The Great Socialist People's Libyan Arab Jamahirya

Alfatah University

Faculty of Science

Department of Computer Science

Tripoli – Libya

"Mixed Languages Programming Technique Based on Data Variables Emigration"

A Thesis Submitted to the Department of Computer Science in Partial Fulfilment of the Requirements for the Degree of Master of Science in Computer Science

By: Rehab Abdallah Rajab Ben Abdallah

Supervisor: Dr. Musbah Mohammed Elahresh

Spring 2009

المستخـلـص

تم إعداد تقنية لتطوير البرامج بدمج لغتي C و FORTRAN كمزيج للغة برمجة. وبهذه التقنية يمكن كتابة البرنامج على شكل مقاطع ، و كل مقطع مكتوب باحدى اللغتين على ان يتم الفصل بينها داخل بيئة تطوير البرامج. داخل كل مقطع يمكن الوصول و استخدام متغيرات المقاطع الأخرى بالصور ة اللغوية للمقطع الأصلي. وهذه المتغيرات يتم استخدامها بالمقطع المضيف و الاستفادة من تمثيل البيانات بها. حيث المتغيرات المعرفة في المقطع المكتوب بلغة C يمكن استخدامها في المقاطع المكتوبة بلغة FORTRAN. تمكن هذه التقنية من تعريف المتغيرات كهياكل بيانية خاصة بلغة ما واستخدامها في مقاطع أخرى مكتوبة بلغات برمجة أخري أعطي لها الأسم " هجرة متغيرات البيانات".

تم تطوير معالج مبدئ (Preprocessor) يقوم بقراءة البرنامج المصدري الخليط ويقوم بفصل مقاطعه وتحديد المتغيرات المهاجرة وتعريفها حسب قواعد اللغة المهاجر اليها. ثم يقوم المعالج المبدئ بارسال المقاطع الى المترجمات للحصول على برامج هدفية (Object Files) ومن ثم ارسال هذه البرامج الى رابط مشترك (Common Linker) لينتج برنامج تنفيذي واحد.

Abstract

A technique for mixed – languages program is introduced to mix C and FORTRAN programming languages. This technique enables the programmer to develop programs composed of different code sections each written in either C or FORTRAN. Each section can access and use the variables decleared in other sections. The variables decleared in C sections can be used in FORTRAN sections.

This process in which a data variable is declared in a program section of a certain language and used in another section of another language is given the name "Data Variable Emigration".

A preprocessor is designed to read the mixed source program, separate the code sections in different files and determine the emigrated variables and define them by the syntax of the language they emigrated to. Then the preprocessor send the sections files to the compilers to get the object files and send these files to a common linker to produce one execution file.

TABLE OF CONTENTS

List of Figures III

List of Tables. V

List of Programs VI

List of Terms VIII

Acknowledgement IX

Chapter One: Introduction. 1

1.1 Preview 1

1.2 Previous work in mixing C and FORTRAN 77 2

1.3 Objectives of the research 3

1.4 Thesis organization 4

Chapter Two: C and FORTRAN Programs. 6

2.1 Introduction 6

2.2 C Programs 7

2.2.1 Data Types in C 7

2.2.1.1 Basic data types 7

2.2.1.2 Compound data types 11

2.2.2 C Program Organization 15

2.2.2.1 Function modules 15

2.2.2.2- Parameters passing 18

2.2.2.3 - Multi file program 21

2.3 FORTRAN-77 Programs . . . . , . .22

2.3.1 Data Types in FORTRAN-77 22

2.3.2 Basic vs. compound data types 25

2.3.3 Program Structure and parameters passing in FORTRAN-77. 26

2.3.3.1 Program units 26

2.3.3.2 Information Transfer. 27

2.4 Differences and Similarities between C and FORTRAN- 77 29

2.4.1 Similarities 29

2.4.2 Differences 30

2.5 Summary 31

Chapter Three: Data Variables Emigration Method 33

3.1 Method assumptions 33

3.2 The pre-processor structure and functionality 34

3.3 Mapping between C and FORTRAN variables 37

3.4 Modules calling process 40

3.6 The operational flowchart 42

3.7 Illustration Examples 55

3.8 Summary 65

.**Chapter Four: Evaluation and Results** 67

4.1 Evaluation Environment 67

4.2 Examples and results 69

4.3 Method performance 78

4.4 Applications of the method 78

Chapter five: Conclusion and Future Work 81

5.1 Conclusion 81

5.2 Future Work 82

References 83

Appendix A: The Preprocessor Program List. 85

List of Figures

Figure 2.1: Memory allocation for a record 12

Figure 2.2: Memory allocation for a union 13

Figure 2.3: Pointers and data values in memory 15

Figure 2.4: Modular Program structure in C. 16

Figure 2.5: Function module organization 16

Figure 2.6: Modules data exchange by passing values. 19

Figure 2.7: Modules data exchange by global variables 20

Figure 2.8: Modules data exchange by pointers. 21

Figure 2.9: Multi file program linking 22

Figure 2.10: Function structure in Fortran. 26

Figure 2.11: Subroutine structure in Fortran 27

Figure 2.12: Using common blocks 29

Figure 3.1: Parameters passing between C and Fortran modules 38

Figure 3.2: Sequence of operations 43

Figure 3.3: Operational flowchart: Phase I. 45

Figure 3.4: Original File Splitting 46

Figure 3.5: Modules encapsulation process. 49

Figure 3.6: Modules call process. 51

Figure 3.7: Encapsulating F-Modules and renaming of emigrated variables.52

Figure 3.8: Compile each module separately and linkage. 54

Figure 4.1: Evaluation Environment 67

Figure 4.1: Spliting Phase Information. 70

Figure 4.2: Phase II outputs 71

Figure 4.3: Compilation and Linking. 72

Figure 4.5: Extern struct and Common area statements. 74

Figure 4.6: Run output of Example2. 75

Figure 4.7: Preprocessor outputs for Phase I. 77

Figure 4.8: Example3 results. 77

List of Tables

Table 2.1 Basic data types in C with their modifiers 8

Table 2.2: Memory classes in C 9

Table 2.3: Declaration Keywords in Fortran-77 24

Table 3.1: The Emigration Symbol Table (EST) 36

Table 3.2: Data types in C and Fortran 37

Table 3.3: Naming conventions of C and Fortran data types. 37

Table 3.4: The Modules table 44

Table 3.5: Example1: Modules table 1. 56

Table 3.6: Example1: Modules table 2. 57

Table 3.7: Example1: Modules table 3. 57

Table 3.8: Example1: The EST table 1. 59

Table 3.9: Example1: The EST table 2. 60

Table 3.10: Example 1: Modules table 4. 61

Table 3.11: Example2: Modules table 1. 63

List of Programs

List 3.1: Common area implementation 39

List 3.2: Variables alignment 40

List 3.3: C function calls Fortran subroutine 41

List 3.4: Fortran program calls a C function and a Fortran Subroutine . . .41

List 3.5: Mixed program example c 47

List 3.6: Example1, passing parameters as arguments. 55

List 3.7: Example1.c 56

List 3.8: addInteger.for 57

List 3.9: output.c 58

List 3.10: Example1.c 58

List 3.11: Example1.c 59

List 3.12: addInteger.for 60

List 3.13: Finale Example1.c 61

List 3.14: Final addInteger.for 61

List 3.15: Example2.c 62

List 3.16: AddInteger.for in Example 2 62

List 3.17: output.c in Example2 63

List 3.18: Example3.c 64

List 3.19: AddInteger in example3 64

List 4.1: Example1 Mixed-Language program 70

List 4.2: Example2 Mixed-Language program 73

List 4.3: Encapsulated C and Fortran Modules. 74

List 4.4: Example3 Module. 76

List of Terms

ADT	Abstract Data Type
AEEE	Augment Entry External Table
CM	C Module
CS	C Section
EET	Entry External Table
EST	Emigration Symbol Table
F	FORTRAN 77
FM	FORTRAN 77 Module
FS	FORTRAN 77 Section
M	Module
OC	Origin C (main function)
S	Section
UDT	User Defined Type

Acknowledgment

I would like to express my deep thanks to my research supervisor *Dr. Musbah Elahresh*, who was behind the idea of this research and gave generously of his time and experience to complete this work.

My full and great thanks to the staff members in the Computer Science Department especially *Dr. Nasser El-Den Alzoghbi* , the head of the department and *Dr. Naji Bazina* , the M.Sc. Program coordinator for their support.

Finally I thank the people at the Higher Institute of Computer Technology for their permission to use the Library and Computer Facilities during my work.

Chapter One: Introduction

Computer programming languages have been evolving since the computer put in use and its applications were very rapidly and wide spread. Programming went in different directions to satisfy the programmer's requirements, the target domain, data modeling and procedures implementations.

1.1 Preview

In the selection of a programming language that suits an application, programmers wish if they could use the benefits of different programming languages. Programmers endlessly debate the relative merits of their favorite programming languages, sometimes with almost religious zeal. On a more academic level, computer scientists search for ways to design programming languages that combine expressive power with simplicity and efficiency. One of these ways is the development of methodologies for mixing programming languages.

There are many ways for mixing programming languages, either by calling and using procedures from other languages, or by converting programs from one language to another, or by data variable emigration method that is introduced in this research. Programming languages tend to focus on data types related to its domain application. However, programming languages share in supporting some common used data types like integers, real numbers, characters and others. But each language may have its own support of certain data types specially the compound ones like records, ranges, sets and others.

The selection of a programming language that is suitable for an application depends on many factors. One of these factors is the data types supported by the language and how well it models the application real-world data. This selection will not of course be an absolute and complete. Always programmers think if a data type from another language could be exit in the on-hand programming language. They may go to a solution to construct that data type as a user define or an abstract type from the existing data types supported by the language they use.

Another problem is that each programmer prefers to program with a certain language and wants to get benefits and power from other programming languages. The question is why not to program an application with different programming languages at the same time to get the benefits of those languages in data modelling and manipulation and to use previously developed routines written in a different language at the source level or as object library modules?

This research presents a new technique to mix C and FORTRAN 77, this technique focuses on the data types supported by the two languages.

1.2 Previous work in mixing C and FORTRAN 77

Many methods have been developed to mix C and FORTRAN 77, since many programmers who had worked with FORTRAN 77 have many useful code and they gained a good experience with programming using FORTRAN 77 style for scientific applications. C programming language is more popular in many applications including the system design and the scientific ones. Those programmers face now a problem how to use their old code written in FORTRAN 77 as they want to get the use of the C power. There exist some solutions to follow:

1. Rewriting the same code using C language: by keeping the same application

analyze and design but change the coding phase. This solution is a tedious, time consuming, affordable and costly task. So this solution reduces programmer productivity.

2. Converting the FORTRAN 77 code into C code using available conversion programs: One of these ways is the f2c Converter developed by *AT&T Bell Laboratories* is a program that translates FORTRAN 77 into C or C++. *f2c* lets one portably mix C and FORTRAN 77 and makes a large body of well-tested FORTRAN 77 source code available to C environments. This solution solves the problem partially since the conversion is not done 100% absolutely and its weakness is the handling of I/O statements: they are translated to calls to a run-time library which is then required each time the program is linked. The produced C code version has to be farther modified and adapted by the programmer. So still the programmer intervention takes place which reduces its productivity [5].

3. The third available solution for the programmer is to throw away the FORTRAN 77 code and developing the same application from scratch by redeveloping all the application development phases again using C language.

4. Calling FORTRAN 77 procedures from C program using Burkhard Burrow's header file cfortran.h (version 2.8) to provide an interface between C and FORTRAN 77 the problem is the programmer must use the correct libraries at link time to enable the FORTRAN 77 routines to be used properly.

1.3 Objectives of the research

The main work of this research is to introduce and present a methodology for mixing two programming languages (C and FORTRAN 77) based on data variable emigration between these two languages.

