Definition of Global Variables for Programming Languages

Dr. Ali Abdelaziz Musa

System Engineer, Saudi Telecom Company (STC)

3135 Al-Khaldia Towers, Al-Khazan Street, Riyadh 11417, Saudi Arabia

Email: amusa07@yahoo.com, aamusa@stc.com.sa

ABSTRACT

The paper addresses two issues: (1) is there any standard to be followed on the definition of Global Variables for Programming Languages? (2) is the new approach will maintain the same philosophy of assignment techniques adopted by the legacy scope for Local/Global variables? Answers to these kinds of questions and more will be addressed in this paper. The paper proposes a solution based on the implementation of a proper use of Global Variables, which can be accessed by different modules, procedure, functions and main program. The proposed solutions are explained using the definition of Global Variables in C Language as a template. The new approach suggested a list of options for the declaration of Global Variables. 1) Will they take different data-type style than the style applied on Local Variables, 2) with these improvements and by adding new functionalities, will lead the programmers to have good capability in writing their procedures and functions and in a proper way and they can distribute them to different files or include them into one file with the main program. Finally, the fundamental objectives of this study are to (1) provide easy-to-use manipulation of Global Variables in C language, (2) achieve high performance via the

execution using the new mechanism, and (3) facilitate the execution of the new mechanism across a wide range of platforms using C language.

1. INTRODUCTION

Global Variable is a variable that is known throughout the program and its procedure or functions. Local variables are used most often in user-defined functions. If you try to call a local variable outside of its specific function, you will either get an error or the computer will evaluate the Global Variable instead (if the local variable has the same name as a Global Variable). The scope of a Global Variable is anywhere at any time. They exist outside of any functions regardless of whether they are called in one. If a local variable is declared inside a procedure with the same name as the Global Variable, the Global Variable is overridden.

Hence, adding Global Variables to logic programming can solve some common problems of reliability and programmer productivity in large logic programs. This addition has the consequences that conjunction is neither commutative nor absorptive, but we show that the practical loss is small.

This paper describes the experiment we carried out and reports our findings and conclusions. Section (2) provides background to Global Variables that describes the manipulation of global variables by C languages, reports its implementation Section (3) introduce the new approach compares to different approaches for defining Global variables and their assignment and concludes and summarizes our findings.

Overview

It is not our intention in this paper to debate the relative merits of procedural (e.g. Pascal or C), functional (e.g. Lisp), declarative languages (e.g. PROLOG), and parallel language (e.g. NESL). Our intention, rather, is to urge people to improve the definition of Global Variable and their assignment.

Improving the definition of Global Variable and their assignment can achieves great enhancement results such as memory organization, program coding, program clarity, reducing memory allocation, reduce the size of the generated machine code, remove redundancy when calling procedure or functions, ease the interlink between procedures or functions within the program and performance. These expected augmentations, reflected the need of computer programmers to write powerful programs when using C programming languages.

Findings

Some analyst believes that it is sometimes advisable that Global Variables should be avoided when possible. However, sometimes you really need them or the program is so small that it isn't dangerous to use them. They stated that the intensive use of data exchange, through Global Variables makes the reading and understanding of the source code more difficult. And which is not very consistent with the philosophy of abstract data types that is used with the Object Oriented languages.

Some of the analyst found out that the best way of using Global Variables in traditional languages like C is as if throwing all your data into a big structure and accessing the data there is not protecting your data or making it easier to read or understand your program. A solution is to writing access routines but, the tendency is to write one big structure to hold everything, that way you only need one access routine to get to all the data. Data should be kept local to the modules using it, and multiple access routines should be written to get to the data in any module. They said also, the indirect links created between different parts of the program are confusing and hard to trace, and a constant source of nasty bugs.

Some analysts encourage the use of fewer Global Variables which would increase the comprehensibility of the code. One of the big reasons to use Global Variables is to pass default values to your programs. Global Variables or an external file are good ways to save values from one run of your program to the next.

Some analysts found that globalization (i.e. replacing function parameters by Global Variables) is appropriate to reduce the time and space of the stack (or heap) allocation of function parameters when possible (D. Peter Sestoft, October 1988).

The global name space in Object-oriented programming language (OOP) like C++ is the class hierarchy and so one cannot create a variable outside of a class. This removes the possibility of side effects occurring on a system-wide basis due to some change in the state of a Global Variable. It is extremely difficult (if not impossible) in object-oriented languages to ensure that a Global Variable is changed in a consistent and controlled manner (Herbert Schildt 1992).

