Fallacy of global variables, and how to avoid them

If you are still new to programming, the journey from college grad to employed software developer is a fascinating one full of learnings. With this transition, you learn new design patterns and methods while discarding old, outdated or even wrong practices learnt in college.

One of the first problems you will notice as you code is the danger of using global variables. The more of those that you use, the tougher it is to manage them and more bugs that will inadvertently creep in.

This is due to many reasons-

- Occurrence of naming conflicts, when two or more variables intended to be separate share the same name.
- Global variables are anti information-hiding, aka encapsulation. Encapsulation is a good practice, particularly when writing libraries.
- It is tough to determine (even with testing) the flow of the program, as multiple files and functions can change the state of your variable. Unintended side effects can be particularly devastating for your code as they can be tough to detect or may appear only under edge cases.
- Global variables hog memory.

Ways to avoid them

1. **Getters and setters.**

Although C is not an object-oriented language, there are many ways to achieve encapsulation, one of which is *getters* and *setters*. Getters are functions that provide access to retrieve the value of a private variable, while setters are functions used to modify the value of a private variable. By encapsulating the state of a variable within a structure or module, getters and setters allow controlled access and manipulation without directly accessing the variable.

How exactly would you achieve private variables and getters/setters? Here is an example:

Abhijith CV
Senior Engineer, DTRI


```
#include "myfile.h"

int main ( )

     {

          set_pi(3.1416);

          float pivalue= get_pi();

          printf("%f", pivalue());

          return 0;

     }
```


```
static float pi; // works as private variable
void set_pi (float num)
{
        pi = num;
}
float get_pi (void)
{
        return pi;
}
```

Voila, now you cannot access or modify the variable ***pi*** from any external file as you have restricted the data variable ***pi*** to file-scope. Only your getter and setter functions have access and you need not keep track of any global variables.

2. Opaque structures.

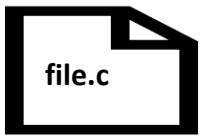
Opaque structures are abstract data types, which are yet another effective way to avoid global variables. An opaque structure hides the implementation details of a data structure, preventing direct access to its members. Instead, the structure is only accessed through functions defined within the same module or library.

```
// file.h

typedef struct opaquedata *op;
op create_op(int value, const char* string);

//main.c

op op1 = create_op(100, "DTRI");
```




```
struct opaquedata {

int val;

char string[10];

};

op create_op(int value, const char* string)

{

    op new_op = malloc(sizeof(struct opaquedata));

    new_op->val = value;

    strncpy(new_op->string, string, 9);

    new_op->string[9] = '\0';

    return new_op;

}
```

3. Passing local variables around.

Passing local variables as parameters to functions and using return values is another simple and great way to avoid global variables in C. By limiting their scope and passing them explicitly, we reduce the reliance on global states that can be altered anywhere and anyhow. If needed, use *static* for a local variable to hold its value throughout the entirety of the program.