This research, as a dissertation of a M.Sc. degree, aims to satisfy the following

objectives:

1. Investigating various data types, data modelling techniques supported by the two selected programming languages(C and FORTRAN 77) and their program organization.
2. Providing the programmer with mixed language programming technique to enable him to program with two different languages at the same time and using previously developed FORTRAN 77 programs with new developed C programs.
3. Freeing the programmer from selection of a programming language between C and FORTRAN 77, so he can take the benefits from the two languages.
4. Increasing programmer productivity by eliminating the time required to create new abstract and user defined data types and code reusing in two different programming languages.
5. Introducing a new technique for data modelling and variable emigration between program sections written in different programming languages.
6. Using old developed FORTRAN 77 programs with C programs at the source level and as library object modules.

1.3 Thesis organization

The document is organized in five chapters as follow:

Introducing the research domain and the work to be done are given in this introductory chapter one. The objectives of this work, previous work in mixing C and FORTRAN 77, Problem description and the proposed solution are also briefly highlighted here. Chapter two analyzes C language and FORTRAN 77 language data types and program organization. Matching and mapping of those data types between the two selected languages and program organization in both languages are the expected outputs of this chapter. In chapter three the proposed data variable

emigration methodology is introduced and explained highlighting the requirements needed from each language and the development environment. The fourth chapter presents the results and evaluation process of the method. Evaluation of practical work and results with demo prototype examples are presented in this chapter with analysis and discussions of variables emigration between program sections written in C and FORTRAN 77 languages. Finally a conclusion of what has been done in this research and what is left for future work is set in the sixth chapter. A selection of bibliographical resources supporting this work is attached at the end of this document.

==

Chapter Two: C and FORTRAN Data and Programs

It is well known that C and FORTRAN are both imperative general purpose programming languages. However programmers using these two languages look for bringing the past FORTRAN-77 programs to their present C code applications.

2.1. Introduction

Although C is a system programming language and FORTRAN-77 is a scientific (mathematics) programming language, programmers tend to use both languages for scientific applications exploiting their generality features. FORTRAN-77 is the most widely used programming language in the world for numerical applications a long time ago. Both languages enables programmers to build sub programs as functions. However FORTRAN-77 also enables the programmer to build sub programs as subroutines as well. In all cases data is exchanged between the subprograms and the calling program using the known parameters passing strategies. In C language data is passed to a function by its value not by its name i.e call by value. Whereas in FORTRAN-77 the name of the data is passed to a function (or subroutine) e.i call by name. Another way to pass parameters to a function is call by reference. In this case an address of a data item in memory is passed to a function rather than the data value itself. This call by reference is supported by C and is not supported by FORTRAN-77[3, 5].

This similarity in program organization in functions in both C and FORTRAN-77 opened the door of interfacing programming modules written in these two languages. However this task is not directly established. Some problems have to be adapted to overcome the differences between these two languages. These differences arise from two sources: First

is the data types supported by these two languages and second from the way of passing data values between those modules.

The factors that encourage calling a FORTRAN-77 module from a C module are summarized as follows:

- Both languages are of imperative programming class.
- Both languages enables building program modules as functions.
- Calling by name in FORTRAN-77 can be equivalent to call by reference in C.
- Both languages support equivalent basic data types.
- Memory allocation in global area is similarly organized for both languages.
- Both languages are compiled languages with a linkage stage.
- Both languages are strong typed languages with explicit declaration in both languages with some distinct implicit declaration in FORTRAN 77[7, 16].

2.2 C Programs

Programs written in C language are usually organized as modules each is called a function. Data is passed between these modules from their variables by values. This gave C language the feature to be a modular programming language and generally called C consists of functions and variables [15].

2.2.1 Data Types in C

C has a very few data types. As a general purpose programming language, C has the general fundamental basic and compound data types that can be used to build useful applications.

2.2.1.1 Basic data types

The basic built-in primitive data types are integers and real single precision floating point numbers, double-precision floating point numbers and character data type. The

compound data types supported by C language include arrays of different types (from the mentioned basic data types or from compound data types as well) and structs and unions as records data types (a union is a form of struct where all elements share the same address of the allocated memory space).

Variables declarations in C language are explicit and mandatory. A valid identifier (variable name) is a sequence of one or more letters, digits or underscores character. The identifier cannot match any keyword of the C language nor reserved keywords and should begin by a letter. C is a case sensitive language where upper case letters are different from lower case letters. Data binding with variables names in C is a static binding at compile time but the memory allocation for that data can be static at compile time or dynamic at run time. Data binding and memory allocation for variables is established by a declaration statement. A general form of the C declaration statement is given below [12]:

[memory class] [modifier] type variable-name [, variable name …] [=data value] ;

There are five basic data types in C. They are given in Table 2.1 with their modifiers:

Table 2.1(a) **B**asic data types in C with their modifiers:

TYPE	C KEYWORD
Character	char
Integer	int
Floating point	float
Double floating point	double
Valueless	void
Pointer	*

All of these except for void, may be modified by the following keywords:

Table 2.1(b) **B**asic data types in C with their modifiers:

TYPE MODIFIERS(prefixes)
signed
unsigned
short
long
far and near for pointers

The term ***[memory class]*** is optional and specifies how, when and where a memory space is allocated for the declared variable(s) [15]. The following memory classes are supported by C as shown in Table 2.2.

Table 2.2: Memory classes in C

Specifier Key word	Memory class	description
Auto (or none)	automatic	Space is allocated in stack area
register	CPU registers	Data value is in CPU register
extern	External	Data in stack fixed area
static	Stack fixed area	A variable retains its value if local. A variable scope within its file.

For a full understanding of memory class requirements, here is a good grasp of three distinct but related concepts of its need:

- duration of the valid value of a data item.
- scope of the declared variable.
- Linkage and when binding and allocating take place.

Storage class specifiers help to specify the type of storage used for data objects. Only one storage class specifier is permitted in a declaration. But if the storage class specifier in the declaration statement is omitted, a default is taken as automatic.

There are only two types of duration of objects: static duration and automatic duration. Static duration means that the object has its storage allocated permanently as long as the program is under execution. An automatic memory class means that the storage is allocated and freed as necessary an according where a variable is declared. Thus the duration of a data object depends on the storage class specifier used and the position (block, module or file scope) of the declaration concerned [13].

The different types of scopes of a variable (and also for a function) determine where in the program spin a variable (or a function(can be seen and is alife so that it can be accessed. Usually the scope of a variable starts from the point where it has been declared until its life time is terminated. The scope of a variable (or a function) could be one of the following types:

- *Function scope:* the variable is visible within the function where it is declared from the point of declaration up to end of the function module. Note that this type of the scope is not valid for functions since C language does not allow declaring a function inside a function. Such variables are called local variable for that function.
- *File scope:* a variable is declared as a global variable for program modules (functions) and is visible only within the file where it is declared. Such global variables are not visible in the next files if exist. Usually those variables are classified as static global variables.
- *Block scope:* C language allow a variable to be declared within a block of statements. A block of statements is a sequence of statements encapsulated between the symbols '{' , '}'. It could be an iteration block or a selection block.

- *Function prototype scope:* It is the same as global variables for functions declarations. A function is visible from the point where its prototype is declared until end of program. If a program consists of many files then a function scope can be limited to its file by classifying its type as static.
- *Program scope:* a variable is declared as a global, non static or as the case of extern. The variable is visible to all the files constituting the program [9, 5].

Linkage is used to determine what makes the same name declared in different scopes refer to the same data object. The rules of linkage process are a little complicated. There are three different types of linkage that are described as follows:

1. A declaration outside a function (file scope) which contains the static storage class specifier results in internal linkage for that name. (The Standard requires that function declarations which contain static must be at file scope, outside any block)
2. If a declaration contains the extern storage class specifier, or is the declaration of a function with no storage class specifier (or both), then: If there is already a visible declaration of that identifier with file scope, the resulting linkage is the same as that of the visible declaration; otherwise the result is external linkage.
3. If a file scope declaration is neither the declaration of a function nor contains an explicit storage class specifier, then the result is external linkage.
4. Any other form of a declaration results in no linkage.

2.2.1.2 Compound data types

The C language has two compound data types: records (structs and unions) and arrays. Strings in C are arrays of characters, so they are not distinct compound data type. Follows a brief description of these compound data types:

Records: a records in C is either a struc or a union. In both cases more than one data item of different types are encapsulated in one data structure under one name. In a record data items are stuffed in memory one after the other and each occupies a memory space according to its type. The address of the record in memory is the address of its first element. Thus each element has its own address. Whereas in a union all elements share the same address which is in turn the address of the union and its first element. Data in a union is valid only for the last accessed element since all elements overlap each other. The following examples shows some records declaration in C:

```
struct gun {
           char name[50];
            int magazinesize;
            float calibre;
          } rec;
```

This is a record of three elements (fields) has the name rec.The first element is an array of type string of 50 characters, the second element " magazinesize" of type integer and the third field is named" calibre " of type float. Figure 2.1 shows how this record is allocated in memory.

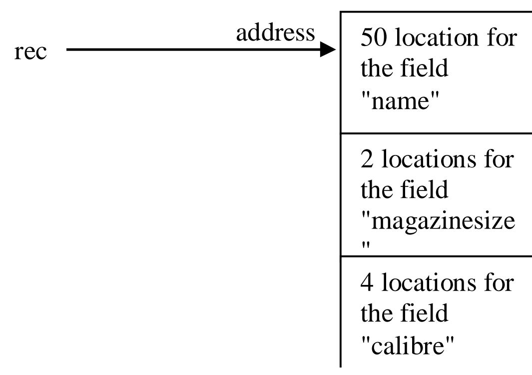


Figure 2.1: Memory allocation for a record

The next example is a union named anumber. The first element is an array of type short integer and named "shortnumber", the second element " longnumber" of type long integer and the third field is named" floatnumber " of type double. Figure 2.2 shows how this union is allocated in memory.

```
union number {
            short shortnumber;
           long longnumber;
          double floatnumber;
      } anumber;
```

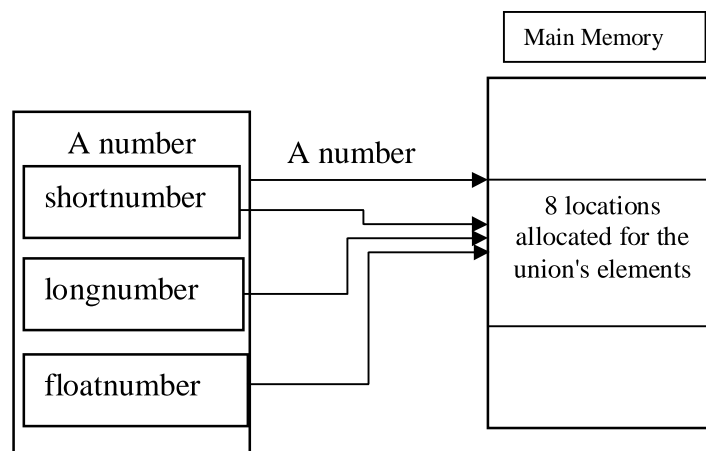


Figure 2.2: Memory allocation for a union

Arrays: In principle arrays in C are similar to those found in FORTRAN-77 that it is an aggregate of elements of the same type under one name. In C language there is a great relation between arrays and pointers. A pointer to a data type can be considered as an array of that type and an array name is considered as a pointer to its first element. The next statement shows how arrays are declared in C:

Type Name[size]

The array can be declared of any class and type and can be initialized with certain values. In C

Array subscript starts from 0 and end one less than the array size and can be of any dimension. In C Strings are defined as arrays of characters . Global arrays and static arrays in C are static arrays in which the following properties are valid:

- Range of subscripts is statically bound (at compile time).
- Storage allocations are static (initial program load time)

While local arrays that are not static are fixed stack-dynamic arrays for which the following properties are valid:

- Subscript ranges are statically bound.
- Allocation is done at declaration time (on the stack) [9, 15].