But, Java does allow a modified form of the Global Variable called static variables. Java language used the interfaces to import shared constants into multiple classes (Herbert Schildt 2000),

2. Manipulating Global Variables in C Programming Languages

Why we have chosen C language to be the issue of our study, is based on the fact that the creation of C was a direct result of the need fro structured, efficient, high-level language that could replace assembly code when creating system program. Also, the history of C is inextricably linked with the history of UNIX. The UNIX operating system is itself written in C, as are the majority of utility programs which come with UNIX and to the best of my knowledge a C compiler comes with every distribution of UNIX, whereas it is harder to get compilers for other languages under UNIX. For a UNIX user who wanted to do any programming, a competence in C was almost essential.

Since C language has built up a large user base also, C has remained widespread once, it develops an unstoppable momentum. The most common reasons are 1) easy availability of inexpensive compilers; 2) extensive subroutine libraries and tools; 3) everyone else uses it (the ready availability of compilers, libraries, and support tools is, of course, a direct consequence of the large number of users); 4) portability (available in all hardware); 5) saves keystrokes since C programmers can be able to write a statement like ****p++^=q++=*r——s**; 7) faster compared to more structured language.

Defining Global and Local Variables in C Programs

The topic covered in this section and the subsequent subsections presented the manipulation of the global variables in C language. The topic is a collection taken from different readings and books published by Brian W. Kernighan and Dennis M. Ritchie specifically the one of (1988), and Herbert Schildt (1992), according to their books any variable declared on the top of the main() function (section) of the program is treated as Global Variables. It is also possible to pre-initialize Global Variables using the = operator for assignment. Following is an example of the source code of a simple C program that describe Local and Global Variables:

```
#include <stdio.h>              /* Standard I/O library header that contains
                                   macros and function declarations such as
                                    printf used below */
#include <math.h>               /* Standard math library header that
                                   contains macros and function declarations
                                   such as COS used below */
#define NUM 46.0                /* Preprocessor directive */
double x = 45.0;                /* External variable definitions */
double y = NUM;                 /* End of Global Variables */
int main(void)                  /* Function definition for main function */
{
    double z;                   /* Local variable definitions */
    double w;
    z = cos(x);                 /* cos is declared in math.h as double
                                   cos(double arg)    */
    w = cos(y);
    printf ("cosine of x is %f\n", z);      /* Print cosine of x */
    printf ("cosine of y is %f\n", w);    /* Print cosine of y */
    return 0;
}                                                                    (1)
```

The above source program defines *main()* and declares a reference to the functions *cos* and *printf*. The program defines the Global Variables x and y, initializes them, and declares two local variables z and w.

2.1.1. Writing Smaller C-Programs

In the case of smaller C-program means that to include both main function of the program and its associated modules into a single source file. Since functions are blocks, you can not share variables that are declared inside a function. They are called local variables. Instead *Global Variables*, you have to declare them **in the beginning** of the file. Now, they are weak cycle, visible for **all** functions in your program (with all consequences). The memory for those variables is allocated just before the ``main'' function is started. Still Global Variables are defined above main() in the following way:-

```
Int globalInt;
int func(int t);                    → Global Variables Area
int main()
{
        int tt;                  → Local Variable
        globalInt = 10;          →  Global Variable updates
        tt = func(10);
}

int func(int t)
{
       return(t == globalInt);
}                                                          (2)
```

Un-initialized external-scope variables in C, without the *extern* keyword, are normally implemented similarly to FORTRAN Common blocks. The variables are treated as imported symbols, but are allocated automatically by the linker if no definition for the symbol is found in the program. As in (2), it is also possible to pre-initialize Global Variables using the = operator for assignment.

2.1.2. Writing Larger C-Programs

When writing large C programs we should divide programs up into modules. These would be separate source files. *main()* would be in one file and not ne ccessary alone, main.c say, the others will contain functions.

We can create our own library of functions by writing a *suite* of subroutines in one (or more) modules. In fact modules can be shared (i.e. Global Variables) amongst many programs by simply including the modules at compilation as we will see shortly. There are many advantages to this approach:

1 The modules will naturally divide into common groups of functions.

2 We can compile each module separately and link in compiled modules (more on this later).

2.1.3. Header Files

If we adopt a modular approach then we will naturally want to keep variable definitions, functions prototypes *etc.* with each module. However what if several modules need to share (i.e. Global Variables) such definitions? It is best to centralize the definitions in one file and share this file amongst the modules. Such a file is usually called a **header file**. Convention states that these files have a .h suffix. We have met standard library header files already *e.g.*: as in (1) (#include <stdio.h>) that represents a Standard ***I/O library header*** and Standard (#include <math.h>) ***math library header*** or we can define our own header files and include them in our programs such as #include ``my_head.h''.

