Scope of Variables

Scope describes the lifetime of a variable. Local variables are preferred over those which are global. This is why you will often see me code in this manner:

```
void function(int variable)
   {
   int i;

   for (i=0; i<N; i++)
      {
      use index variable i in routine
      }
   }
```

for strict C code compiled with gcc, or

```
void function(int variable)
   {
   for (int i=0; i<N; i++)
      {
      use index variable i in routine
      }
   }
```

for code compiled with g++.

The index variable i only exists within this function. Once you leave this function the variable i no longer exists because it has gone out of scope.

On the other hand, if I had written the code:

```
#include <stdio.h>

#define N 20

int i;

int main(void)
   {
   for (i=0; i<N; i++)
      {
      // use the index variable i in a routine
      printf("Index equals %d\n", i);
      }
   return 0;
   }
```

the variable i is global to all the code within the file xyz.c The same index variable can be used throughout xyz.c using only one memory space. In large programs this causes problems, since you could inadvertently create an error which is difficult to find. It is best to keep the scope of index variables as small as possible, limiting their lifetime to within a function or a portion of a function.

While you are in a creative frenzy you can use global variables willy nilly, but when it comes time to clean things up, and make your code look professional, it is best to limit their use to as few as possible. None is considered best but I find a few OK, as long as they are defined close to the top of the file.

A variable can be defined within a for loop so it only exists for a short while. If you define a variable outside of the main() function it is global to the file it is in. It can be used by any function in that file. If you place that definition inside the main() function then it is accessible only within the main() environment. Or you could declare a variable EXTERN which means it was defined elsewhere, but available for use in all of your code files. If you define your variable in a header file, which is called project wide, those variables are global to the entire project. If you call the header file from only one source file, those variables are available to that file alone.

Scope ranges:

- inside of a loop within a function
- limited to inside of a function
- available to all of main() but not the rest of the file
- global to a file
- global to the entire application
- external variable definitions and usage

Software engineers try to balance two parameters: the coupling of functions and their cohesion. Coupling refers to how many variables, or structures, are affected by the function. This includes the subject of global variables. If you can write a function which only affects its input values, and internal variables you will have the least coupling. If your function affects variables in the main program, then its coupling coefficient is higher. Cohesion is the term for the complexity of a function. Does it accomplish only one task or does it perform multiple tasks under input control? A simple screwdriver or a multi-tool? Low coupling and high cohesion is the goal. If you can make your functions simple and concise, you can use them in more places. I find high cohesion is something which comes naturally to me. I rarely write a function to do two things unless absolutely necessary. Normally, I will write one function then clone and edit new functions, which are very similar. I also don't like to pass a lot of variables, while allowing the data structure to be available globally. I don't abhor global

variables, as I have been taught. I find a few of them can often make the code much simpler. I comment them in blocks so they are quite visible as I read the code.

Languages and their effects on me

FORTRAN taught me how to use index variables i, j, k for traversing one, two, and three dimensional space.

BASIC showed me how to code with an interpreter, and the limitations of doing so.

Forth influenced my naming conventions, introduced me to decomposition, and writing functions which fit on one page.

Pascal taught me how to format my code, using tabs & white space, so it is readable years in the future.

Assembly language forced me to write comments for almost every line. That has saved me more than once.

The Windows API showed me their Hungarian notation variable naming scheme; I was not convinced.

But they did push me toward camel code naming schemes influenced, of course, by Forth.

C got me looking at decomposition, and function creation, far more than I had done previously.

PERL beat regular expressions into me so they were important in my every day work.

C++ had me thinking in objects. How can I use a lot of little gadgets to solve a problem?

The many other languages I have learned have helped me know more about programming and about computer hardware. So you see, it really helps to learn more than one language. Once you do so you'll find yourself examining your problem and the data, then choosing which language fits best. Which language will be fastest? Which language will use the least amount of memory? Not only the fastest running code but the one which took the least amount of time to write, and which one is easiest to maintain and supplement? Dad always told me to get as many references on a subject as possible to not limit my view of the subject. Seeing something from all sides is much better than a single view of it. Having different ways to craft the solution is even better.