Pointers: Pointer is a fundamental and important part of C. If pointers are not used properly then basically all the power and flexibility that C allows are lost. A pointer is a special integer data type. A variable declared as a pointer holds an integer data value of the form of unsigned hexadecimal. That data value is interpreted as an address of a memory location where the pointer points to. In other words a pointer holds an address of another variable that is bound with data value in memory. It points to any variable type; basic and compound; or a function or to another pointer or to nothing (void pointer). The form to declare a pointer is given next:

```
int *p;   //pointer declaration
int y=25;   // normal integer declaration
p=&y;   // pointer assignment
```

In the first statement p is a variable of type pointer that points to an integer value in memory. In the third statement p holds the address of y, which means it now points to a memory location where the value 25 is stored as shown in Figure 2.3.

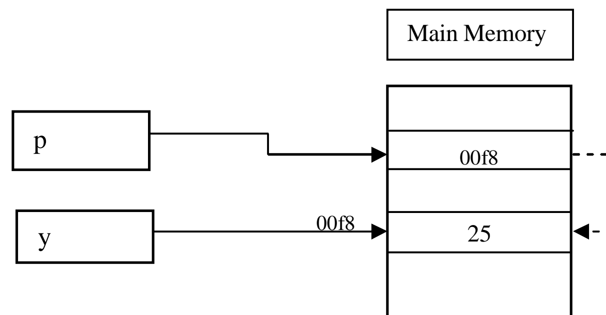


Figure 2.3: Pointers and data values in memory

A pointer is initialized as variables but usually this is done with great care and its displacement is done with the size of the data type it points to. Two operators are associated with pointers: a start '*' is the referencing operator and '&' is the dereferencing operator. The referencing operator is used to declare a variable as a pointer in the declaration statement and if it is used with a pointer elsewhere means the contents of the memory location where the pointer points to. The dereferencing operator means address of a variable and is used with data variables to get the address of a data item in memory [12, 14].

2.2.2 C Program Organization

A typical C program is organized as modules called functions. As a result of a top down design the whole program becomes a set of functions interacting by passing parameters as data values.

2.2.2.1- Function modules

A module is an encapsulated subprogram that doses a certain task for the whole program. To coordinate the operation of the whole program a main function is needed and has the dedicated name ***main*** in C. This function is the entry point of the program and from which other functions can be called. At least one function has to be called from

the main function. The main function controls the program transfer sequence [10]. This structure builds a modular program structure as given in Figure 2.4

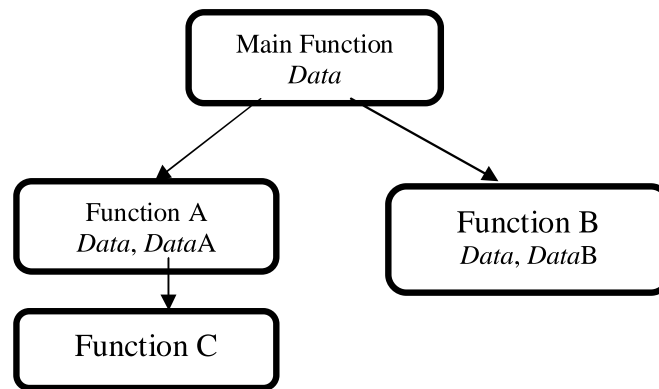


Figure 2.4: Modular Program structure in C

Each module as a C function has a standard organization as given in Figure 2.5. The function head is its interface with the other modules in the program.

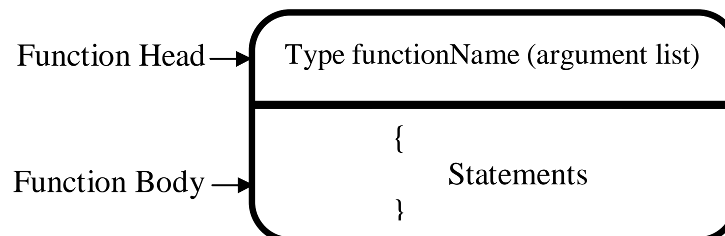


Figure 2.5: Function module organization

The returned type specifies the type of the value that is returned by the function name. This type might be of a basic data type or a compound data type or a pointer or nothing (void). However for a compound data type it is better to return a pointer to that type to avoid moving huge amount of data values between the program modules. This factor is a very important for both memory space utilization and program performance.

As the function header acts as an interface, it contains an identifier for the function as a function name. This name follows the variable naming conventions in C programming and has to be used by the other program modules (including the main function) to call this function. The argument list between the brackets is individual declarations of each parameter. The variables names are treated as local variables for the function and they receive the data values sent by the modules that call this function.

The two symbols '*{*' and '*}*' encapsulate the function body as a one programming units. The first symbol works as the entry point to the function and the second as the normal exit point. The function body contains the language statements starting with declaring local variables that are used by that function.

If the function returns a value then the statement ***"return variable name;"*** is used at the exit points. Otherwise the returned value from the function should be of type ***void*** and no return statement is used.

Functions in C language exist in three categories. The first one is Standard Library functions and the second one is System Library function whereas the third one is the user-defined functions. The first two categories are precompiled and ready to use functions and classified under header files as object modules. The programmer is allowed to call and use these functions by including their header files as required. Those functions are of general benefits and common use routines.

The third category is left for the user to develop his own functions as C text program encapsulated in a module following the described organization. However since a function name is treated as a variable in C so that it has to be declared. Declaring a user-defined function in C follows the following form:

Returned-type function name(parameters type);

This declaration statement tells the followings:

- reserving a memory space for the function returned value and the parameters. This space works as a template for the data values to be passed and returned from the function.
- binding that space with the function name.
- The statement determines the function scope and is treated a global variable. Note that in C it is not allowed to declare a function inside a function to retain the modularity principle and concept in programming. On the other hand any function should be declared to make it visible to other program modules that follow [15, 17].

2.2.2.2 Parameters passing

Parameters passing between program modules can be established in three ways. The selection of which way to use depends on how the called module interface is organized and the amount of data to be passed to that module. Follows a brief description of these three ways:

1. Passing the values through the function parameters using local variables: here the number and type of the function parameters should agree in its head, call statement and its declaration (prototype statement). The call statement sends the values of its variables list (or constant values if exist) to the variables listed in the function head. Those variables are treated as local variables in the function. Thus the variables in the callee module (the module that calls the function) do not lose their values even not changed and retain their original states. When the function completes its process and if it returns a value, the callee module should receive that value in a local variable of a type the same as the type of the function name.

This type of modules data exchange enables the program modules to exchange data values without altering the local variables of each other. Also it provides a clear independent and isolated interface between the modules. Modules using this way of data

passing are independent from the callee modules and can be reused safely without the program modification or restrictions and can be replaced by enhanced modules which increases the program maintainability. The drawback of this strategy is that if the amount of data values to be passed is large enough this will result in duplication of memory space required for the same data (in the callee module and in the called function is the same copy of data values and occupies the same space size). Another drawback is that moving huge data between program modules results in overhead time that increases program execution time. The scope of local variales is limited within the function and hence cannot be accessed elsewhere. Figure 2.6 shows how modules exchange data by passing values.

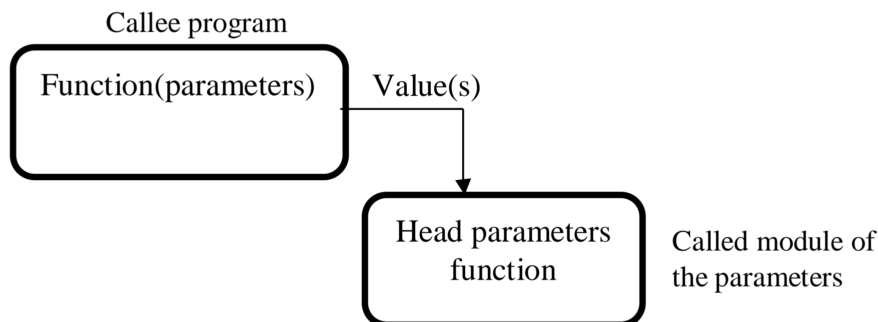


Figure 2.6: Modules data exchange by passing values

2. Passing the values through the heap area using global variables: This strategy uses global area to pass the data values through global variables. The global variables are declared outside program modules encapsulation. So that they are visible from the point they are declared up to the end of the program unless they are classified as static where their scope is limited to the program modules within the file they are declared.
Since global variables are visible by the modules following their declaration, any module can write data values into these variables and another module (even the same module)

can read those values as shown in Figure 2.7. Thus data passing and returning is easy especially for a huge amount of data. This scheme speeds up program execution but it consumes much memory from the global area. A module should be aware about the names, type, size of the global variables and is not allowed to use the same names as local variables.

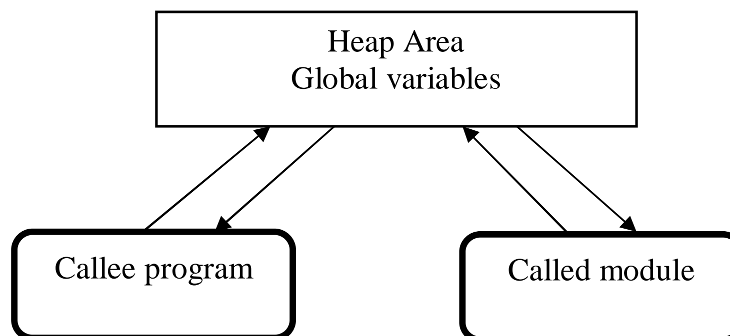


Figure 2.7: Modules data exchange by global variables

3. Passing a pointer to a data value: To utilize the memory space and increasing program speed, pointers to data values are passed between program modules without moving the data itself. A module passes a pointer of the data to be passed showing its space and location in memory as shows in Figure 2.8. The called module receives that pointer as a parameter and uses it to dereference data values in the main memory where read or write operation are performed. This scheme also isolates the modules and increase program generality and reusability. Most of the library functions use this strategy for its advantages. Care should be taken when dealing with pointers and type casting has to be used for void pointers.

It is worthily to note that programmers tend to mix one strategy with another in their applications.

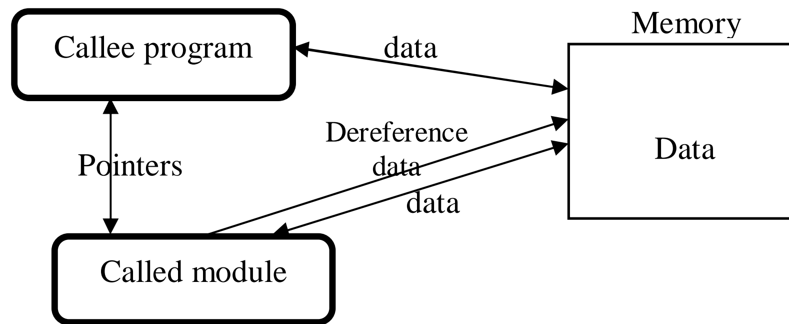


Figure 2.8: Modules data exchange by pointers

2.2.2.3 - Multi file program

A source program written in C language can exist in one text file or distributed in many files. A one file program accommodates its all variables (especially those declared in the global area) and its all modules. So for global variables they retain their scopes from the point they are declared up to the end of the program file. Local variables however have no problem since they have scope and lifetime duration only inside their modules (functions).

A multi file program have some problems with the global variable and functions modules that are declared in one file and used in another file. Those variables and functions are called external where they are externally defined of this file. Such variables and functions should be of type ***extern*** memory class. The compiler then sets aside an unresolved symbol whenever it founds a reference to such a variable or a function. Then the linker resolves these symbols if they are found in other files object codes. This is a very important concept in developing large programs in C as a project distributed among many files or using other developed source modules kept in separate files. Each file can be compiled independently producing object code file with unresolved symbols as shown in figure 3.9. To produce an executable program code all object codes are linked to gather. During the linkage process all missing symbols (those are defined as externals) are

matched and resolved. If all symbols are resolved then a combined machine code is produced otherwise a linking error is raised as shown in Figure 2.9.