NOTE: Header files usually ONLY contain definitions of data types, function prototypes and C pre-processor commands. Consider the following simple example of a large program presented in Figure 1:

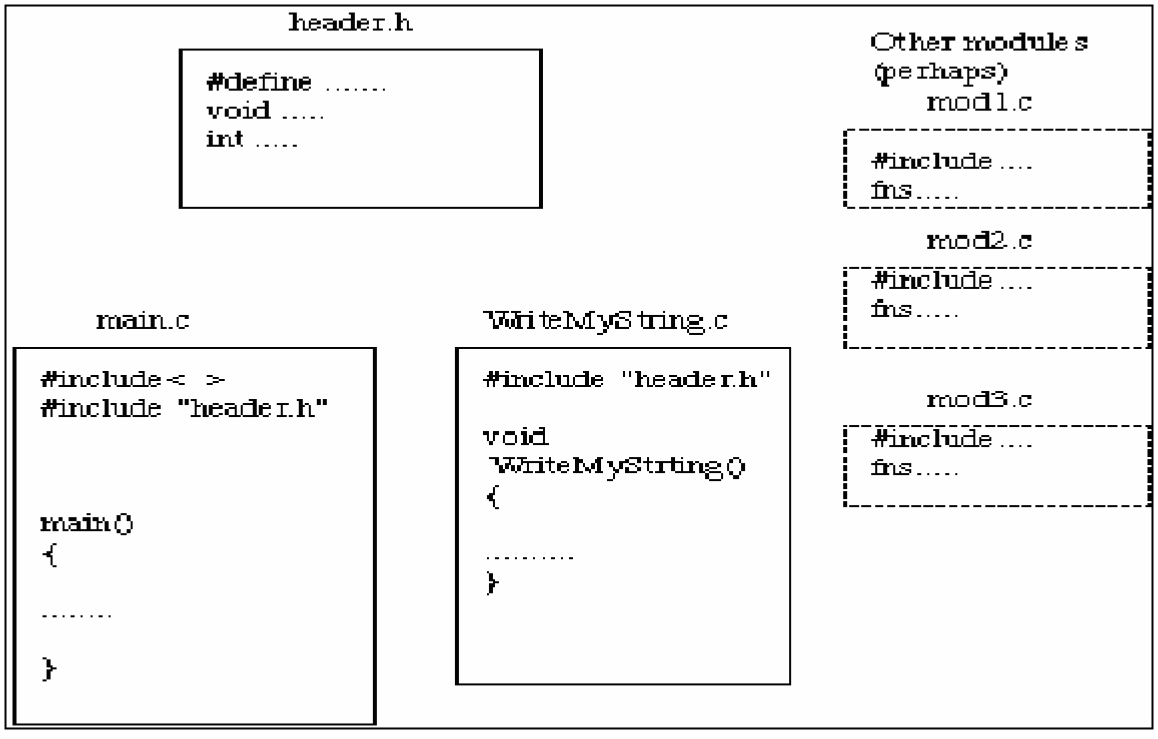

Figure 1

The above figure shows a modular structure of a C program. The full listings main.c, WriteMyString.c and header.h as follows:

```
/***main.c file *****/

#include "header.h"
#include <stdio.h>
char       *AnotherString = "Hello Everyone";

main()
{
    printf("Running...\n");
    /**  Call WriteMyString() - defined in another file  **/
    WriteMyString(MY_STRING);
    printf("Finished.\n");
}

/**  WriteMyString.c  file ***/

extern char      *AnotherString;        /*  Global Variable inside a
                                              Function or  Procedure */
void WriteMyString(ThisString)
char       *ThisString;
{
    printf("%s\n", ThisString);
    printf("Global Variable = %s\n", AnotherString);
}

/**  header.h file ***/

#define MY_STRING "Hello World"
 void WriteMyString();                                              (3)
```

We would usually compile each module separately (more later). Some modules have a "#include ``header.h" that share common definitions (i.e. Global Variables). Some, like *main.c*, also include standard header files also, main calls the function WriteMyString() which is in *WriteMyString.c* module. The function prototype void for WriteMyString is defined in *Header.h*

NOTE: that in general we must resolve a trade-off between having a desire for each .c module to have access to the information it needs solely for its job and the practical reality of maintaining lots of header files. Up to some moderate program size it is probably best to one or two header files that share more than one module definitions.

2.1.4. External Variables and Functions

``Internal'' implies arguments and functions are defined inside functions – **Local** ``External'' variables are defined outside of functions -- they are potentially available to the whole program (Global) but **NOT necessarily**. External variables are always permanent.

