automatically generated

This program *does not compile* because, the compiler compiles the contents of code files sequentially. When it reaches the line in main() with function “add”, it generates an error because it doesn’t know what “add” is because it isn’t defined until line 9.

**Best Practice:** When addressing compilation errors or warnings in programs, resolve the *first issue* listed then compile again.

**Option 1 (To fix the function above): Reorder the function definitions –**

A screenshot of a computer code

Description automatically generated

This way when *main* calls *add*, the compiler knows what *add* is. In large programs it can be tedious trying to figure out which functions call which functions (and in what order) so they can be declared sequentially.

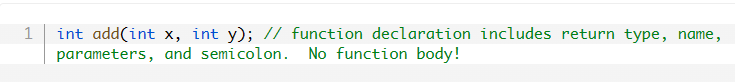
Furthermore, this way is not always possible. For example: If a program has two functions, A and B, and function A calls function B, and function B calls function A, then there’s no way to order the functions that will make the compiler happy; if you define A first, the compiler will complain it doesn’t know what B is and vice versa.

**Option 2: Use a forward declaration –**

A **forward declaration** tells the compiler about the existence of a function before defining the identifier.

This allows us to tell the compiler about the existence of a function before we define the function’s body, so when a compiler encounters a call to a function, it’ll understand that we’re making a function call, even if it doesn’t know yet how or where the function is defined.

To write a *function declaration* we use a function declaration statement (**function prototype).** The *function prototype* contains the function’s return type, name, and parameter types, terminated with a semicolon. The names of the parameters can be optionally included. This function body is not included in the declaration.



Here’s the original program with a *function declaration* as a forward declaration:

A screenshot of a computer code

Description automatically generated

Function declarations *don’t* need names for the parameters:

A white background with black text

Description automatically generated

**Best Practice**: Include parameter names in function declarations.

Tip: You can easily create function declarations by copy/pasting your function’s header and adding a semicolon.

**Forward declarations** are used to tell the compiler about the existence of some function that been defined in a different code file. Reordering isn’t possible in this scenario because the caller and callee are in **different files!**

Forward declarations can also be used to define our functions in an *order-agnostic* manner. This allows us to define functions in whatever order maximises organisation for reader understanding.

Less often, there are times when two functions call each other. Reordering isn’t possible in this case because there is no way to reorder the functions such that each is before the other. Forward declarations give us a way to resolve such circular dependencies.

**Forgetting the function body –**

If a forward declaration is made, but the function is never called, the program will compile and run. *However*, if a forward declaration is made and the function is called, but the program never defines the function, the program compiles fine, but the linker will complain that it can’t resolve the function call.

A screenshot of a computer code

Description automatically generated

In the program, *add* is declared and called, but it isn’t defined. When the program is ran it produces an error because *add*(int, int) was never defined.

*Forward declarations* are most often used with functions, however they can also be used to identify variables and types.

**Declarations vs. Definitions –**

A **declaration** tells the compiler about the existence of an identifier and its associated *type* information. For example:

A white background with black text

Description automatically generated

A **definition** is a *declaration* that implements (for functions and types) or *instantiates* (for variables) the identifier.

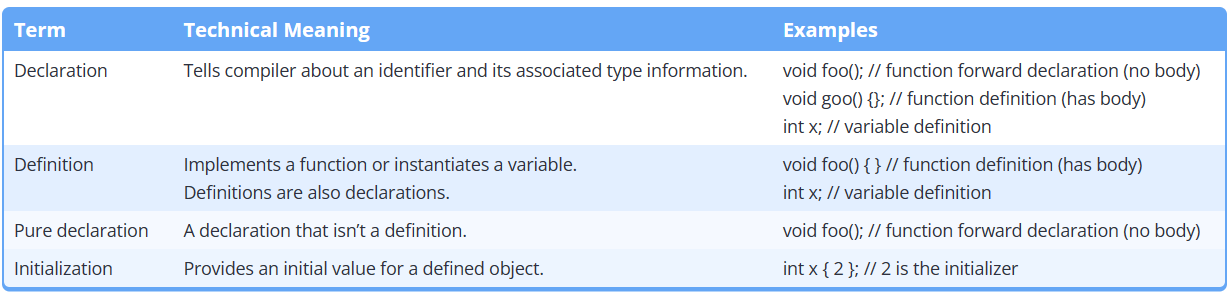
A screenshot of a computer code

Description automatically generated

In C++, all definitions are declarations. Therefore, *int x;* is both a definition and declaration.

**Note:** Not all declarations are definitions. Declarations that aren’t definitions are called **pure declarations.** Types of pure declarations include forward declarations for function, variables, and types.