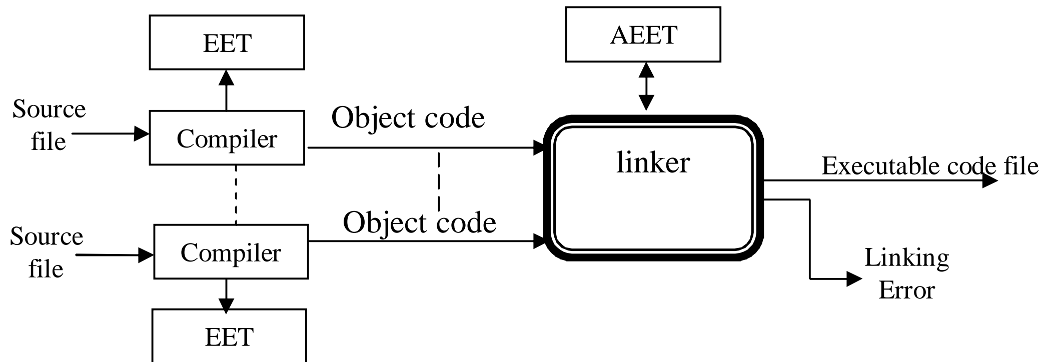


Figure 2.9: Multi file program linking

2.3 FORTRAN-77 Programs

FORTRAN is a general-purpose, procedural, imperative programming language that is especially suited to numerical computations and scientific applications. Originally developed by IBM in 1950s for scientific and engineering applications and given a name FORTRAN as FORmula TRANslation language. It was developed earlier for scientific applications that require high precision arithmetic calculations. It has many versions and FORTRAN-77 is the widely used version before the recent evolution of computer programming. Many applications are still exist that were written in FORTRAN-77 [14]. And that is the reason for selecting it as a counterpart for C language in this research.

2.3.1 Data Types in FORTRAN-77

Fortran-77, with its emphasis on numerical operations, has four data types just for numbers. These are collectively known as the arithmetic data types. Arithmetic expressions can include mixtures of data types and, in most cases, automatic type conversion is provided. The arithmetic data types are integer, real, complex, double.

Other data types rather than the arithmetic ones include logical and character data type. FORTRAN has an implicit data type mechanism, where the first character of a variable name determines the data type of the variable. If the first character of the variable name is one of the characters **I**, **J**, **K**, **L**, **M**, or **N**, then the variable has an integer type unless it is explicitly overridden with a type declaration statement. Variables beginning with other characters rather than those mentioned are assumed to be of type real.

It is usually better to explicitly declare each variable name for the expected data type that the variable to contain with a type declaration statement at the beginning of a program unit. The declaration statement is of the form:

<data type> <variable list>

where ***<data type>*** can be INTEGER, REAL, DOUBLE PRECISION, COMPLEX, CHARACTER or LOGICAL and ***<variable list>*** is the name of one or more variables separated by commas. For example the following declarations are valid:

```
INTEGER COUNT,DAY
REAL AMT,AVG,BAL
LOGICAL YES,NO
CHARACTER MASK
```

FORTRAN variable name may be from one to six characters long. A variable name may consist of letters and numbers only, no special characters, and the first character must be a letter. Fortran-77 is not case sensitive so lower case and upper case letters are the same. However programmers tended to use capital letters only for readability purposes. There are no restrictions to use the language keywords as variables names. Table 2.3 shows the declaration keywords used by FORTRAN-77:

Table 2.3: Declaration Keywords in Fortran-77

TYPE	**FORTRAN KEYWORD**
characters	CHARACTER
single precision complex number	COMPLEX
double precision floating point number	DOUBLE PRECISION
integer	INTEGER
boolean (true or false)	LOGICAL
single precision floating point number	REAL

In FORTRAN-77 data binding is static. That means all variables are bound to data values at compile time and the data type remains unchanged along the program span. So it is a strong typed language as well. FORTRAN-77 only allows static storage allocation, so that variable allocation is only static at compile time.

FORTRAN-77 distinguishes between global and local variables. Local variables are declared with a program unit (function or subroutine), while global variables are declared outside program unit. The scope of a local variable is limited to the subroutine or function in which it appears and declared; so that it is not visible elsewhere in the program. Semantically, the lifetime of a local FORTRAN variable encompasses a single execution of the variable's subroutine [10].

The CHARACTER type declares a variable that can hold an ASCII character. Only one character is allowed between single quotes. While the LOGICAL type declarers a variable to be of a Boolean value True or False. Other types acts as in C language.

2.3.2 Basic vs. compound data types

The basic data types in FORTRAN include integer, real, double precision real numbers, character, and logical data. While The compound data types in FORTRAN are complex, arrays, strings and unions (called equivalence). Complex numbers, stored as an ordered pair of real numbers. The first number represents the real part while the second number represents the imaginary part of the complex number.

The DIMENSION statement is used to tell the compiler how many elements will be contained in an array and must appear before any executable statement in the program as it is the case with all declaration statements. The format to declare an array is:

***DIMENSION** <v>(<s>) [, <v>(<s>), ... <v>(<s>)*

Where ***<v>*** is the name of the array variable and **<s>** is the maximum number of elements that will be stored in the array. FORTRAN allows for the use of arrays with up to three dimensions, so the entry for **<s>** may specify from one to three values, separated by commas.

If an array variable is explicitly typed with a REAL, INTEGER, DOUBLE PRECISION, COMPLEX, or LOGICAL statement, it is possible to specify the characteristics of the array on the type statement. It is then not necessary to include a DIMENSION statement defining the array. For example, to declare an array of type DOUBLE PRECISION to hold 50 values it could be as follows:

DOUBLE PRECISION AVG(50)

The lower bound of each dimension is one unless it is declared otherwise. There is no limit on the upper bound provided it is not less than the lower bound. Arrays of

characters form strings, for example an array of 30 characters called LINE would be declared as follows:

CHARACTER LINE(30)

Unions or **EQUIVALENCE** in FORTRAN are like records that their elements share the same address and declared by the following statement :

EQUIVALENCE (A, B, C, D), (X(1), Y(1))

This statement allows all listed variables to share the same memory address. Those listed variables could be of different types and are not related to each other. So each variable had to be declared indecently. This means that there is no encapsulation of the listed variable under one name. Each variable is accessed directly by its own name. However the validity of the data in this shared memory space is for the last written element [10].

2.3.3 Program Structure and parameters passing in FORTRAN-77

FORTRAN-77 is not a modular programming language. However it enables the programmer to build subprograms as functions or subroutines each called a program unit.

2.3.3.1 Program units

Functions and subroutines have to be in the same file where the whole program is located. A function has the structure shown in Figure 2.10.

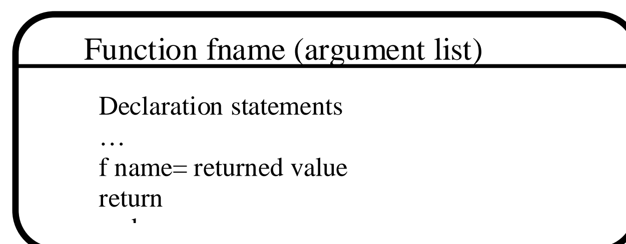


Figure 2.10: Function structure in Fortran

The function subprogram begins with the keyword ***function*** and ends with the keyword ***end***. Only the names of the received parameters are used within the function head. Inside

the function those variables are declared as necessary. The keyword **return** is used whenever a value has to be returned. Thus somewhere in the function body the name of the function is assigned the value to be returned to the callee program unit. So the function in Fortran may receive many values but it returns only one by its name.

A subroutine in FORTRAN-77 does not return a value through its name. However it receives a list of variables names to work on. Figure 2.11 shows the structure of the subroutines in FORTRAN-77:

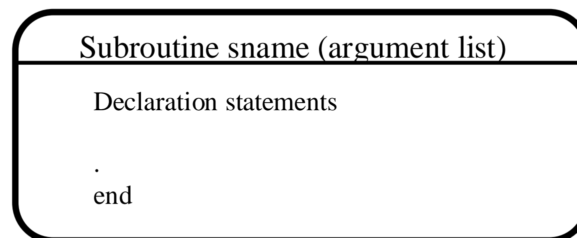


Figure 2.11: Subroutine structure in Fortran

It is best to think of the subroutine as the more general form of procedure; the function should be regarded as a special case for use when it's only need to return a single value to the calling unit.

2.3.3.2 Information Transfer

Information can be transferred to and from an external procedure (function or subroutine) by one of two methods:

1- **As an argument list**: This is the preferred method of interfacing as it is the most flexible and modular. Information is sent / received by its variables names in the calling statement, program unit argument list and the return statement.

2- **Common Blocks:** A common block is a list of variables of any type stored in a named area which may be accessed directly in more than one program unit (the original program ,functions, subroutines). Common blocks are mainly used to

transfer information from one program unit to another globally. They can be used as an alternative approach to argument list transfers or in addition to them. This is equivalent to C global variables. A COMMON block is declared together with any other declarative statements just with a sub-program (function or a subroutine):

```
PROGRAM EXAMPLE
REAL X
DIMENSION X(100)
COMMON X
.
END
SUBROUTINE PASS1
REAL Y
DIMENSION Y(100)
COMMON Y
.
.
END
```

This has the effect of allowing the data stored in array X in the main program segment to be used as array Y in the subroutine. This particular form of the COMMON is known as ‘blank’ COMMON, since the common block has not been given a name. A name might be assigned to a common block, so that many common blocks can distinguish easily:

```
COMMON /A/ X(100)
COMMON /XRAY/ Y(20),B(30),Z
COMMON /HEAT/ A(25)
```

Using common blocks, the communication between program units is effected through the sequence of the variables: a section of memory is set aside so that more than one program unit accesses it. Figure 2.12 shows how this is done by the following example:–

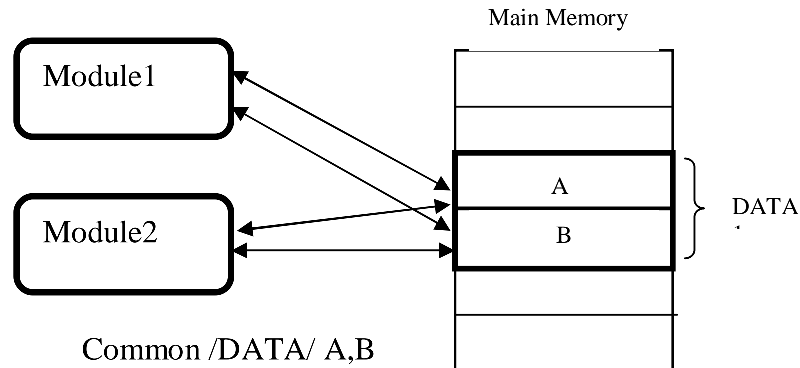


Figure 2.12: Using common blocks

The fact that these had different descriptions in the calling routine is of no significance to Fortran. Programmers must be careful with common blocks, especially when they note that anything in a named common block may become undefined when the block exited or leave a sub-program using END or RETURN, unless the SAVE declaration is used. Essentially, this would only apply to the case where a return to a routine which does not have the named common block in it [17, 10].

2.4 Differences and Similarities between C and FORTRAN- 77

There are many similarities and differences between C and FORTRAN-77 in the supported data types and program organization. These are summarized in the next subsections.

2.4.1 Similarities

Both languages support functions as encapsulated one program unit that returns a single value through its name. While some Fortran implementations permit a function to be referenced by a CALL statement, it is decidedly nonportable to do so, and is a violation

of international standards for Fortran. So that a function is referenced similarly by its name as in C.

Good programming practice declares non-value-returning functions to be of type void in C, and value-returning ones to be of any non- void scalar type. Both languages support nearly the same basic data types and the commonly used compound data types that include arrays and structs (only unions and Complex in Fortran-77).

Both languages are strongly typed languages although Fortran-77 has implicit declaration depends on the first letter of a variables. And in both languages binding and allocation is done statically at compile-time (C has also dynamic allocation at run time).

Both languages have the strategy to pass parameters to functions as arguments or global area variables. C uses call by value and by reference which is equivalent to call by name in Fortran-77 to pass parameters to functions as arguments. The global area in C is similar to the common area in Fortran-77. Fortran-77 provides *COMMON blocks* for shared global memory via COMMON statement.

