Programming Patterns Think Globally, Act Locally

by Nabeel Azar

Global variables may look like a good solution to a programming problem, but changing your programming patterns to avoid them can be a better one in the long run.

The problem isn't that global variables are bad, but that there are usually better alternatives. Some of the drawbacks of using globals are:

- Naming conflicts and unintended data overwrites can occur because there is only one global workspace.
- Code flow can be difficult to follow if globals are used frequently.
- Code using global variables doesn't scale well or integrate safely into other applications.

This article discusses several situations in which globals are used and suggests alternatives that can make your programming clearer and more robust.

Function Functions

Globals are sometimes unnecessarily used with the MATLAB function-functions (functions that take other functions as parameters). In these cases, globals are used to pass optional parameters to the function parameter. For example, let's assume we want to minimize a function of x for several different values of p. With global variables, we can write:

```
function y = fun(x)
global p
y = 100*(x(2)-x(1)^2)^2+(p-x(1))^2

>> global p
>> p = 1;fminsearch(@fun,[1 1])
>> p = 2;fminsearch(@fun,[1 1])
```

While this works, it's not an ideal solution. One problem is that it's hard to determine the value of p by looking at the call to `fminsearch`. In a complicated program, the call to `fminsearch` may not be adjacent to the parameter definition, making it difficult to follow your code's behavior or realize that additional parameters are being used. To make matters worse, other functions may place variables named p in the global workspace. Because there is only one global workspace, the value of p could change without your knowing it, causing problems that are difficult to track down.

Instead of using globals, provide the value of `p` as an additional input argument to `fun`, and pass `p` to fun through `fminsearch`:

```
>> help fminsearch
X = FMINSEARCH(FUN,X0,OPTIONS,P1,P2,...) provides for additional
arguments which are passed to the objective function, F=feval
(FUN,X,P1,P2,...).
```

```
function y = fun(x,p) y = 100*(x(2)-x(1)^2)^2+(p-x(1))^2;
```

```
>> fminsearch(@fun,[1 1],[],1)
>> fminsearch(@fun,[1 1],[],2)
```

This is shorter and cleaner than using global variables. And by glancing at the function definition line, we can see that `fun` is a function of `x` and `p`.

To support this programming pattern, you should let users of your function-functions pass additional parameters to their function arguments, in the same way that `fminsearch` does, by using `varargin` to hold the extra inputs. For example:

```
function output = yourfunction(fun,varargin)
output = feval(fun,varargin{:});
```

```
>> yourfunction(@someFunction,[1 1],2)
```

If you haven't seen the use of {:} to generate a comma-separated list, take a look at the Spring 2001 *News and Notes* article at:
www.mathworks.com/company/newsletter/spring01/patterns.shtml

Growing Inputs—Doing More Than You Thought You Would!

During the development process, it's common to write a function with only one or two inputs and later to find that you need to pass in additional arguments. You could change the function argument list and add the extra parameters. For example, `fun(x)` becomes `fun(x,p1)`, and later can grow to become `fun(x,p1,p2,p3,p4)`. The problem is that you'll need to modify every call to this function to provide values for `p1, p2, p3,` and `p4`. It would be more convenient to define argument defaults in one place and not worry about changing preexisting function calls.

Global variables are sometimes used to provide these extra definitions without changing a function's argument list. This causes problems similar to those seen when using globals with function-functions, such as code becoming unclear and difficult to follow.

