
SIMULATING A PROGRAM'S EXECUTION

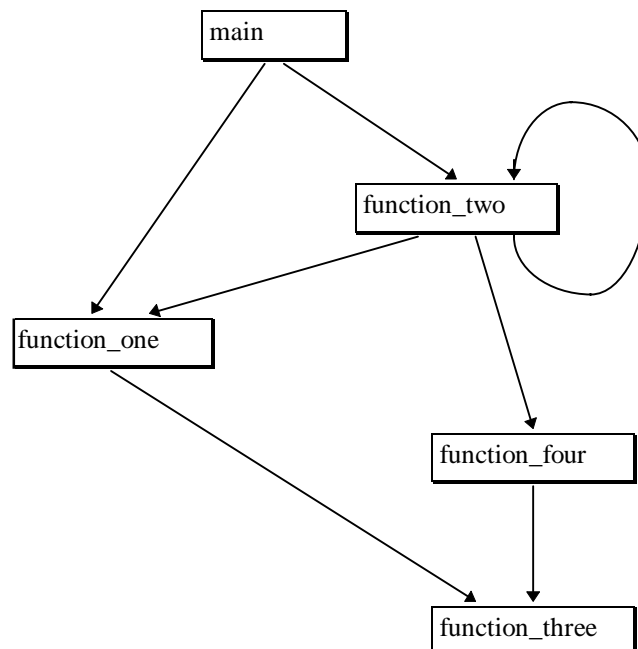
INTRODUCTION

A useful skill that all good programmers acquire is the ability to read a program's source code, and to simulate all or part of it on paper. Although there are computer-based tools for doing the same thing, doing it by hand is often quicker and can be just as effective. Moreover, doing it by hand can improve your understanding of the program's execution, which is just as useful as the simulation itself.

CALLING DIAGRAMS

Prior to simulating source code, a diagram that shows how the program's various execution units fit together is useful. A diagram that shows the structure of the program in terms of one function calling another is a *calling diagram*. Since all programs have a 'main' function, or its equivalent, there is always one function that can serve as the top of the diagram, the 'root' of the tree. Each node in the tree is the name of a function. A branch is drawn if a function can call another; the arrow head on the branch indicates the direction of call. Some functions do not call any others (except perhaps library functions, although even this is not necessary) and so they are 'leaves' of the tree. If a function calls another function that is already in the diagram, a branch (it may have to be curved) is drawn from the caller to the callee. Below is an example sketch program and its calling tree. Of course, there are other operations carried out by the program, but these are not mentioned in the diagram.

See Source Code Examples #1



SIMULATING CODE

It is possible to simulate any part of a program, as long as the initial inputs can be calculated, or even guessed. Some functions, or groups of functions have a more general requirement i.e. they should work on *any* input, but that problem we will leave until we discuss program testing. For now, let us assume that we know the inputs to the program, and proceed from there. In order to understand how a program runs, we need a model of execution, and this concerns what we call the program *state*.

DATA FLOW AND PROGRAM STATE

The arrows in the calling diagram can serve another purpose; they can be considered as carrying data from function to another. Data is passed through the parameters of the function, and passes from the arguments in the function call to the function body. Within the function body, statements process the data and data flows from one statement to another through the variables that we use to hold the data values. If we lump all of the variables used by a program together, we get the idea of a program state. At any one time the values in all the variables is a snapshot of the state of the program at that time. As statements are executed, the state changes through assignments to variables. It is this changing state that we are going to simulate. Eventually we reach a terminating statement (or just the end of the main function) and the program state at that point is the final state. Consider this program fragment:

See Source code examples #2

Although there may be many more variables in the whole program, only two are used in statements: v1 and v2. So our simulation starts with the initial values of v1 and v2, thus:

v1	v2
10	1000

These initial values come from the (local) definitions of v1 and v2, but you may have to choose values in order to start the simulation. The next step is to locate the first statement. In our case, there is only one - a while loop statement. With compound statements like loops almost always is, you need to understand how it operates. We can refer to the flow chart for a while loop in order to do this. Each of the control statements *if*, *if-else*, *while*, *do-while*, and *for*, has a flow chart which you can find in your textbook.

Simulating the while loop consists of following the flow-chart, having identified the test and the body. The values change as follows. Note the output column for anything printed by the program.

v1	v2	output
10	1000	
9	900	Excess over 500 = 500
8	800	Excess over 500 = 400
7	700	Excess over 500 = 300
6	600	Excess over 500 = 200
5	500	Excess over 500 = 100

Each line in the simulation trace (or just ‘trace’) corresponds to a statement in the program. Often a value does not change in a particular statement, so there is no need to put anything in that column.