Parameters passing between program functions is nearly similar in both languges, passing by name in Fortran is equivalent to passing by pointer in C, global variables in C is equivalent to common blocks in Fortran. The calling process is similar in both languages.

2.4.2 Differences

Fortran-77 stores arrays in row order, with the first subscript increasing most rapidly. C stores arrays in column order with the last subscript increasing most rapidly.

Fortran-77 array indexing starts at one, while C indexing starts at zero. An N-by-N matrix is declared in Fortran as type name A(N,N), and indexed from 1 to N, while a corresponding C array is declared as type name A[N][N] and indexing starts from 0 to N-1 in both dimensions.

Fortran-77 has SUBROUTINE as a program unit, which does not return a value, and must be invoked by a CALL statement, and FUNCTION, which returns a scalar value, and must be invoked by function-like notation in an expression. Whereas C does not have subroutines but supports functions as Fortran-77.

Fortran-77 functions can return only scalar values, not arrays or any compound data tyoes whereas C does return any data type including arrays, structs, pointers and so on [11]. Calling by value is not supported by Fortran and calling by name is not supported by C.

2.5 Summary:

the outcome of this chapter compares C and FORTRAN-77 data types, binding, allocation and program organization. The following points concluding this chapter:

- In both languages, any variable of any type should be declared before use in order to allocate the required memory space and binding it with data value.
- In C language any function should be declared to make it visible to other program modules.
- Passing parameters through the call statement as arguments is done in C by value and by reference and in Fortran-77 is done by name. However calling by name is equivalent to calling by reference.
- Passing parameters through global variables in C is equivalent to passing parameters through common blocks between functions in Fortran-77.
- In C a function returns a value of any type through the function name whereas in Fortran-77 only scalar data value is returned by the function name. So in a fortran-77 function should exist an assignment statement to assign that value to the function name.

- To mix these two languages, the similarities proper have to be utilized and the differences must be defined and studied for solutions.
- Management of dynamically-allocated heap memory is similar in both languages and has to be utilized for proposed solution.
- Even though each language has its syntax and naming convention for variables it is clear that the mapping between the two languages is not a hard task.
-

==

Chapter Three: Data Variables Emigration Method

The proposed method for mixed-languages programming in this research enables the programmer to develop programs using C and FORTRAN 77 77languages together. A program has to be organized in sections and modules each written in one of these two languages.

A solution of reusing previous FORTRAN 77 code with C code is to develop a method that enables the programmer to write and mix his programs in both languages (C and FORTRAN 77) at the same time. The method should be transparent from the programmer so that he feels smooth as if he were writing his code using one language. By this solution the programmer can:

- Develop programs in both C and FORTRAN 77 77languages.
- Reusing previously developed code written in C or FORTRAN 77 either exist in source format or object format as library modules.
- Getting a language benefits of data presentation and procedures performance that not exist in the other language.
- The same methodology can be used for other languages by following the same concepts of the proposed method.
- Code conversion/ data types mapping are transparent from the user so that increasing his productivity.

3.1 Method assumptions

The method is based on a proposed technique in this research that enables data variables to emigrate between different program sections and modules. Here we deal with FORTRAN 77

77code lines inserted within a C program and calling a FORTRAN 77 77 module from a C function. To simplify the development of the proposed method, the following assumptions are taken into account:

1- The original mixed-language program is a C program that uses FORTRAN 77 77code (F-code). The F-code is either embedded code lines within the C code or separate FORTRAN 77 77function.

2. The C code is organized as a main module (main function) that contains C code and FORTRAN 77 code sections. These C and FORTRAN 77 code sections needn't be in a specific order.

3. C function (user defined function code) is called from C program sections and FORTRAN 77 functions are called from both C or FORTRAN 77 77 sections or modules.

4- A FORTRAN 77 77can access and use variables from a C section and vice versa.

5- A pre-processor has to be developed to partition the original mixed-language program into separated C and FORTRAN 77 77 modules according to the above assumptions. The pre-processor also has to define the way to exchange data between the C modules and FORTRAN 77 functions.

3.2 The pre-processor structure and functionality

By processing we mean compiling and then linking program modules to generate an executable code. By pre-processing we mean preparing the source code written in a mixed-language for that compilation and linking process.

The pre-processor is the entry point for the proposed mixed-languages programming methodology. It is a software (has been developed here) that takes the original program written in C and FORTRAN 77 77languages and does the following tasks:

<u>1. Identifies the program sections</u>: i.e which section contains a C code and which section contains a FORTRAN 77 code. The pre-processor utilizes the layout organization of the

mixed-language program proposed by this method. The source program contains preprocessing directives that are described as follows:

%OC name: indicates the starting section of a mixed-language program. As it was mentioned, the first section is a C main function. This symbol also identifies the proposed methodology programming style. The *name* is a tag name for the mixed-language program entry point. This entry point is the C main function.

%CS name: a start of C-code lines section, this section contains code lines written in C language that follow FORTRAN 77 code lines. The *name* identifies this code section name. The pre-processor adds the extension ***.c*** for this name during partition process.

%FS name: a starting point of a FORTRAN 77-code section. The name will be extended by the extension ***.for*** by the pre-processor and assigned to the FORTRAN 77 77module generated during preprocess. The end of any section is detected by the starting tag for the next section.

%CM name: here starts a C function module. It is a user defined function and has to follow the rules used by C programming organization. The ***name*** will be given to the module that is produced during the program preprocessing after adding the extension ***.c*** to it.

%FM name: here starts a FORTRAN 77 77function module. It is a user defined function and has to follow the rules used by FORTRAN 77 77programming organization. The name will be given to the module that is produced during the program preprocessing after adding the extension ***.for*** to it.

$ var-list $: list with variables names that emigrate from the previous section to the next code section. These variables are declared and defined in C-code section and used in F-code section including any returned value from a FORTRAN 77 module.

%END this directive indicates the end of the source program written in the mixed-language programming method. By detecting this tag the pre-processor terminates its first

phase (program splitting into C and FORTRAN 77 modules) and enters the next phase for emigrated variables locating and mapping as it will be described in next.

In all cases the given name follows the naming conventions used by the system under use.

2. Preparing a table with the emigrated variables: The preprocessor uses the emigration statement to define the emigrated variables types and names and the way how they emigrated. More specific format of the emigration statement is given below:

$ tag , type varname, type varname, . . . ,<type varname $

Tag:

> The variables are passed as arguments in the call statement.

^ The variables are passed as global parameters.

! No variable is passed.

< The variable is a returned value.

The preprocessor builds an Emigration Symbol Table (EST). This table list all emigrated variables with their names and the module name for variables name mapping between C and FORTRAN 77. This table has the form shown next:

Table 3.1: The Emigration Symbol Table (EST)

Variable-name /type	Source module	Modified name/type	Destination module

3. Building the required statements: The preproessor builds the required statements to encapsulate each module, declaring of each module, calling each module and parameters passing. It also builds the required statements to introduce the external modules.

4. Preparing all generated modules for compilation and linking: The prepocessor passes each module to its compiler for compilation.

3.3 Mapping between C and FORTRAN 77 variables

Both languages support the most basic data types and other compound data types. Data types differ in size, type declaration convention and variables naming conventions as given in Table 3.2.

The naming conventions of data variables in C differ slightly from those in FORTRAN 77. Any variable that is declared in a C-code section and emigrated to a F-code section has to be mapped into the corresponding type and name convention used by the FORTRAN 77 language as given in the Table 3.3.

Table 3.2: Data types in C and Fortran

C	FORTRAN 77
short int	integer*2
long int or int	integer
int iabc[3][2];	integer iabc(2,3)
int	logical
or unsigned char	logical*1
float	real
double	real*8
struct{float r, i;}	complex
struct{double dr, di;}	double complex
char abc[6];	character*6 abc
#define *PARAMETER value*	parameter

Table 3.3: Naming conventions of C and Fortran data types

Type	C	FORTRAN 77
Basic *	**A**	**A**
Array	A[7]	A(7)
Record	Struct { .....} R;	none
Pointer	*p, &p	none
complex	Struct { ouble, double	dr, di
*integer, character, float, double		

The emigrated variables from a C-code section to a F-code section are treated as parameters passed between a C-Function and an F-Function. The call process is for C call by reference and for FORTRAN 77 call by name. There are three ways how parameters are passed between C function and

FORTRAN 77 function as illustrated by figure 4.1:

First: A C code calls a FORTRAN 77 module and passes the parameters as arguments in the call statement. Here variables are emigrated from a C function to a FORTRAN 77 function and the returned value is emigrated back to the C function through the FORTRAN 77 function name as shown in Figure 3.1 (a).

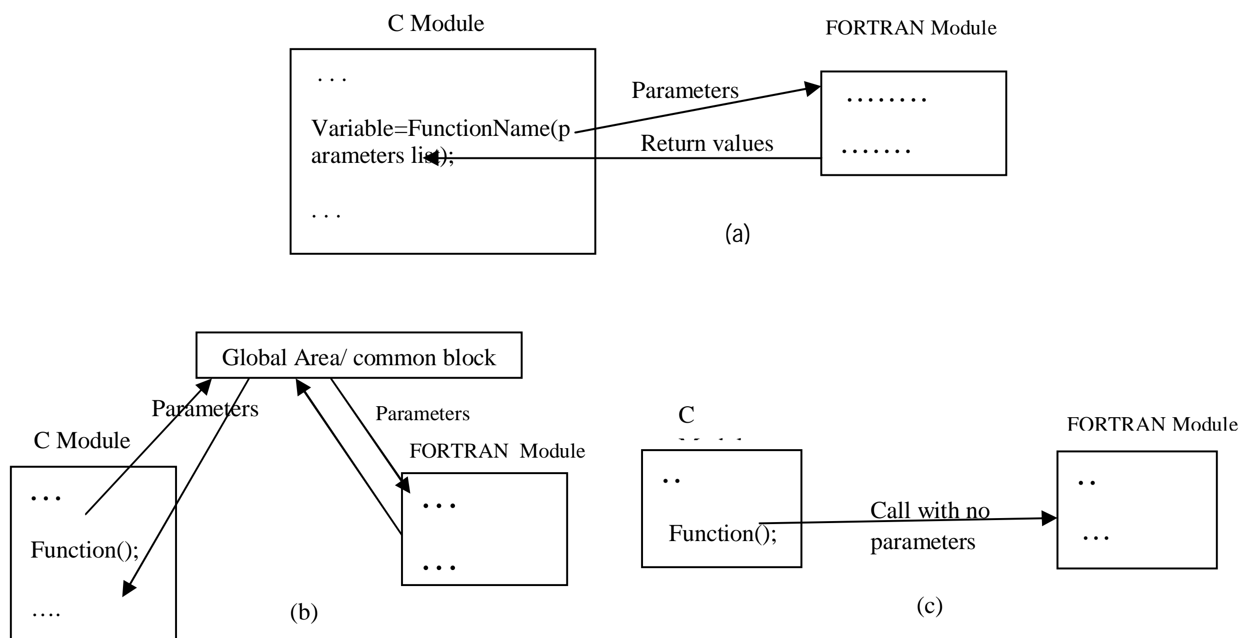


Figure 3.1: Parameters passing between C and Fortran modules
a: as arguments
b: as global variables
c: no parameters

Second: A C code calls a FORTRAN 77 module and passes the parameters as global variables. Variables are emigrated externally from the function call. They are placed by the C function in a global area where they become accessible by a FORTRAN 77 module in a common block area as shown in figure 4.1 (b).

Third: A C code calls a FORTRAN 77 module without passing any parameters. The

FORTRAN 77 module works as standalone and does its task independently form the C module.

These three ways of parameters passing can be mixed as needed. But to simplify the evaluation of the proposed method, each way is implemented and tested seperatly.

When using a common area to transfer the values of the emigrated variables some notes have to be considered:

1. FORTRAN 77 common block and global C struct of same name are equivalent.
2. Never use un-named common blocks. A common block must have a name. So that it become known for both modules.
3. Reference variables within the common area in same order, same type and with the same name for both C and FORTRAN 77 modules. So that valid data is obtained.
4. Character data is aligned on word boundaries and other types have to be considered as shown in Tables 3.1 , 3.2.