NOTE: in C, all function definitions are external. We CANNOT have embedded function declarations like in PASCAL.

2.1.5. Scope of Externals

An external variable (or function) is not always totally global. C applies the following rule: The scope of an external variable (or function) begins at its point of declaration and lasts to the end of the file (module) it is declared in. Consider the following:

```
main()
{ .... }
int what_scope;
float end_of_scope[10]
void what_global()
 { .... }
 char alone;
float fn()
{ .... }                                                    (3)
```

main cannot see what scope or end-of-scope but the functions what-global and fn can ONLY fn can see. This is also one of the reasons why we should *prototype* functions before the body of code *etc.* is given. So here main will not know anything about the functions what global and fn. What global does not know about fn but fn knows about what-global since it is declared above.

NOTE: The other reason we *prototyped* functions is that, some checking can be done parameters passed to functions.

If we need to refer to an external variable before it is declared or if it is defined in another module we must declare it as an extern variable. *e.g.*

```
extern int what-global;                                              (4)
```

so returning to the modular example. We have a global string variable AnotherString declared in main.c and shared with WriteMyString.c where it is declared extern. **BEWARE** the extern prefix is a *declaration* NOT a *definition*. (*i.e.* **NO STORAGE)** is set aside in memory for an extern variable it is just an announcement of the property of a variable.

The actual variable must only be defined once in the whole program you can have as many extern declarations as needed. Array sizes must obviously be given with declarations but are not needed with extern declarations. *e.g.*:

```
main.c:
  int arr[100]:
file.c:
     extern int arr[];                                              (10)
```

2.1.6. Shared Memory

In single process programming, different modules within the single process can communicate with each other using *Global Variables*, functions calls arguments and the result passed back and forth between functions and their callers. Where as for two processes to communicate with each other, they must both agree to it, and the operating system must provide some facilities for Inter-process communication IPC (a method innovated by UNIX).

So the interaction is occurred using IPC as a method used to communicate with each other. Indeed, there are many instances where IPC can and should be used between processes on single computer system. There are a few methods which can accomplish this task, such as Pipes, Signals, Message Queues, Semaphores, Shared Memory, and Sockets (W. Richard Stevens 1990). Using IPC mechanism, two process are run in a unit may reference common data that has the global attribute shared in memory.

2.1.7. Global Variables in JAVA

This section is a collection of various reading on Java language and specifically the book of JAVa-2 4th edition written by (Herbert Schildt 2000). The reason for introducing Java in this topic because Java is related to C++ which is a direct descendent of C. Most of the character in Java is inherited from these two languages. From C Java derives its syntax. Many of the Java's object-oriented features were influenced by C++.
In Java, the global name space is the class hierarchy and so one cannot create a variable outside of a class. This removes the possibility of side effects occurring on a system-wide basis due to some change in the state of a Global Variable. For example, it is extremely difficult (if not impossible) to ensure that a Global Variable is changed in a consistent and controlled manner. Java does allow a modified form of the Global Variable called static variables.

You can use interfaces to import shared constants (global) into multiple classes by simply declaring an interface that contains variables which initialized to the desired values. When you include that interface in a class (that is, when you "implement" the interface), all of those variable names will be in a scope of constants. This similar to using a header file in C/C++ to create a large number of **#define** constants or **constant** declarations.

The next example using this technique to implement an automated "decision maker". Also, the following example illustrates how the interface class may be accessed:

```
import java.util.Random

interface globalVariables
{
        int NO       = 0;
        int YES      = 1;
        int MAYBE = 2;
        int LATER  = 3;
        int SOON    = 4;
        int NEVER  = 5;
}
class Questions implements globalVariables
{
        Random rad = new Random();
        int ask()
        {
             int prob = (int) (100 * rand.nextDouble();
             if (prob <  30)
                return NO;              // Declared and initialized in the
                                        // interface class
             else if (prob <  60)
                return YES;              // Declared and initialized in the
                                         // interface class
             else if (prob <  75)
                return LATER;        // Declared and initialized in the
```

```
                                    //    interface class
        else if (prob <  98)
           return SOON;             // Declared and initialized in the
                                    //      interface class
        else
           return NEVER;            // Declared and initialized in the
                                    // interface class
}


class AskMe implements globalVariables
{
      static void answer(int result)
      {

      switch(result)
      {
                  case NO:          // Declared and initialized in the
                                    // interface class

               System.out.println("NO");
               Break;
                  case YES:          // Declared and initialized in the
                                     // interface class
               System.out.println("YES");
               Break;
                  case MAYBE:       //Declared & initialized in the
                                    // interface class
                System.out.println("MAYBE");
                Break;
                  case LATER:       //Declared & initialized in the
                                    // interface class
               System.out.println("LATER");
               Break;
                  case SOON:         // Declared & initialized in the
                                     // interface class
               System.out.println("SOON");
               Break;
                  case NEVER:        //Declared & initialized in the
                                    // interface class
               System.out.println("NEVER");
               Break;
       }
}

public static main (String args[])
```

```
{
        Question q = new Question();
        answer(q,ask);
        answer(q,ask);
        answer(q,ask);
        answer(q,ask);
}

}                                                              (11)
```

