Contents

[The Scope of Things 1](#_Toc57058453)

[Multi-File Programs 2](#_Toc57058454)

[Getting a Header Yourself 3](#_Toc57058455)

[How to Get Your Functions Inline 3](#_Toc57058456)

[Default Arguments 4](#_Toc57058457)

[Function Overload! 5](#_Toc57058458)

[Function Templates 6](#_Toc57058459)

[How to Build Function Templates 6](#_Toc57058460)

[Review 7](#_Toc57058461)

## The Scope of Things

Take a look at the program below. We have a void function named favorite\_animal() and main() with a few statements inside.

#include <iostream>  
   
std::string sea\_animal = "manatee";  
   
void favorite\_animal(std::string best\_animal) {  
   
  std::string animal = best\_animal;  
  std::cout << "Best animal: " << animal << "\n";  
   
}  
   
int main() {  
   
  favorite\_animal("jaguar");  
   
  std::cout << sea\_animal << "\n";  
  std::cout << animal << "\n";  
   
}

When this program is compiled and executed, sea\_animal will print, but animal won’t. Why do you think that’s the case?

*Scope* is the region of code that can access or view a given element.

* Variables defined in *global scope* are accessible throughout the program.
* Variables defined in a function have *local scope* and are only accessible inside the function.

sea\_animal was defined in global scope at the top of the program, outside of main(). So sea\_animal is defined everywhere in the program.

Because animal was only defined within favorite\_animal() and not returned, it is not accessible to the rest of the program.

### Multi-File Programs

Programs can grow quickly. With a few functions, you can declare the function above main() and then you can define the function below main() like this:

#include <iostream>  
   
// Declaration at the top:  
void eat();  
   
int main() {  
   
  eat();  
   
}  
   
// Definition at the bottom:  
void eat() {  
   
  std::cout << "nom nom nom\n";  
   
}

But this isn’t ideal when your code gets longer; it’s common to use the same function in more than one **.cpp** file.

To make your code cleaner and more modular, you can move the function definitions over to another specialized **.cpp** file (e.g., **my\_functions.cpp**), leaving a list of declarations above main().

But files, like functions, have scope. So, how does the main() program know about the function definitions?

Before your program even compiles, it links together any files you list in your compilation statement into a single executable:

g++ main.cpp my\_functions.cpp

And voila! Your program knows the function definitions.

## Getting a Header Yourself

If your program keeps growing, you may have to scroll through many declarations before you see main(). That doesn’t seem like the best way to do things. Plus you don’t want to keep declaring the same functions over and over for different files — making changes would be incredibly tiresome!

Well, you can take those function declarations and move them all over to a *header file*, another file — usually with the same name as the file with all of the function definitions — with the extension **.hpp** or **.h**. For example, if your function definitions are in **my\_functions.cpp**, the corresponding header file would be **my\_functions.hpp** or **my\_functions.h**.

So how do you bring everything from a header file into scope for another file? Do you just link the header in the compilation statement like you did with the second **.cpp** file?

As it turns out, with headers, you can just add #include "my\_functions.hpp" to the very top of **main.cpp**:

#include "my\_functions.hpp"

Boom! This line pastes in everything from **my\_functions.hpp**. Now you have access to all of the function declarations you stowed away in your header.

## How to Get Your Functions Inline

Once you set foot in the wild of C++ development, you may encounter the term “inline functions” with a couple different meanings. An *inline function* is a function definition, usually in a header file, qualified by inline like this:

inline   
void eat() {  
   
  std::cout << "nom nom\n";  
   
}

Using inline advises the compiler to insert the function’s body where the function call is, which sometimes helps with execution speed (and sometimes hinders execution speed). If you do use it, we recommend testing how it affects the execution speed of your program. The bottom line is inline is something you’ll probably encounter, but may never use.

However, you will sometimes also hear about “inline functions” that are just member functions (i.e. functions inside of classes — we’ll explain classes later) which have been defined and declared in a single line in a header file because the function body is so short:

// cookie\_functions.hpp  
   
// eat() belongs to the Cookie class:  
void Cookie::eat() {std::cout << "nom nom\n";}

Please note that you should ALWAYS add the inline keyword if you are inlining functions in a header (unless you are dealing with member functions, which are automatically inlined for you).

## Default Arguments

If you add a parameter to a function in C++, then an argument will be required when you call the function. What does “required” mean here? Well, you’ll get an error. But what if 9 times out of 10, you know you’ll use the same input value? It would be really annoying to have to enter the same value over and over again.