The following program segment **List 3.1** illustrates how to build the common area for data values transfer between FORTRAN 77 and C and its naming conventions.

<table>
<tr>
<td>C:<pre>
extern struct{
   double x;
   int a, b, c;
} abc_;
</pre></td>
<td>FORTRAN 77:<pre>
DOUBLE PRECISION X
INTEGER A, B, C
COMMON/ABC/ X, A, B, C
</pre></td>
</tr>
</table>

List 3.1: Common area implementation

Note that the use of extern requires that the common block be referenced first by FORTRAN 77. If it is referenced first by C then drop the extern as in this research. The extern statement states that it is trying to reference memory area which has already been set aside elsewhere by another program module reside in another file.

Byte alignment can be a source of data corruption if memory boundaries between FORTRAN

77 and C are different. Each language may also align data structure differently. So it must preserve the alignment of memory between the C "struct" and FORTRAN 77 "common block" by ordering the variables in the exact same order and exactly matching the size of each variable [19]. It is best to order the variables from the largest word size down to the smallest starting with "double" followed by "float" and "int". Bool and byte aligned data should be listed last as it is shown in the following program segment **List 3.2**:

C:	**FORTRAN 77:**
extern struct{ int a; double d; unsigned char flag; int b; double e; } abc_;	INTEGER A, B, C DOUBLE PRECISION D, E, F LOGICAL*1 FLAG COMMON/ABC/ A, D, FLAG, B, E,F

List 3.2: Variables alignment

3.4 Modules calling process

The emigrated variables from a C-code section to a F-code section are treated as parameters passed between a C-Function and a F-Function using the described strategies to transfer the values of these variables. The entry point names for some FORTRAN 77 compilers have an underscore appended to the name. This is also true for common block/structure names and has to be considered in the calling process as shown below:

C	FORTRAN 77
subra_(...)	call subrA(...)

All arguments in FORTRAN 77 are passed by reference and not by value. Thus C must pass function arguments as pointers to dereference those values as shown next:

C	**FORTRAN 77**
subra_(int *i, float *x)	Call subra(i,x)

The following program segment List 3.3 shows a C program calling a FORTRAN 77 subroutine using passing the parameters as arguments and has the following form:

testC.c	**testF.for**
#include <iostream>	subroutine **fortfunc(ii,ff)**
using namespace std;	integer ii
extern void fortfunc_(int *ii, float *ff);	real*4 ff
main()	write(6,100) ii, ff
{	100 format('ii=',i2,' ff=',f6.3)
int ii=5;	return
float ff=5.5;	end
fortfunc_(&ii, &ff);	
return 0;	
}	

List 3.3: C function calls FORTRAN 77 subroutine

While the following program segment List 3.4 shows a FORTRAN 77 program calling a C function by passing the parameters as global variables through the common area:

testF.for	**testC.c**
program test	#include <stdio.h>
integer ii, jj, kk	
common/ijk/ ii, jj, kk	**extern struct**
real*8 ff	**{**
character*32 cc	**int ii, jj, kk;**

<pre>   ii = 2
      jj = 3
      kk = 4
      ff = 9.0567
      cc = 'Example of a character string'
      write(6,10) ii, ff
10   format('ii= ',i2,' ff= ',f10.4)
      call abc(ii)
      write(6,20) ii
20   format('ii= ',i2)
      write(6,30) ii, jj, kk
      call doubleIJK(cc)
      write(6,30) ii, jj, kk
30   format('ii= ',i2,' jj= ', i2, ' kk= ', i2)
      write(6, 40) cc
40   format(a32)
      stop
      end
      subroutine abc(jj)
      jj = jj * 2
      return
      end</pre>
	<pre>**} ijk_;**
	
	int **doubleijk_**(char *cc, int ll)
	{
	  printf("From doubleIJK: %s\n",cc);
	  ijk_.ii *=2;
	  ijk_.jj *=2;
	  ijk_.kk *=2;
	  return(1);
	}</pre>

List 3.4: FORTRAN 77 program calls a C function and a FORTRAN 77 Subroutine

3.5 The operational flowchart

The sequence of the operations followed by the proposed method as a software development environment is described first by the diagram shown in Figure 3.2 and the emphasis of each phase is individually given later.

The source program is a mixed language source program, prepared by any text editor in the form organized in the way described earlier. C and FORTRAN 77 sections and modules

should begin with the required directives. All emigrated variables have to be encapsulated in the emigration statement. The mixed-language program is saved in a file under a name with the extension .mix. Preprocessing of this files goes into four phases to obtain an executable code. These phases are described next.

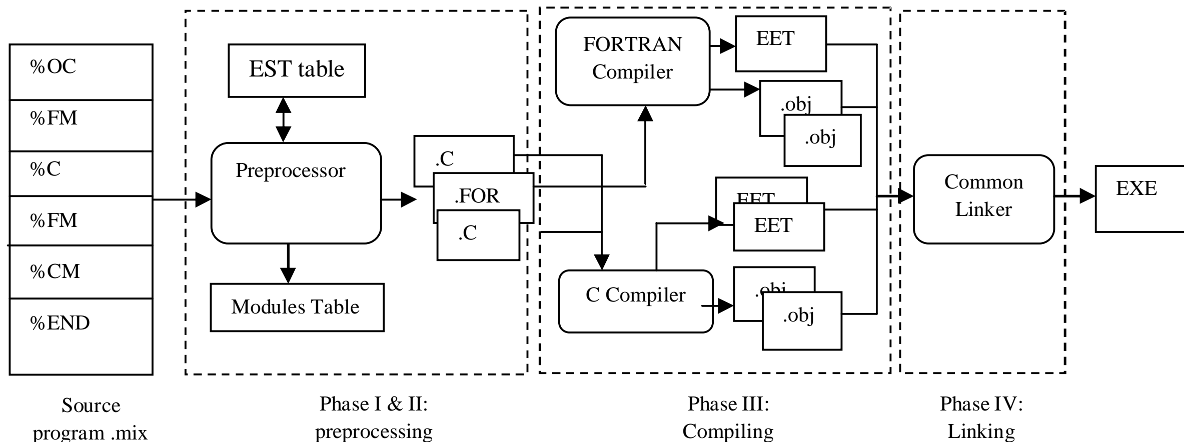


Figure 3.2: Sequence of operations

Phase I: *Splitting the mixed-language program into C and FORTRAN 77 separated modules:*
The pre-processor distinguishes the program sections and modules from the directives statements inserted at the beginning of each section and module. Code lines located between two directive statements are considered as a module of the type specified in the first directive statement. Each module is cut and saved to a separate file with the name given in the modules tag identifier.

The pre-processor also builds a table with the modules names and type (C or FORTRAN 77 as appeared in the sequence of the original program) that to be used by next phase. The following operational flowchart in Figure 3.3 describes this process.

The Modules Table is needed to save information about the splited modules. It contains three columns as shown in Table 3.2. Each entry in this table saves the name given to the separated

module (it is taken from the second token from the directive statement). The file type shows whether the module contains C code or FORTRAN 77 code. The third field indicates whether the code is F-code lines embedded within C-code or a standalone module of C or F code. The table is then used by the next phases to access these modules for a farther preprocessing.

Table 3.4: The Modules table

File name	Type: .C/.for	Section/module

FORTRAN 77 code exists in the original program in two forms: F-code lines embedded into C-code module or a separate F-module (function/subroutine) that is called from a F-Module. For the first form the F-code lines have to be organized and encapsulated as a FORTRAN 77 function. This results in adding a C call to this module in the place of the F-code lines and adding the required extern struct/common area constructs to the generated F-module at the right places according to the strategy to be used to pass the parameters section that follow. For the second form C calls F-module. The pre-processor has to add the extern struct/common area constructs in the right places in each module.

The above description of phase I results in that no need to separate the C-code lines in a separate modules since they are treated as a continuation in their C-module. So we have an original program that is written in a mixed-programming languages (C and FORTRAN 77) with the following properties:

(a) The overall view of the program is a C program with a main function as an entry point of program execution control flow and other functions (modules) as given in Figure 3.4-a.

(b) F-code lines (as a continuous block) are inserted within a C-program (in the main function)