In the above sample program, the two classes, **Question** and **AskMe,** both implement the **globalVariabes** interface where **NO, YES, MAYBE, SOON, LATER,** and **NEVER** are defined. Inside each class, the code refers to these constants as if each class had defined or inherited them directly. Here is the output of a sample run of this program. Note that the result each time it is run (i.e. Later, Soon, No, and Yes respectively).

3. CONCLUSIONS

Today it is quite common to see the tendency of future changes is the real need to use Global Variables in a functional languages, parallel languages and single-threaded or multi-threaded programming techniques. This tendency forces most of the compiler designer like C to find solution for the definitions and the scope of the variables to meet this requirement. Also, in the innovation of Object-Oriented Programming languages (OOP) including C++, and Java which encourage very much the idea of organizing the access, sharing and protecting of data among different objects shared in the program. Each self-contained object in surrounded by clouds of protection, so that to access the data contained in an object you need to define a proper method accepted by the object.

These modifications to the way we access data allow us to create powerful abstractions, and then control the way data is accessed in a global way.

Based on the previous presentation, we have a very good understanding of how we declared, defined the scope and a memory allocation for Local and Global Variable in C language, in addition to IPC. The previous discussion highlights many weakness points that encouraged us to employ a proper mechanism for the utilization of memory to achieve better performance results through the use of reliable techniques to minimize variable definitions and their assignments. Also, to design tools that will help us in drawing better techniques to declare Variables after they have been tagged as Global Variables and in defining a proper method for assigning their values.

What we concluded for C language will apply on C++, since C and C++ share common features in defining Global Variables, variables has to be declared in the top of the main program functions and subprograms belong to the main program. However, C++ has additional features of defining data as global, a variable or functions will be defined in a public class block that can be accessed by the following modules of the program. Hence, both C and C++ offer excessive flexibility in the scope of variables.

Our goal is to put forward a mechanism that improves performance of applications with mixed-modules (inter-procedure) nature while improving programmer productivity. The paper will only suggest how the current Global Variables accessing infrastructure can be augmented by another accessing method to deal with the question of how to provide better end-to-end integration for relationship of different modules in a C program and how to improve the usability threshold for Global Variables. Unlike previous handling of global shared variable, whose lifetime begins after its declaration and ends at the end of the file, an automatic shared variable's lifetime ends at the end of the enclosing scope. We must carefully examine the execution of modules that use automatic shared variables and ensure that those variables are not out of scope when they are in use.

3.1. NEW approach for Defining Global Variables

Apparently clear that there always problem with Sharing Module Approach, that is if we have Global Variables declared and instantiated in one module how can we pass knowledge of this to other modules. Based on our findings, we could pass values as parameters to functions, BUT:

- This can be laborious if we pass the same parameters to many functions and / or if there are long argument lists involved.
- Very large arrays and structures are difficult to store them locally in memory which may cause problems with stack.

A solution to this kind of problems and other more are addressed in this paper. The paper proposes a solution for the utilization of Global Variables which will replace the method being followed in the previous C compilers as illustrated in Section 3. The paper exploited the approaches adopted by different languages such as FORTRAN, COBOL, PROLOG (B- PROLOG and GNU-PROLOG) and LUNA.

Moreover, the paper proposes a solution that utilize the concept of Common blocks adopted by FORTRAN-specific, a mechanism used for indirectly passing variables to procedures, i.e. not by procedure arguments. The common block mechanism is a refinement of the Global Variables used in a C programming language, as you can control which procedures have access to a common block and which will not (by including or not the common statement in these procedures).

3.1.1. Using the keyword Modifier GLOBAL to identify Global Variables

In C sense, a global name can be accessed by the program defining it and by any other program contained directly or indirectly in that program. If the program is detached into modules contained into several files then Global Variable defined by the program are valid for all modules that are included the header file defining them.