One alternative technique is to specify a structure as your input parameter. You can add fields to this structure easily, without changing the function's argument list. Additionally, displaying the structure shows the values of all the variables being passed to your function. This is a significant advantage over global variables, which are difficult to isolate and view:

```
A = fun(myStruct)
% Adjust p3 and leave off the semicolon to print
values myStruct.p3 = 10
y = fun(myStruct);
```

Another alternative uses comma-separated lists. Consider a function that can be called with a varying number of inputs:

```
% heatmodel(sz)
% heatmodel(sz,normalTemp)
% heatmodel(sz,normalTemp,hotTemp)
```

To handle the variable length argument list, you can use a programming pattern that looks like this:

```
function heatmodel(varargin)
defaultValues = {100,72,350};
nonemptyIdx = ~cellfun(‘isempty’,varargin);
defaultValues(nonemptyIdx) = varargin(nonemptyIdx);
[sz normalTemp hotTemp] = deal(defaultValues{:});
```

With this code pattern, you can easily add and remove input arguments by modifying the `defaultValues` cell array and the left-hand side outputs of the deal function. This lets you change the function’s argument list without adversely affecting routines that call your function, and allows you to use [ ] for default values.

Sharing Data and Preserving State

MATLAB programmers typically uset globals to maintain state or pass data between separate workspaces (such as between subfunctions).

Because the global workspace is shared, global variables can be modified unexpectedly by programs you’ve written being called indirectly. Debugging these problems isn’t easy, and so it is best to avoid globals in these situations.

Relying on the global workspace also makes it difficult to run non-modal applications or multiple instances of your program (this problem is common with GUIs). Furthermore, calling CLEAR from a function or script can clear variables in the global workspace. This means that if you’re designing a nonmodal application, other code may alter your global data.

One way to handle this is to store the data inside Handle Graphics objects or inside the “root” object (the root object’s handle is always `0`). The `guidata, setappdata`, and `getappdata` functions provide a safe interface for storing and accessing application-specific data in this way. But if you’re doing GUI programming, you may be able to avoid the need to explicitly pass these parameters in the first place: MATLAB automatically passes commonly used callback parameters to callback functions when those callback functions are specified using function handles. The “Programming GUIs” section of MATLAB help has more information on and examples of this (search for “programming guis” in the MATLAB Help menu).

Manipulating Large Variables and Accessing Commonly-Used Parameters

Sometimes large variables are defined as globals to avoid the overhead associated with passing them between functions as parameters. The MATLAB language behaves in a pass-by-value fashion, making this idea seem appealing.

However, many MATLAB language optimizations avoid the creation of unnecessary copies, and so, while a pass-by-value behavior is what you see as a programmer, MATLAB will pass parameters by reference if they aren’t modified in a function (read-only parameters). This avoids the need to copy function input parameters and negates the need for global variables.

If you need to write to a large variable set, you can create a function that controls access to that variable, which is stored persistently rather than globally. This saves memory and benefits from the safer scoping and controlled access properties of persistent data storage. A similar technique can store commonly used parameters that are accessed from multiple functions. The code below stores parameters in a persistent variable and provides access to them through an accessor function:

```
function y = parameter(varargin)
persistent S
if isempty S
S.parameter1 = ...;
S.parameter2 = ...;
end
switch nargin
case 0 % Return the entire structure
y = S;
case 1 % Return the parameter requested
y = S.(varargin{1});
% For MATLAB 6.1 and earlier, use
% y = getfield(S,varargin{1});
case 2 % Set the parameter requested
S.(varargin{1}) = varargin{2};
% For MATLAB 6.1 and earlier, use
% S = setfield(S,varargin{1},varargin{2});
end
```

This approach gives you controlled access to parameters in a safely scoped way, and is useful when accessing commonly used parameters from multiple locations.

Are There Uses for Global Variables?

There are cases where using global variables can be safe, such as the `tic` and `toc` functions. These functions store and access the state of a timer using a global variable. Because the `tic` and `toc` code is straightforward, using globals won't increase the overall complexity of the code. `tic` and `toc` also don't use an accessor function because doing so would add overhead to their run time, making their results less accurate.

Apart from simple cases like `tic/toc`, it's generally best to avoid global variables. For simple code or code that you don't plan to distribute widely, globals can be convenient, provided you're well aware of their drawbacks. Otherwise, using globals is dangerous and you'll often find that it's better to think globally but act locally.

Next Article

related topics:

Using MathWorks Products For... | Training | MATLAB Based Books | Third-Party Products