Scope | Coursera

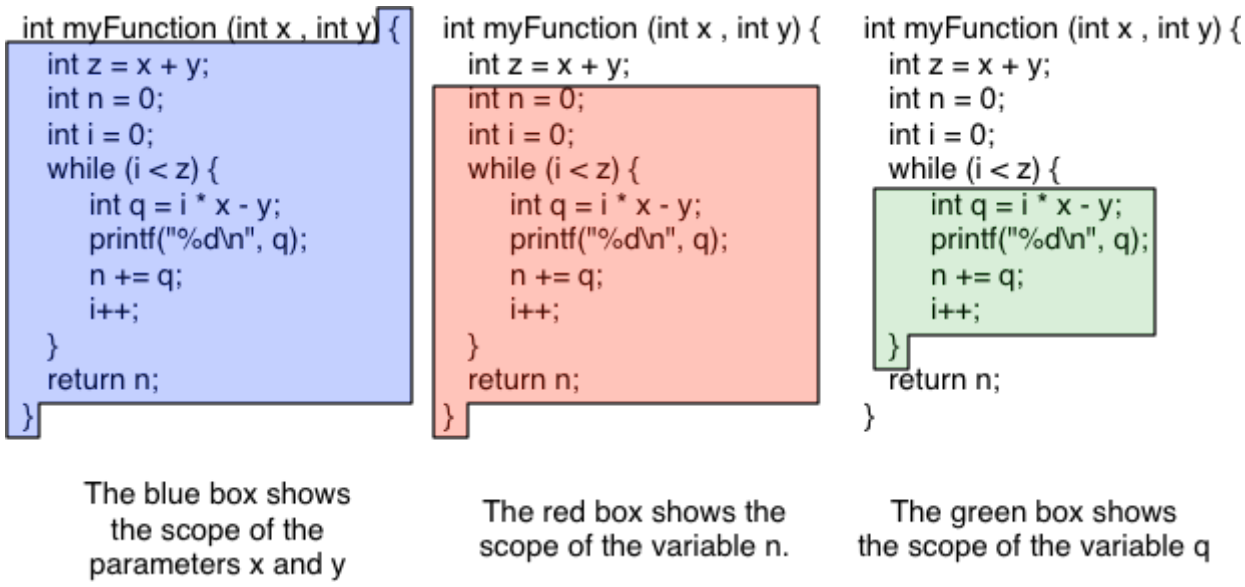
coursera.org/learn/programming-fundamentals/supplement/gSJnU/scope

Scope

Scope

So far, all of our code examples have had only one variable with a particular name. However, in real programs—which may be quite large and developed by multiple people—we may have many different variables with the same name. This possibility means that we need rules to determine which variable a particular name refers to. These rules are based on the notion of *scope*.

The scope of a variable is the region of code in which it is visible. Within a variable's scope, its name may refer to it. Outside of a variable's scope, nothing can refer to it directly. Most variables that you will use will be local variables—variables that are declared inside of a function—and function parameters. In C, the scope of a local variable begins with its declaration and ends at the closing curly-brace (}), which closes the block of code—the code between matching open and close curly braces—that the variable was declared in. Function parameters have a scope of the entire function to which they belong.

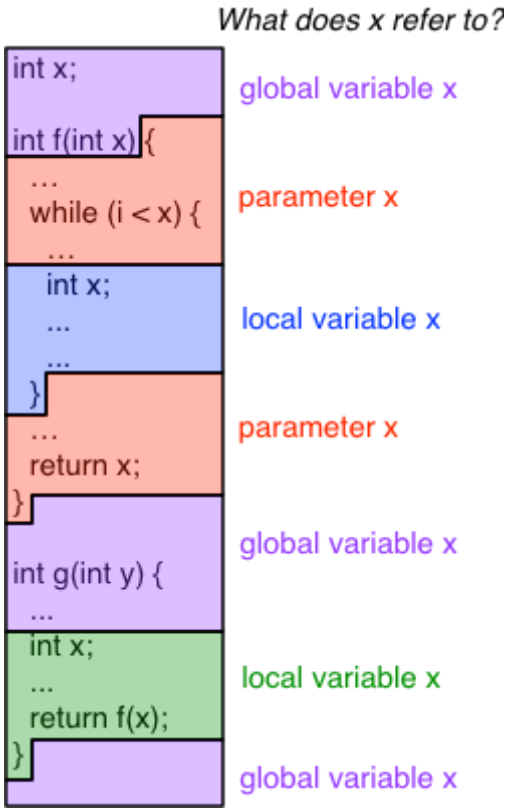


This figure shows a snippet of code (we have not learned the details of what most of this code does, but that is not important—we are just interested in the scope of the variables). The figure shows the same piece of code three times, with different scopes highlighted. The leftmost portion of the figures shows the scope of the parameters (**x** and **y**)—which is the entire function—in a blue box. The middle portion shows the scope of the variable **n**—which starts at its declaration and continues to the close curly brace which ends the function—in a red box. The right portion shows the scope of the variable **q**—which starts at its declaration and ends at the next curly brace—in a green box.

To determine which variable a name refers to, we must first determine which variable(s) with that name are in scope at the reference. If no variables of that name are in scope, then the reference is illegal. If exactly one variable is in scope, then the name refers to that variable. If multiple variables are in scope, we select the one whose declaration is in the innermost enclosing block. That is, if you went backwards out of blocks, through open curly braces, the variable which would go out of scope first is the one to use.

The figure below shows a code fragment with four different **x**'s in it. (As the actual behavior of the code is irrelevant to this example, much of it is replaced with [...]) The first **x** in the figure is declared outside of any of the functions—it is a global variable. The "box" for a global variable exists outside of any frames and is created when the program starts. If the global variable is initialized in its declaration, the value is also placed in the box before the program starts. The areas where **x** references this variable are colored purple.

We note that there is a time and place to use global variables, but their use should be rare. When novice programmers learn about global variables, they often want to use them for all sorts of inappropriate purposes. Typically these uses reflect a lack of understanding of parameter passing or how functions return values. We recommend against using global variables for any problem in this specialization, and more generally unless it is truly the correct design approach.



The next **x** in our example is the parameter to the function **f**. The scope for this **x** begins at the open curly brace ({) of **f**'s body and ends at the matching close curly brace (}). The region of the program where **x** references the parameter to **f** are shown in red. Observe that the red begins and ends with the curly braces surrounding the body of **f**, but has a "hole" where there is a different **x** in a smaller scope in the middle.

The "hole" in the red region corresponds to the portion of the code (shown in blue) where **x** references the local variable declared inside of the while loop's body. After this local variable **x** goes out of scope at the closing curly brace of the block it was declared in, we return to the red region, where the parameter of **f** is what we reference with the name **x**.

Between the end of **f** and the declaration of a local variable named **x** inside of function **g**, the global variable is what the name **x** references—shown in the figure by coloring this region of code purple. When there is a local variable named **x** declared inside of **g**, then the name **x** references it (this area is shown in green) until it goes out of scope, at which point the name **x** again references the global variable.

If all of that seems complicated, you will be comforted by the fact that thinking through such issues should not come up in well-written code. Ideally, you should write your code such that you have at most one variable by any particular name in scope at a time (related to this point: you should name your variables meaningfully—**x** is seldom a good name for a variable, unless of course it represent the **x**-coordinate of a point or something similar). However, you should still know what the rule is, as it is common to many programming languages. You may come across code that has multiple variables of the same name in scope at some point and need to understand how to read it.

Completed