In the new approach we introduce a new keyword modifier called GLOBAL, the new keyword modifier similar to a type qualifier, to distinguish between Global Variable and Local. The GLOBAL new keyword modifier will be placed to the left of the data-type of the identifier in any module of the program. When the new keyword modifier placed will instruct the compiler to consider the variable as a Global Variable.

```
Int var;                              → Local Variable
GLOBAL int var                        →  Defining Global Variable in the
                                          Declaration statement. Global
                                          Variable valid to be defined
                                           at main Program, procedure or
                                           functions and  for any  type of
                                           variables (char, struct, , etc)

GLOBAL var=initVal                    → Assigning Global Variable

GLOBAL int funct(GLOBAL par1)   →  Defining function as global
{
     ….
    return(retVal)
}                                                                  (12)
```

You can also put the GLOBAL keyword modifier to the left of the identifier variable declared in #DEFINE in any module of the program will instruct the compiler to consider the variable preceded #DEFINE as a global variable. I suggested that the values of the global variables to be stored in memory common blocks in a sequential manner separated by special character (i.e. octal 253).

```
#DEFINE   var 100;                         → Local Variable
#DEFINE  GLOBAL var 100;                   → Defining Global Variable
                                             in #DEFINE directive       (13)
```

The compiler will allocate special area in the static memory common blocks to keep the variables. The approach provides a very simple and powerful way to assign and read Global Variables.

Note: This approach requested the compiler at the start of compilation to do full scan on the program to find out all the variables with GLOBAL keyword modifier and put them in the proper common global variables memory storage area. So that for the modules of the program to distinguish between a local variable and a global one.

There is another suggestion that when you want to read a Global Variable you must add a prefix to the variable (i.e. global.).

- This will give the user flexibility in distinguishing between variables.
- This give user opportunity to declare the variable with the same name as local.

The following example illustrated the new assignment of the global variable:

```
global.var=10;
if (global.var != 0
{.....}                                                    (14)
```

The computer will extract the value for the global variable from memory area allocated for global variables based on the name specified as a suffix to global.

3.1.1.1. Using malloc()

To minimize the work for the compiler's designer, I have another suggestion that simplified the previous approach (i.e. putting the **GLOBAL** keyword modifier). But, in this time the new keyword modifier to be used with **malloc()** built-in function only. The way to implement this mechanism is to add the concept of keyword modifier **GLOBAL** to the left **malloc()** built function. Once the keyword modifier **GLOBAL** is added to the function of **malloc()** then the value treated as Global Variable and will be stored in the static memory instead of the heap or dynamic memory that allocated for default usage of **malloc()** function. This will involve a new usage for **free()** function, in order to let **free()** function distinguish between freeing a variable stored in a heap or static memory. Folllowing is an example that illustrate the new technique for using **malloc()** and **free()**.

```
Gvar = GLOBAL (char *) malloc(sizeGvar);
OR
Gvar = (char *) malloc(sizeGvar) IS GLOBAL; /* Using COBOL concept*/
free(GLOBAL var)                                              (15)
```

The reason behind using the **GLOBAL** keyword modifier with **malloc()** other than the built-in function because, **malloc()** is used by the programmer he intended to dedicate to a variable in a protected location in memory with specific size.

Note: Similar functionality can be applied to **new**/**delete** in C++

3.1.1.2. Summary

In summary, when using the keyword Modifier **GLOBAL** to identify Global Variables, you need to follow the general rules in order to properly utilize the new functionality Global Variable:

1. A data-name, whose description contains as a GLOBAL clause is a global name. All data-names procedure or functions transferred to a global name and all condition-names associated with a **global** name are global names.
2. Any subprogram contained within the program that describes the global name may access a global name. The global name does not need to be described

again in the program that references it. In the case of references to identical names, the local names have priority.

3. If both the Parameters passing section and the GLOBAL clauses are specified in a description of a variable, then the Parameters passing -COUNTER where was it in his dual special register is also **global**.
4. If a global data item contains a variable-length array, then the corresponding dependency on element in the same declaration section must also be described as **global**. In the **NEXT** approach we suggested to use special character separators between global variables in memory so that to enable variable-length.
5. The data-names at the top of the program are still always implicitly global.
6. The index of an indexed table assigned to a **global** data item is also global.

3.1.1.3. Related Work

Our analysis also encouraged similar techniques used to handle shared Global Data in multitasking philosophy between different processes. We are optimistic that the work carried out by (Rung-Ji Shang, Wen-Yew Liang, Jen-Chiun Lin, Feipei Lai) of the National Taiwan University to improve utilization of distributed shared memory (DSM) using parallel language will be adopted and their parallel language will be spread. The parallel language is designed based on the C-Language and their assumption is based on the fact that even though current computer technologies have been greatly advanced in architecture and VLSI designs. Many scientific applications still can not be solved in acceptable time. A good chance to solve this problem is to explore the parallelism in these applications and to utilize the multiprocessor to solve the application. The following figure will illustrate how the DSM architectures look like.