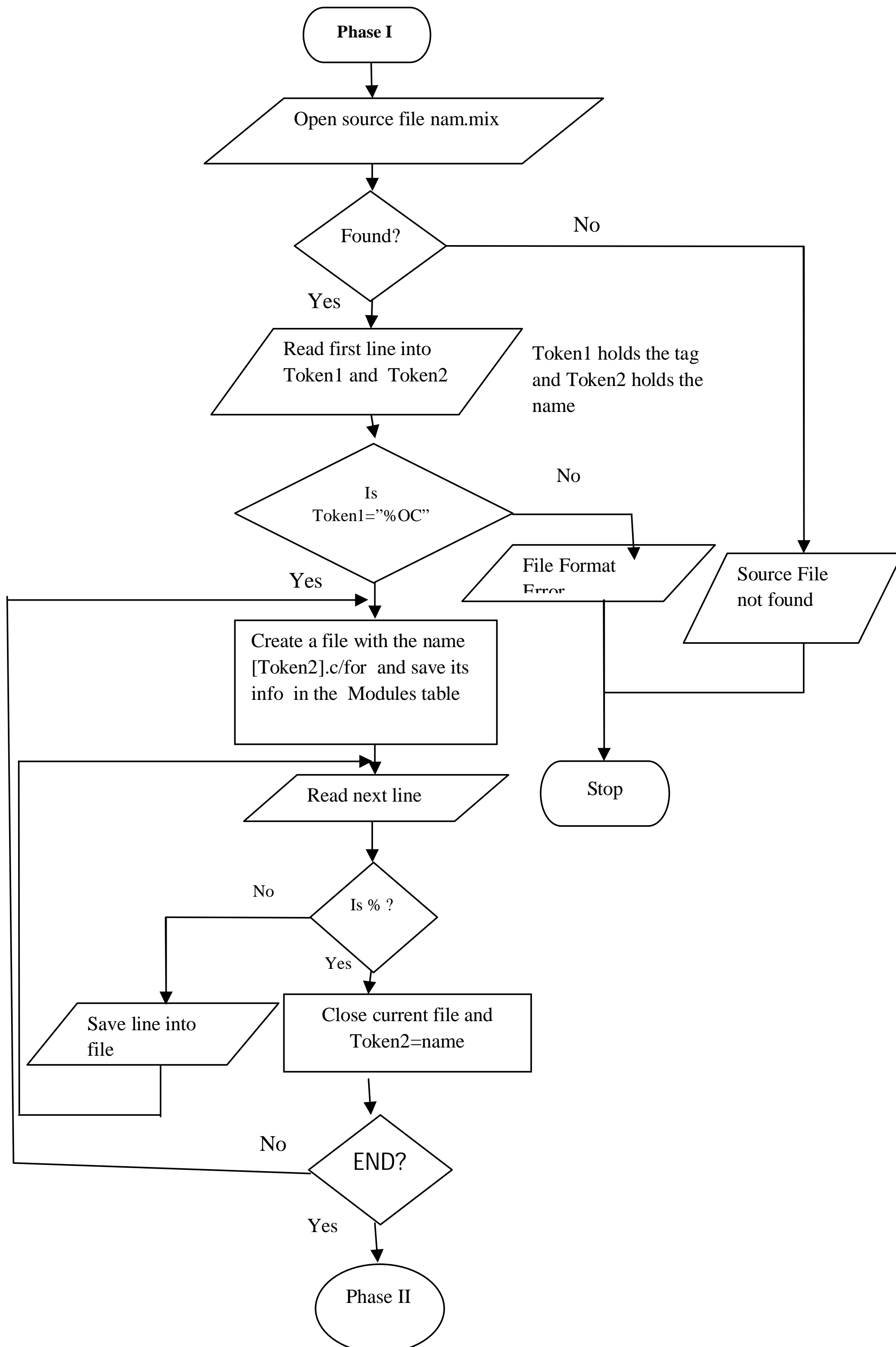


Figure 3.3: Operational flowchart: Phase I

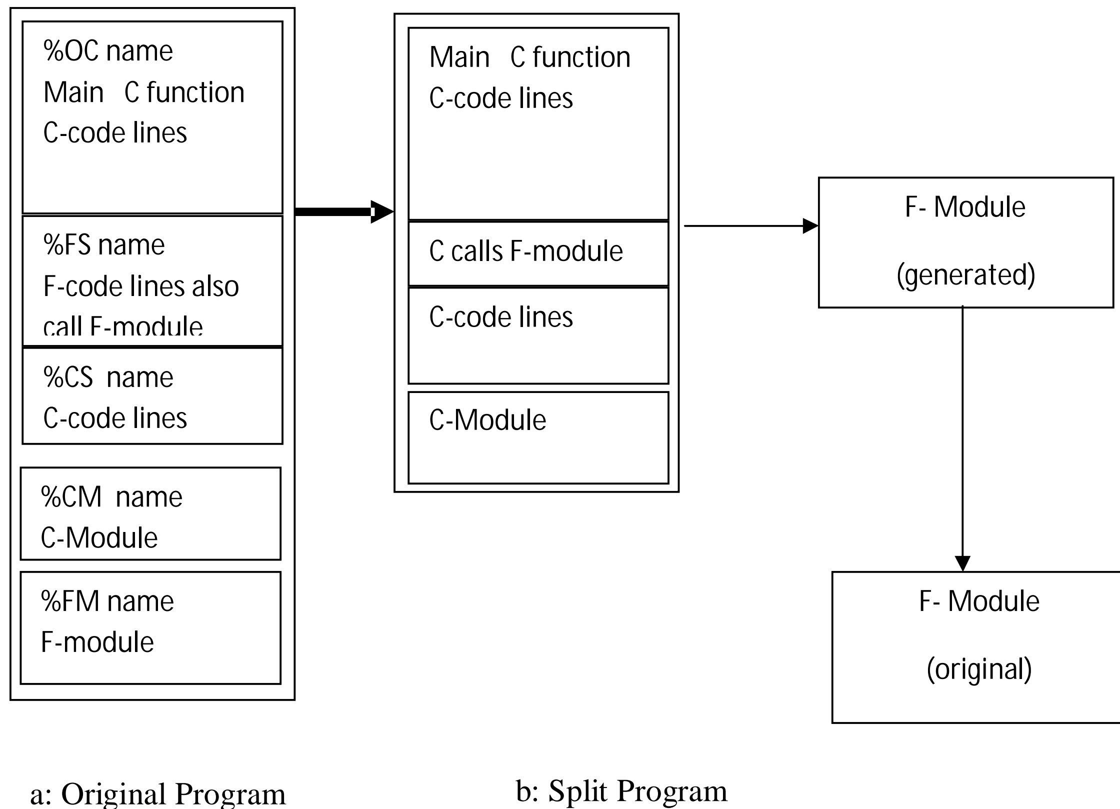


Figure 3.4: Original File Splitting

(c) F-modules are appended to the C program and called from C sections or FORTRAN 77 sections as shown in figure 3.4-b above.

Phase II: *Scanning each module file and generating the Emigration Symbol Table (EST)e:* F-code sections use C-variables with C naming conventions declared at the start of a C-module are treated as F-modules (functions or subroutines). So the emigrated variables have to be sent as parameters to that generated F-module using one of the described strategies. The F-code section may also contain F-variables declared at the start of this section. So they are considered as local variables within the generated module. C-code sections that use FORTRAN 77 variables are beyond the scope of this research for now and left as future work.

Therefore C-code sections are treated as continuation code lines of the last C-code lines. Only the returned value from an F-module is handled in the C section.

To complete the job the following three questions have to be answered:

1. How to collect the emigrated variables ?

2. How to map the variables naming conventions and declarations?

3. How to encapsulate each module and add the required extern struct/common area constructs.

Now we answer the first question with the following description:

- Case of C-variables are used by F-code section: a directive statement for the preprocesor is introduced to accommodate all C-variables that are to be used by a F-code section and declared in a C-code. The directive statement takes the following form:

$ tag,` type name, type name, ...,type name, returned type name $

where type is the type of the emigrated variable and name is the name of the variable in C naming convention. All variables are separated by commas.

An example of this might be as follows:

```
%OC EXAMPLE
   main() {
          int a;
          char b;
          float c;
          scanf("%d %c %f",&a, &b, &c);
 %FS section1
         $ >, int a,char b, float c, < void  $
         DO 10 I=1,5
         c= (c*a)/b
         10 continue
 %CS section2
         printf("c=%f\n",c);
        }
 %END
```

List 3.5: Mixed program example

Thus the pre-processor has to collect all variables listed in this directive statement and treats them as emigrated variables. These variables are then saved in the emigration Symbol Table (EST) for each generated F-module. This set of variables is then converted to its corresponding types and names used in F-module which answer the second question. The converted and the original names are then used to construct the call statement parameters list or the common area space in both C- Module and F-Module. For F-module call statement, call normally and the module specifies $ ... $ the set of the received variables.

To build the common area constructs and to convert naming conventions the method uses this EST table and the process is done automatically by the method preprocessor. The FORTRAN 77 code needs then to be scanned and each emigrated variable is replaced by its corresponding renamed variable from the EST table as described in the section of Mapping between C and FORTRAN 77 variables earlier. What we have now? On one hand the Module Table, EST table and on the other hand the program modules (C and FORTRAN 77 subprograms) and FORTRAN 77 and C sections each in a separate file under its specified name. The process to build a main program in C that calls FORTRAN 77 modules and C modules and collecting the emigrated variables, is done as follows:

1. At each place of a FORTRAN 77 section in the main module place a call statement to FORTRAN 77 function, the parameters are given in the emigration statement with the way how to pass them and the returned value.
2. Each FORTRAN 77 module then calls FORTRAN 77 module(s) normally.
3. Original C modules and F modules are kept as they are and called normally.
4. C sections are returned to their original place with no added information.

The following flowchart illustrates this process:

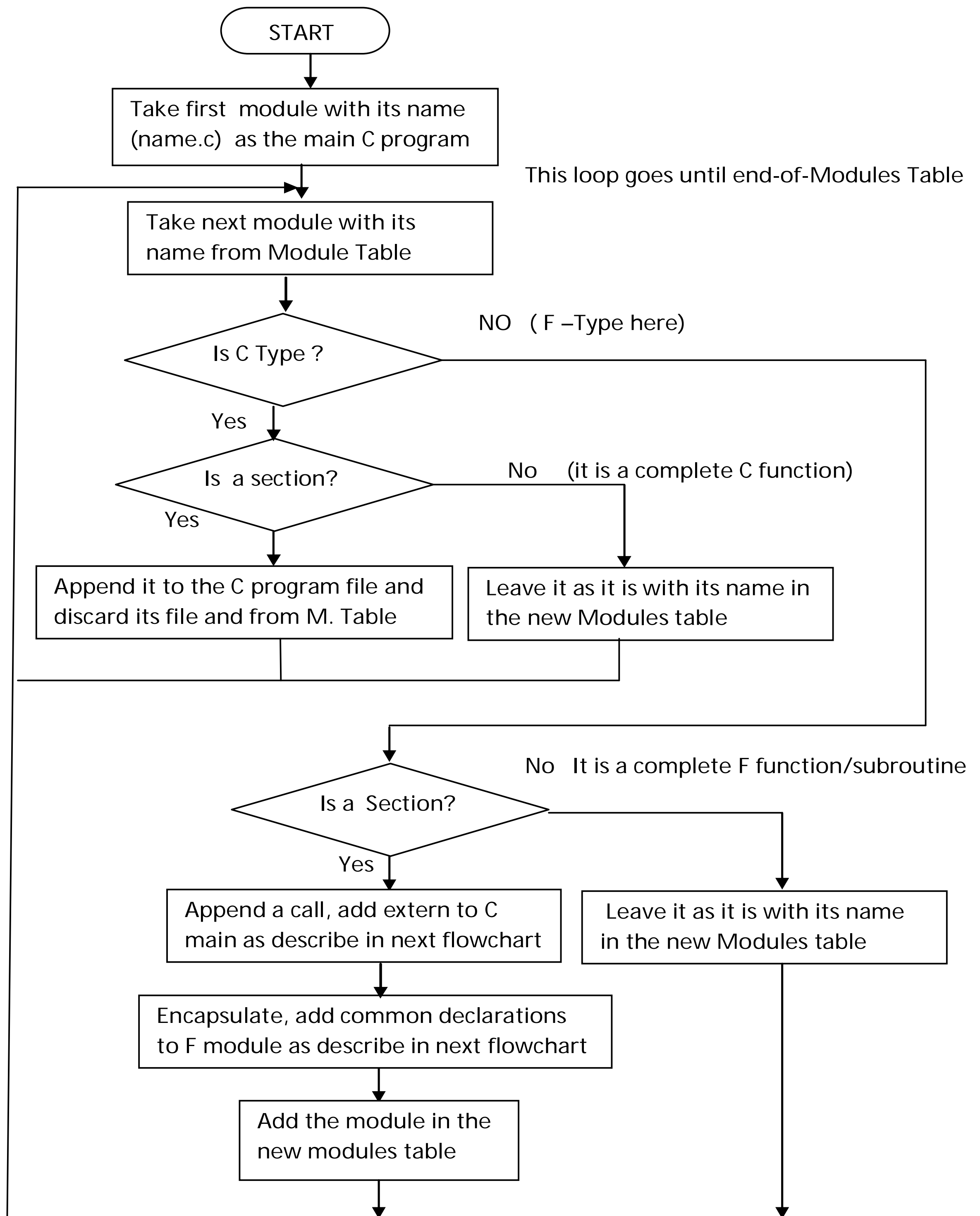


Figure 3.5: Modules encapsula on process

Mapping the emigrated variables and encapsulating each module by adding the common/extern struct constructs and the other statements.

FORTRAN 77 modules are treated as external entities by C modules. This process goes in the following steps:

a: inserting the call statement in the calling C program :

a-1 the call statement and function prototype.

a-2 the extern struct statement if parameters are to be passed globally

The next flowchart in figure 4.6 shows the sequence to be followed to insert the call statement and the extern/ struct statement:

b: Heading the F-module:

b-1 the function head and Variables list

b-2 the common statement and variables declaration

b-3 the return and end statements.

As a result from that sequence we get each module ready to be compiled.