*“The term ‘declaration’ is typically used to mean ‘a pure declaration’, and ‘definition’ is used to mean ‘a definition that also serves as a declaration’. Thus, we’d typically call int x; a definition, even though it is both a definition and a declaration*.”



**The one definition rule (ODR) –**

The one definition rule (or ODR) has three parts:

1. Within a *file,* each function, variable, type, or template can only have one definition. Definitions occurring in different scopes (e.g. local variables defined inside different functions, or functions defined aside different namespaces) do not violate this rule.
2. Within a *program*, each function or variable can only have one definition.
3. Types, templates, inline functions, and inline variables are allowed to have duplicate definitions in different files, so long as each definition is identical.

* Violating **Part 1** of the ODR will cause the compiler to issue a *redefinition error.*
* Violating **Part 2** will cause the linker to have a redefinition error.
* Violating **Part 3** will cause undefined behaviour.

A screenshot of a computer code

Description automatically generated

Function *add(int, int)* is defined twice (in the global scope), and variable *int x* is defined twice (in the scope of main()). This causes compiler errors.

However, it is not a violation of ODR part 1 for main() to have a local variable defined as int x and add() to also have a function parameter defined as int x. These definitions occur in *different scopes* (in the scope of each respective function), so they are separate functions for two distinct objects, not a definition and redefinition of the same object.

**Quiz.**

1. **What is a function prototype?**

A function prototype is a declaration statement of function including its identifier, parameters, type without the body of the function. It tells the compiler about the existence of a function before its defined.

1. **What is a forward declaration?**

A forward declaration tells the compiler that an identifier exists before it’s defined and if the function is defined in another file.

1. **How do we declare a forward declaration for functions?**

Include the function’s identifier, parameters and return type. Parameter names are optional (Function prototype).

1. **Write the function declaration for this function:**

**A close up of text

Description automatically generated**

Int doMath(int first, int second, int third, int fourth); (*Semicolon because it’s a statement)*

1. **For each of the following programs, state whether they fail to compile, fail to link, or compile and link successfully.**
2. **A screenshot of a computer

   Description automatically generated**

Compilation error, *add* has two parameters but the statement in main uses 3.

1. **A screenshot of a computer program

   Description automatically generated**

Linker error (Compilation error), *add* is declared with 2 parameters but the function defined takes 3.

1. **A screenshot of a computer program

   Description automatically generated**

Linker error. The compiler will match the forward declaration of the add function to the function call in main. But no add function that takes 2 parameters was implemented (only the function that takes 3), so the linker will complain.

1. **A screenshot of a computer program

   Description automatically generated**

Compiles and links. The function call matches the forward declaration, and the definition also matches. The names don’t matter as the names in the *declaration* are **optional** (and ignored by the compiler).

1. **A screenshot of a computer program

   Description automatically generated**

Compiles and links.

**Multiple files –**

When programs get larger, it’s common to split them into multiple files for organisational purposes.

**Best Practice**: When you add new code files to your project, give them a .cpp extension.

A screenshot of a computer code

Description automatically generated

This program will fail to compile because it does not know what *add* is since its not defined (or declared) before the main function.

The solution is to use forward declaration to declare the int *add* function and define it another .cpp file.

A black screen with white text

Description automatically generatedA black background with white letters

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer program

Description automatically generated

**Insight –**

When an identifier is used in an expression, the identifier must be connected to its definition.

* If the compiler has seen *neither* a forward declaration nor a definition for the identifier in the file being compiled, it will result in an error at the point where identifier is used.
* Otherwise, if a definition exists in the same file, the compiler will connect the use of the identifier to its definition.
* Otherwise, if a definition exists in a different file (and is visible to the linker), the linker will connect the use of the identifier to its definition.
* Otherwise, the linker will issue an error indicating that it couldn’t find the definition for the identifier.

**Quiz.**

1. **Split the following program into two files. Main.ccp should have the main function, and input.cpp should have the getInteger function.**

A computer screen with text and symbols

Description automatically generatedA screenshot of a computer code

Description automatically generated

A screen shot of a computer code

Description automatically generated

A black background with white text

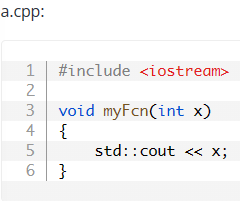
Description automatically generated

**Naming collisions** **–**

If two identical identifiers are introduced into the same program in a way that the compiler or linker can’t tell them apart, the compiler or linker will produce an error. This error is generally referred to as a **naming collision** (or naming conflict).

* If the colliding identifiers are in the same file, the result will be a *compiler* error.
* If the colliding identifiers are in separate files, in the same program, the result will be a *linker* error.

Example:

A screenshot of a computer code

Description automatically generated