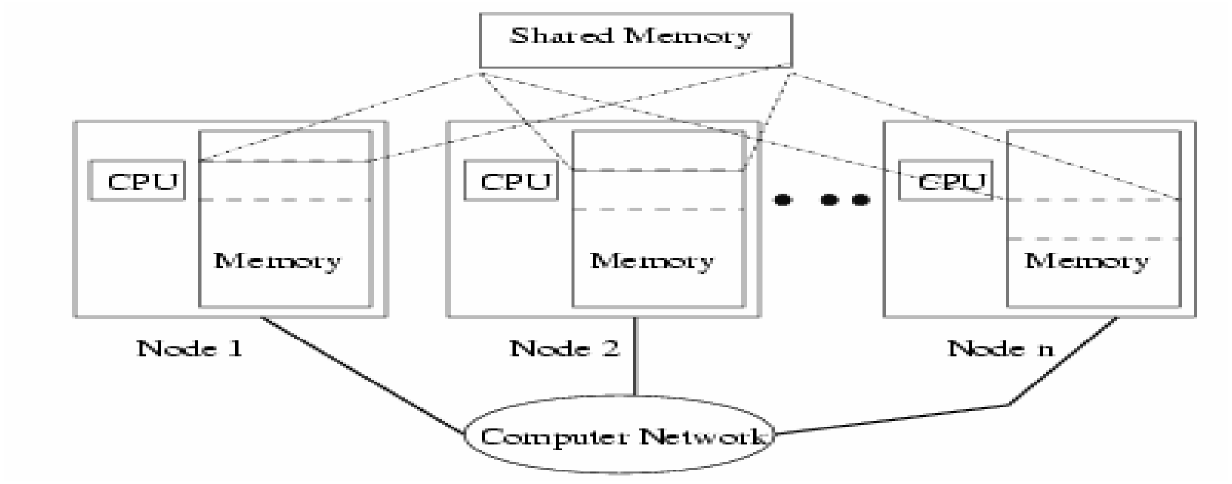

Figure 2

To make parallel program development in this environment easier, you have to design a proper solution for distributed shared memory (DSM). It provides the users with shared memory programming environment on top of physically distributed memory systems. Based on these facts they designed a parallel language that provides the users with a better shared memory environment. The new parallel language adopted a new approach for distributed shared memory (DSM). Which made the design of compiler a simple, portable, and powerful language for developing general purpose distributed applications.

The paper mainly is in favour of the Distributed systems that receive much attention because parallelism and scalability are achieved with relatively low costs. Many researches favour distributed shared memory (DSM). The systems found that the concept of shared memory can increase programmability without much performance loss. The (DSM) provides a simple, efficient, general purpose, and portable solution. For instance let us illustrate this idea by the following example:

```
/* process declaration */
 void p( shared int* psi ) process;

 shared int gi;                    →  /* global */
 void main()
 --
 --
 shared int ai;                    →  /* automatic */
 static shred int si;              →  /*static*/
 process* p1=create p( &gi );
 process* p2=create p( &ai );
 process* p3=create p( &si );
 /* p2 is waited here but p1 & p3 is not */
 /* then ai goes out of scope */                          (16)
```

we can see that since process p2 uses variable **ai** and during the execution of process p2, automatic shared variable **ai** should be alive, so the parent process must wait for process p2 to terminate and then leave the scope. Conversely, process p1 uses only global shared variable **gi**, therefore the parent process does not need to wait for p1's termination. From the above example, it should be clear that automatic shared variable should be used with care since it could lead to lower parallelism. Shared static variables declared in scopes other than the global scope have the same visibility as automatic shared variable but the same lifetime as global shared variable. Referencing these variables will not cause 'implicit barriers' as those automatic ones will do. The C programming language does not provide dynamic memory management facilities. In stead, it is achieved by using standard library calls malloc() to dynamically allocate memory and free() to dynamically free the allocated memory. If we follow this convention and provide library routines such as shared malloc() or shared free(), the semantic will be complicated since the programmers have to use different functions for shared data and none-shared data.

3.1.2. Using functions or procedure to manipulate Global Variables

Another approach that used Functions or procedure to manipulate Global Variable approaches provides a simple and powerful way to assign and read Global Variables. The approach built on the fact that since a Global Variable has a name and a value that associated to it. A name cannot be used at the same time as both a Global Variable name and a predicate name. This approach lead to implement two tasks to perform globalization 1) call a procedure to set the value of the Global Variable 2) then use a function to get the value of the allocated Global Variable.

