Functions

Functions

main()

The only exception to the rule that a value-returning function must return a value via a return statement is for function main(). The function main() will **implicitly return the value 0** if no return statement is provided. That said, it is best practice to explicitly return a value from main, both to show your intent, and for consistency with other functions (which will exhibit undefined behavior if a return value is not specified).

1. doPrint(); // t**his call has no arguments**
2. printValue(6); // **6 is the argument passed to function printValue()**
3. add(2, 3); // **2 and 3 are the arguments passed to function add()**

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

The Google C++ style guide recommends using a comment to document what the unnamed parameter was:

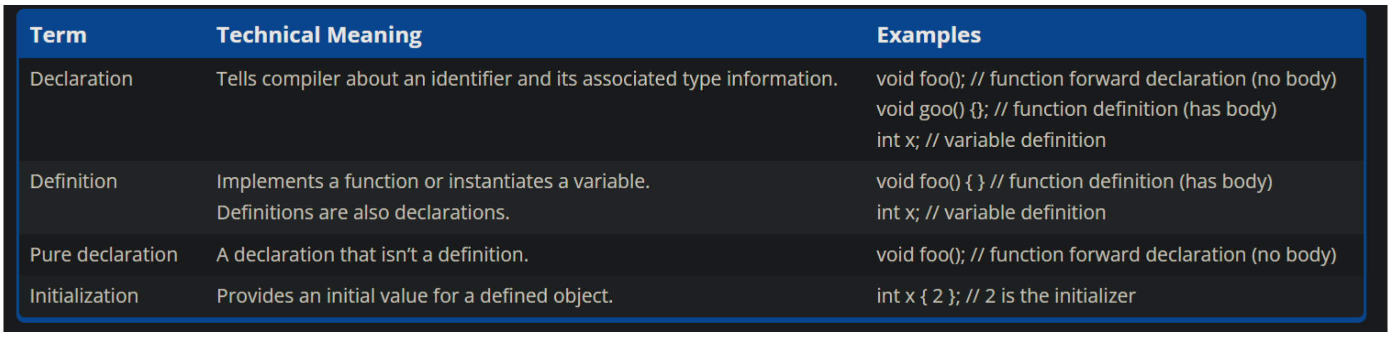
```
void doSomething(int /*count*/)
{
}
```

Why use functions

Organization -- As programs grow in complexity, having all the code live inside the main() function becomes increasingly complicated. A function is almost like a mini-program that we can write separately from the main program, without having to think about the rest of the program while we write it. This allows us to reduce a complicated program into smaller, more manageable chunks, which reduces the overall complexity of our program.
Reusability -- Once a function is written, it can be called multiple times from within the program. This avoids duplicated code (“Don’t Repeat Yourself”) and minimizes the probability of copy/paste errors. Functions can also be shared with other programs, reducing the amount of code that has to be written from scratch (and retested) each time.
Testing -- Because functions reduce code redundancy, there’s less code to test in the first place. Also because functions are self-contained, once we’ve tested a function to ensure it works, we don’t need to

test it again unless we change it. This reduces the amount of code we have to test at one time, making it much easier to find bugs (or avoid them in the first place).
Extensibility -- When we need to extend our program to handle a case it didn't handle before, functions allow us to make the change in one place and have that change take effect every time the function is called.
Abstraction -- In order to use a function, you only need to know its name, inputs, outputs, and where it lives. You don't need to know how it works, or what other code it's dependent upon to use it. This lowers the amount of knowledge required to use other people's code (including everything in the standard library).

Conversely, 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.



Term	Technical Meaning	Examples

Declaration	Tells compiler about an identifier and its associated type information.	void foo(); // function forward declaration (no body)
		void goo() {}; // function definition (has body)
		int x; // variable definition

Definition	Implements a function or instantiates a variable. Definitions are also declarations.	void foo() { } // function definition (has body) int x; // variable definition
Pure declaration	A declaration that isn't a definition.	void foo(); // function forward declaration (no body)
Initialization	Provides an initial value for a defined object.	int x { 2 }; // 2 is the initializer

The one definition rule (ODR)

The one definition rule (or ODR for short) is a well-known rule in C++. The ODR has three parts:

1. Within a file, each function, variable, type, or template in a given scope can only have one definition. Definitions occurring in different scopes (e.g. local variables defined inside different functions, or functions defined inside different namespaces) do not violate this rule.
2. Within a program, each function or variable in a given scope can only have one definition. This rule exists because programs can have more than one file. Functions and variables not visible to the linker are excluded from this rule (Internal linkage).
3. Types, templates, inline functions, and inline variables are allowed to have duplicate definitions in different files, so long as each definition is identical. We haven't covered what most of these things are yet, so don't worry about this for now -- we'll bring it back up when it's relevant.

Naming collisions / namespaces

The :: symbol is an operator called the scope resolution operator
Avoid using-directives (such as using namespace std;) at the top of your program or in header files. They violate the reason why namespaces were added in the first place.

The #ifdef preprocessor directive allows the preprocessor to check whether an identifier has been previously #defined. If so, the code between the #ifdef and matching #endif is compiled. If

not, the code is ignored.

Consider the following program:

```
#include <iostream>

#define PRINT_JOE

int main()
{
#ifdef PRINT_JOE
    std::cout << "Joe\n"; // will be compiled since PRINT_JOE is defined
#endif

#ifdef PRINT_BOB
    std::cout << "Bob\n"; // will be excluded since PRINT_BOB is not defined
#endif

    return 0;
}
```

if 0

One more common use of conditional compilation involves using #if 0 to exclude a block of code from being compiled (as if it were inside a comment block):

```
#include <iostream>

int main()
{
    std::cout << "Joe\n";

#if 0 // Don't compile anything starting here
    std::cout << "Bob\n";
    std::cout << "Steve\n";
#endif // until this point

    return 0;
}
```

When we use double-quotes, we're telling the preprocessor that this is a header file that we wrote. The preprocessor will first search for the header file in the current directory. If it can't find a matching header there, it will then search the include directories.

Best practice

To maximize the chance that missing includes will be flagged by compiler, order your #includes as follows:

1. The paired header file
2. Other headers from your project
3. 3rd party library headers
4. Standard library headers
   The headers for each grouping should be sorted alphabetically (unless the documentation for a 3rd party library instructs you to do otherwise).

Header Guards

square.h

```
#ifndef SQUARE_H
#define SQUARE_H

int getSquareSides()
{
    return 4;
}

int getSquarePerimeter(int sideLength); // forward declaration for
getSquarePerimeter

#endif
```

Header guards are designed to ensure that the contents of a given header file are not copied more than once into any single file, in order to prevent duplicate definitions.

Duplicate declarations are fine -- but even if your header file is composed of all declarations (no definitions) it's still a best practice to include header guards.

Note that header guards do not prevent the contents of a header file from being copied (once) into separate project files. This is a good thing, because we often need to reference the contents of a given header from different project files