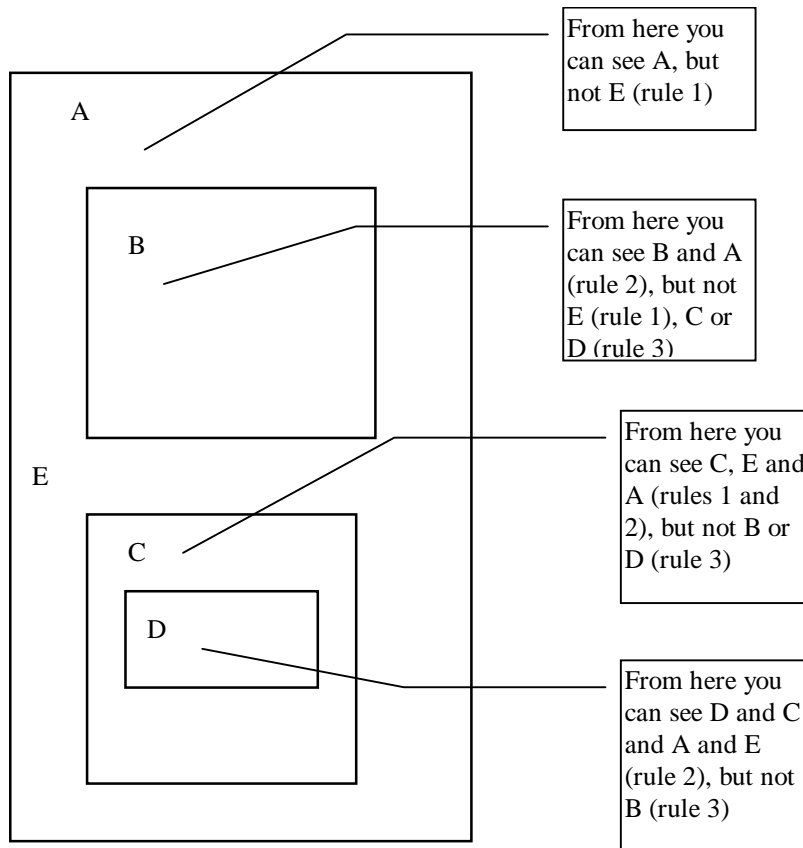
REGIONS OF SCOPE

Names are subject to the rules of *scope* which can be understood through a visual metaphor called *visibility*. There are two main kinds of scope in a program: file scope and block scope. File scope is sometimes called global scope because names declared at the topmost level are available throughout the whole program. These include any global variable names, type definitions and *all* function names. Block scope (sometimes called function scope, although it covers more than just functions) is defined by the region covered by the body of a procedure. This covers all function bodies in C and C++; methods in Java; subroutines in Fortran and functions and procedures in Pascal. Such declarations are called *local* to distinguish them from global.

The visibility metaphor invites you to imagine you are standing in the file, trying to see where there is a declaration of a name. There are three rules:

1. You can see names above you, but not below you, where above and below refer to the normal reading order, i.e. top to bottom, down the page.
2. You can see out of a region of scope (imagine there are walls around the region, as defined by matched braces) to any outer enclosing scope, including the global scope which is the outermost region.
3. You cannot see into any other scope, i.e. the walls are like one-way mirrors that you can see out of, but not into.

The diagram below illustrates these three rules:



A program corresponding to the above diagram might be:

See Source Code Examples #3

DRAWING THE HAND SIMULATION CHART

FIND THE SCOPE REGIONS

1. Look for global *variable* names. Put these in a global scope column in the chart.
2. Look for functions (including function main). These will each get a column in the chart. Label the column with the function name.
3. Look for local declarations and in the column corresponding to the function where the local declaration lives, put a sub-column for each one, labeled with the variable name. You can also include any inner nested scopes in a similar fashion by grouping the inner locals together.
4. Add a column for output.

RUN THE SIMULATION

Start at the first executable statement in function main, and, using one row per assignment show the changes of variables as the program execution proceeds. Draw a horizontal line when a function is called and when it returns. A row can contain more than one entry where variable initialization and parameter passing is done; look at the example below for how to do this.

EXAMPLE

The simulation for the program below follows the code.

See Source Code Examples #4

The initial table is the result of analyzing the regions of scope. The global region has two variables, `g1` and `g2`. There are three regions apart from this the functions `f1`, `f2` and `main`. `f1` has one local variable, `w`, and one parameter, `xx`. `f2` has a local variable also called `w`, but the rules of scope ensure that this is distinct from the one in `f1`. The function `main` has a local called `r`. We also add a column for output printed by the program.

global		f1		f2	main	output
g1	g2	xx	w	w	r	

Initialization of global variables occurs before the program starts running, so we start the table by inserting the initial values of `g1` and `g2`. They can be put on the same line since they are effectively done at the same time.

global		f1		f2	main	output
g1	g2	xx	w	w	r	
100	20					

The first executable statement in `main`, which is the entry point of the program, is the line `r = f2()`; This is a call to function `f2`, so we draw a line in `f2`'s column showing that `f2` is called. We can also put the initialization of any parameters (there are none) and any local variables. `w` gets the value 5.

global		f1		f2	main	output
g1	g2	xx	w	w	r	
100	20					
				5		

The next line is $g1 = g2 + w$; so we put the new value of $g1$ in the next row. Notice how global variables are affected by assignment inside an inner scope.

global		f1		f2	main	output
g1	g2	xx	w	w	r	
100	20					
				5		
25						

The next line is $\text{return } w + 2$; so we return the value 7 as the value of the call to function $f2$, and immediately assign it to the variable r in main. We draw a line in $f2$'s column before the assignment to r , however, to show that $f2$ has exited.

global		f1		f2	main	output
g1	g2	xx	w	w	r	
100	20					
				5		
25						
					7	

Now the main function calls `f1`, passing it the value of `r`, which is 7. Another line, this time in `f1`'s column is drawn.

global		f1		f2	main	output
g1	g2	xx	w	w	r	
100	20					
				5		
25						
					7	

The value of its parameter, `xx`, and its local variable, `w` are put in the next row. Again this is because initializations like this are effectively carried out at the same time.

global		f1		f2	main	output
g1	g2	xx	w	w	r	
100	20					
				5		
25						
					7	
		7	10			

`f1` executes the statement `w = w + g1`; producing the value of 35 in `w`.

global		f1		f2	main	output
g1	g2	xx	w	w	r	
100	20					
				5		
25						
					7	
		7	10			
			35			

The value of the expression w / xx is then printed. Finally `f1` returns, a line is drawn in its column and the main function prints the value of `g1`, and then returns, thus ending the program's execution. A line in `main`'s column signifies that the program has terminated.

global		f1		f2	main	output
g1	g2	xx	w	w	r	
100	20					
				5		
25						
					7	
		7	10			
			35			
						5
						25