3.1.2.1. Set Global Variable Procedure

To store a value in a Global Variable memory area, call a procedure which will tell the computer to Push the variable to a (named) static memory location and change the scope of a variable to Global Variable. The variable may have different type of format (int, char, struct, date, Boolean, etc). For instance the following example illustrated how to store a value to a Global Variable, and push it to Global Memory Area:

```
Void setGlobal (memBlock,  GvarName, GvarValue, GvarType)              (17)
```

As appear from the above example, once you call the **setGlobal()** procedure, it will push onto the memory block the value of the variable and the variable will be available for use to any modules in the program.

memBlock: Memory area called named COMMON blocks dedicated to restore the global variable. The name of the block specified by the name memBlock (if 0 specified: means store in common blocks).

GvarName : The name of the global variable that will be inserted in the name memory block used to restore the global variables.

GvarValue: The value to be assigned to the global variable (i.e. GvarName).

GvarType: The type of the global variable will be either of these types of format (integer, char, Boolean, etc).

Following is an example for using the setGlobal procedure::

```
gvarName=setGlobal(0,"gvarName","ali musa","char").
gvarName=setGlobal("blk1","gvarName",10.2,"real").          (18)
```

Note: In order to reserve a static memory topology and to enable variable-length for the values global variables. We suggested that the values for the global variables to be stored in static memory in a series manner separated by special character (i.e. octal 253 and octal 254). The first special character octal 253, separate variable from the other, where is the second special octal 254, character is to separate variable properties (i.e. GvarName, GvarValue, GvarType) within the area dedicated for the variable.

3.1.2.2. Read Global Variable

We also suggested a mechanism to read the value of a global variable, the new technique will assign the resulting value returned from the function the global variable from specific memory block location. This equivalent POP from the memory block specified by memBlock the value being stored for the given Global Variable. For instance the following example illustrated the function used to read the Global Variable:

```
getGlobal (memBlock, GvarName);                                    (19)
```

memBlock: Memory area called named COMMON blocks dedicated to restore the global variable. The name of the block specified by the name memBlock (if 0 specified: means store in common blocks.

GvarName : The name of the global variable that will be inserted in the name memory block used to restore the global variables. Following is an example of calling the function:

```
gvarName=getGlobal(0,"gvarName").                                  (20)
```

Using the concept of special character separator suggested in the **setGloba**l procedure. The **getGlobal**, function will fail and return a negative number if either of the following conditions.

- (-1) If **memBlock:** is not a proper memory block
- (-2) If **GvarName** does not exist returned (-2)
- (-3) If **GvarType** compared to the type of variable to the type allocated for the global variable if different then fails and return (-3).

The concept of special character separator specified in **setGlobal** procedure will lead to implement a lot of conversion functions, when using system functions that read memory location.

3.1.3. Summary

To put it briefly, this study proposed two options for solution that lead to improve the definition of the Global Variable 1) Using the keyword Modifier GLOBAL to change the scope of a variable to Global Variables, 2) Using functions or procedure to change the scope of a variable to Global Variables. Both options will provide a comprehensive solution that can easily be implemented by C compiler designer without causing major changes to their compilers. In addition, the study recommended the work carried out by National Taiwan University to improve utilization of distributed shared memory (DSM) as a good platform of solution for the case of parallel language.

The solution can be properly implemented, if the compiler designers introduce some tools that help in the utilization of COMMON and Named block memory. As well, to ensure better performance and precautions should be taken when dealing with COMMON and Named block memory.

REFERENCE

Brian W. Kernighan and Dennis M. Ritchie (1988), The C Programming Language, Second Edition by Prentice Hall, Inc., 1988. ,ISBN 0-13-110362-8

Rung-Ji Shang, Wen-Yew Liang, Jen-Chiun Lin , Feipei Lai, A Compiler Supporting Distributed Shared Memory System, Dept. of Computer Science and Information Engineering, Dept. of Electrical Engineering ,Dept. of Computer Science and Information Engineering and Dept. of Electrical Engineering National Taiwan University

D. Peter Sestoft (October 1988), Replacing function Parameters by Global Variables, University of Copenhagen, Denmark

Herbert Schildt (1992), Teach Yourself C++, Osborne/McGraw-Hill,ISBN 0-07-881760-9

Herbert Schildt (2000), The Complete Reference, Java 2, Update to JDK 1.3, 4th Edition, Osborne/McGraw-Hill, ISBN 0-07-213084-9

W. Richard Stevens (1990), UNIX Network Programming, ISBN 0-87692-749-5]