To make your code more flexible for situations like this, you can add *default arguments* to your function declarations. Default arguments are values assigned to parameters when the function is declared and defined:

// Declaration  
void intro(std::string name, std::string lang = "C++");  
   
// Definition  
void intro(std::string name, std::string lang) {  
  std::cout << "Hi, my name is "  
            << name  
            << " and I'm learning "  
            << lang  
            << ".\n";  
}

Then, if you leave the argument blank in your function call, instead of an error, your function will run with the default value. Meanwhile, if you DO have an argument to add when you call the function, that argument will replace the default argument when your code executes.

Either of these will work for the function we defined:

intro("Mariel")  
// "Hi, my name is Mariel and I'm learning C++."

intro("Mariel", "Python")  
// "Hi, my name is Mariel and I'm learning Python."

**Important:** Your computer cannot read your mind; it determines which argument corresponds with which parameter based on order.

Parameters without default arguments come first. This will work for add\_nums(3); because the computer knows 3 corresponds to num1:

int add\_nums(int num1, int num2 = 0);

But the following does NOT work for add\_num(3); the computer assumes that 3 still corresponds to num1:

int add\_nums(int num1 = 0, int num2);

Similarly, when a function has two default arguments, you still need to call with both arguments — if BOTH of the following are true:

* The first argument IS the default value.
* The second argument is NOT the default value.

## Function Overload!

What if you want a function to accept an argument that can be either an int OR a double? Or what if you want some function parameters to be optional? C++ has a trick up its sleeve just for such situations.

In a process known as *function overloading*, you can give multiple C++ functions the same name. Just make sure at least one of these conditions is true:

* Each has different type parameters.
* Each has a different number of parameters.

Overloading enables you to change the way a function behaves depending on what is passed in as an argument:

void print\_cat\_ears(char let) {  
  std::cout << " " << let << "   " << let << " " << "\n";  
  std::cout << let << let << let << " " << let << let << let << "\n";  
}  
   
void print\_cat\_ears(int num) {  
  std::cout << " " << num << "   " << num << " " << "\n";  
  std::cout << num << num << num << " " << num << num << num << "\n";  
}

Given the above functions, you could call the functions like so and C++ will know what to do:

print\_cat\_ears('A');  
print\_cat\_ears(4);

Output:

A   A   
AAA AAA  
   
 4   4  
444 444

## Function Templates

Overloading can be really tedious. Imagine you want to create a new function that works with int, float, double, and other number types. Do you really need to redefine the SAME function body over and over again with different parameters?

Thankfully not! When two functions have different types but the same body — as was the case with print\_cat\_ears() —, there is a cleaner solution you can use: templates.

A template is a C++ tool that allows programmers to add data types as parameters.

This feature comes in handy for classes as well as for functions. In fact, std::string and std::vector are both template-based types. The [C++ standard library](https://en.cppreference.com/w/cpp/header) is full of templates!

## How to Build Function Templates

So how do we implement templates with actual code? Unlike regular functions, templates are entirely created in header files.

Templates let us choose the type implementation right when you call the function. The type we choose may apply to the return type, a parameter type, or both.

Here’s how we could build a template for print\_cat\_ears() so that the parameter type is flexible:

template <typename T>  
void print\_cat\_ears(T item) {  
   
  std::cout << " " << item << "   " << item << " " << "\n";  
  std::cout << item << item << item << " " << item << item << item << "\n";  
   
}

We can call the function for int, char, std::string, or double:

print\_cat\_ears(2);  
   
// the output:  
//  2   2  
// 222 222

Now we can use print\_cat\_ears() with a parameter of any type we want, as long as the type can be used with the methods expected. For example, we couldn’t pass an std::vector into the current print\_cat\_ears() because << won’t work with std::vector.

**Note:** Using templates will slow down the program’s compile time, but speed up the execution time.

## Review

You now know a bit about how scope works for functions and files, as well as how to make functions more flexible for different use cases:

* Scope is the region of code that has access to an element.
  + Globally scoped variables are accessible everywhere.
  + A variable created inside a function has local scope and can’t be accessed outside the function.
* C++ functions are usually split to make code more modular:
  + The declaration in a header file.
  + The definition in another **.cpp** file.
* Programs with multiple **.cpp** files need to be linked at compile time:

g++ main.cpp fns.cpp

* Header files must be included in the file with main():

#include "fns.hpp"

* You can define a function inline using the inline keyword, which may or may not improve execution speed.
* Default arguments can be added to function declarations so that you can call the function without including those arguments.
* You can overload C++ functions so that they handle different types of input and return different types.
* A function template enables a function to behave the same with different types of parameters.