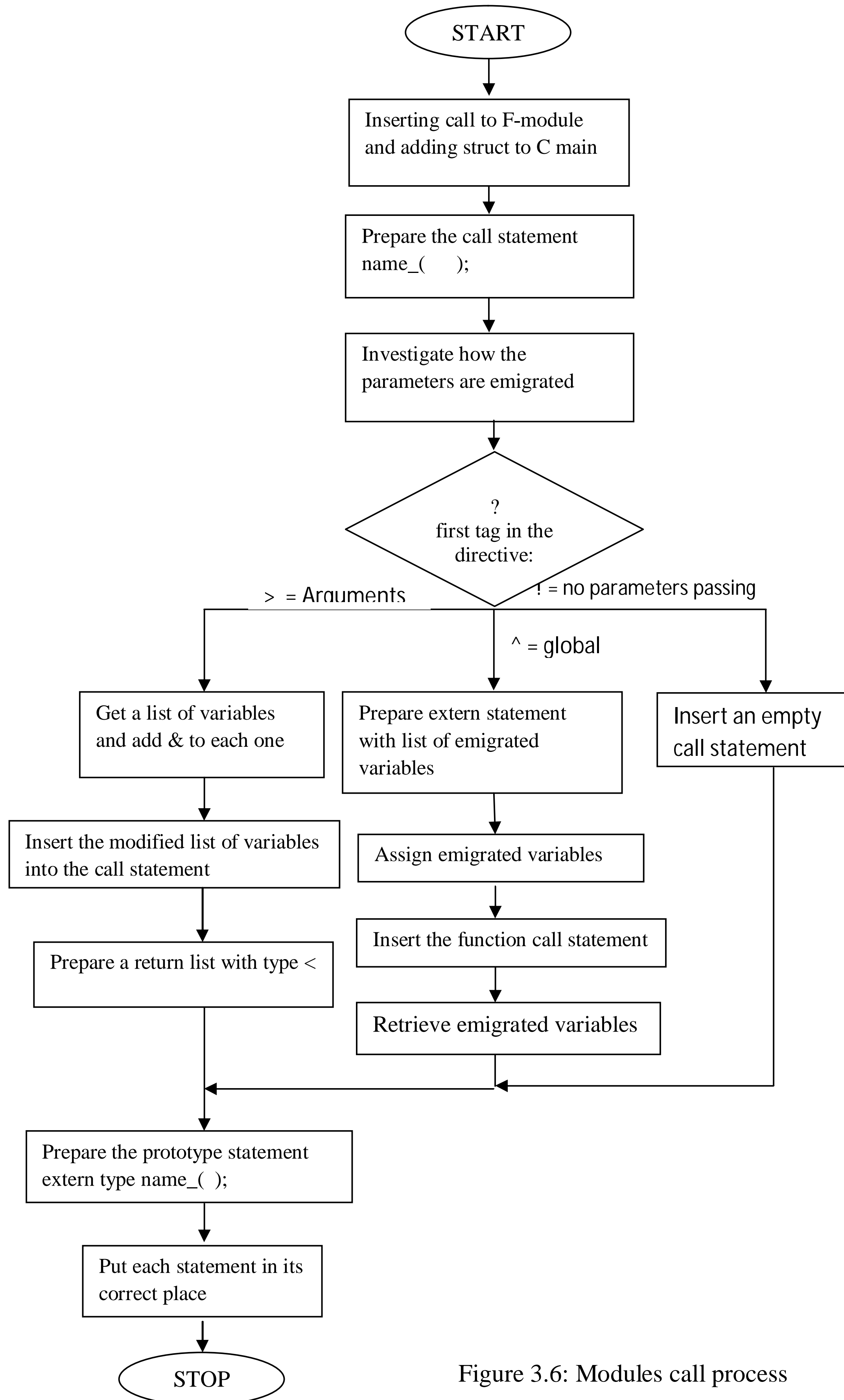


Figure 3.6: Modules call process

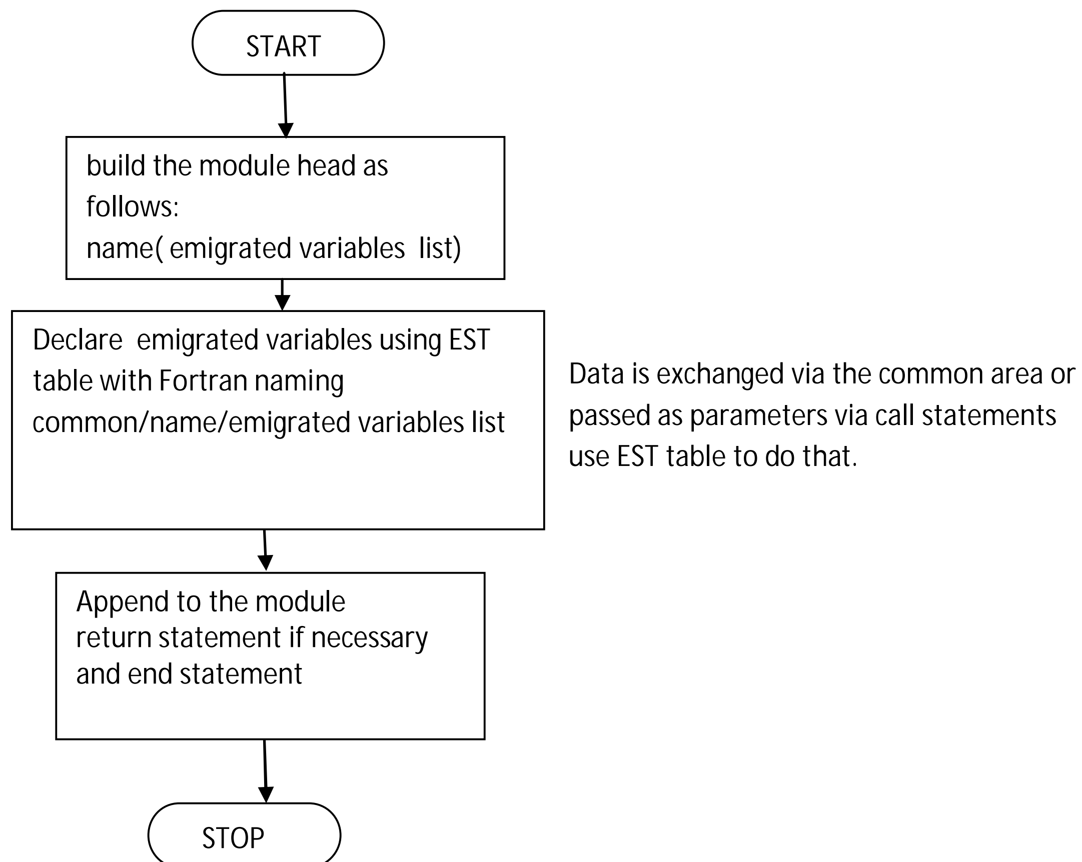


Figure 3.7: Encapsulating F-modules and renaming of emigrated variables

Phase III: *Compile each module separately using its used complier into object code.*

Now each module is compiled separately to produce an object code file with the same name. Save the produced object code files using the names in the Module Table. The produced object files should be of the same format. Any syntax errors are detected here by the used language compiler and to be corrected by the used external editor and have to be preprocessed again. If no errors are detected then link EET table (External Entry Table) is generated for each module. Now all program modules are ready to be linked together.

Phase IV: *Linking and Final executable code generation*. If all modules were compiled successfully then all object files are passed to a common linker to produce the executable code. The common linker has to match all undefined foreign symbols in all EET tables. It generates an augmented Entry External table AEET. From this table all symbols are resolved. If any symbol is not resolved then a linker error is prompted and the error has to be corrected at the source level. Thus the program has to be preprocessed again.

If no errors are detected then an executable code with the name ***name.exe*** is produced and can be run. Figure 3.8 summarize the whole process.

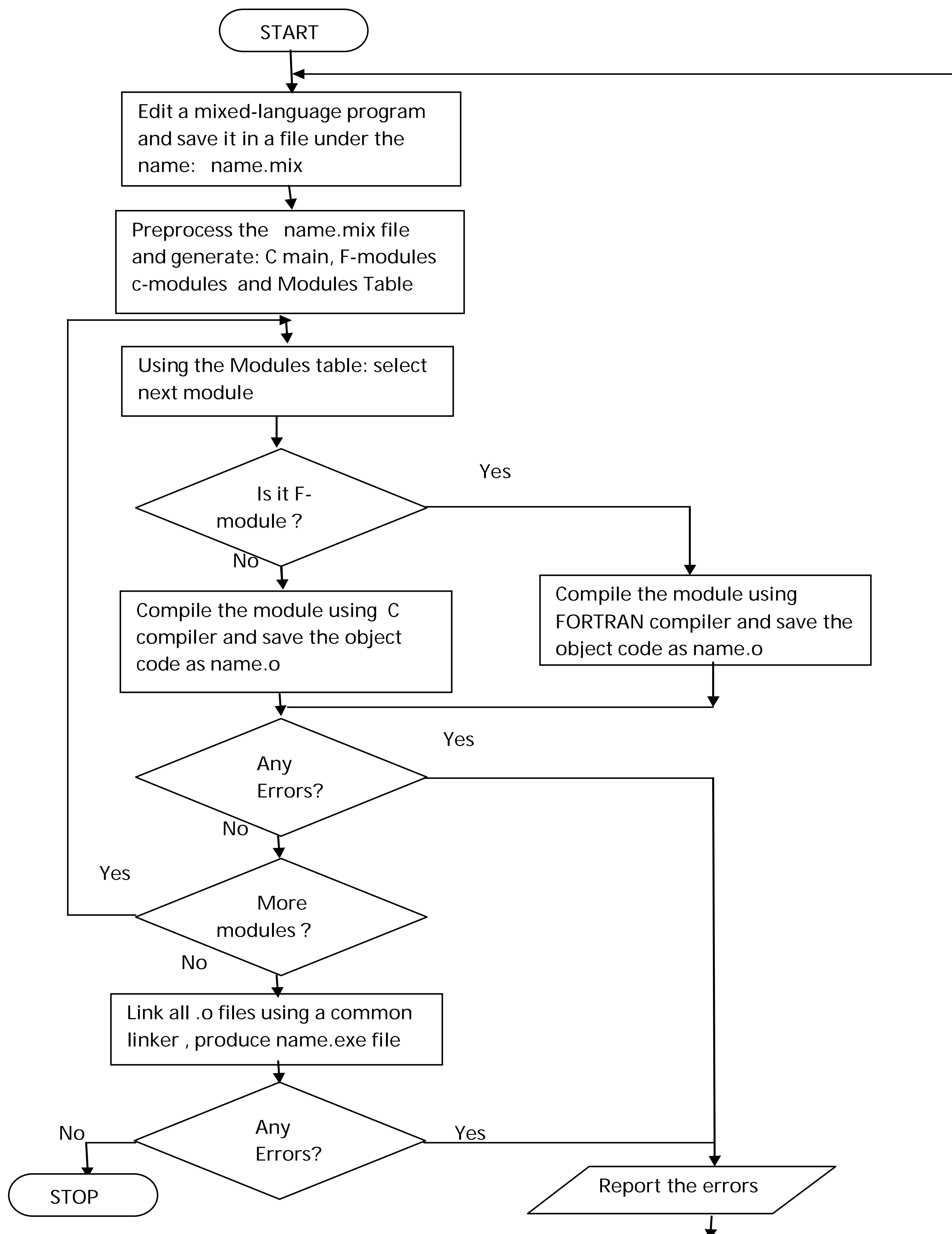


Figure 3.8: Compile each module separately and linking

3.6 Illustration Examples

Three examples are presented here to illustrate the idea and following the given process. The first example uses FORTRAN 77 code lines inserted in the main C function that pass the parameters as arguments. The second example use global area to pass the parameters while example3 doesn't pass any parameters.

Example1: C program uses F-Code section that finds the 10th sum of two integers. The C program reads the two integers then it outputs the result. The whole program in mixed-language format is as follows:

```
%OC Example1
#include <stdio.h>
main() {
        int a, b, sum;
       printf("Enter two integers: ");
      scanf("%d %d",&a, &b);
 %FS AddInteger
    $>, int a, int b, <int sum$
    INTEGER  r
   sum=0
   do r=1 , 10
   sum=sum+(a + b)
10 continue
%CS output
   printf("For a=%d and b=%d ", a , b):
   printf("  the 10th sum is  %d\n", sum);
}
%END
```

List 3.6: Example1, passing parameters as arguments

Now let us preprocess this program and build the required tables and modules.

Phase I: *Splitting the mixed-language program into C and FORTRAN 77 separated modules.*

The given program consists of three sections. The first section is the C main function. The second section is F-code lines and the third section is C-code lines. There is a list with three variables that are emigrated from the C section to a F-code section as arguments. The program ends with END directive statements indicating no more code lines exist.

By following the steps given by the flowchart in Phase I we can build the followings:

the program file is found under the name given by the editor .mix: so we can proceed.

Token1="%OC" Token2="Example1" : the file format is OK.

A file is now created with the name Example1.c and the Modules table looks as follows:

Table 3.5: Example1: Modules table 1:

File Name	Type	Category
Example1.c	C	M

All lines until next "%" are copied directly to Example1.c File since they are C code lines. The File contents will look like this:

Example1.c

```
#include <stdio.h>
main() {
    int a, b, sum;
    printf("Enter two integers: ");
    scanf("%d %d",&a, &b);
```

List 3.7: Example1.c

Now Example1.C is closed and Token1=”%FS” and Token2=”addInteger” so create a file with the name ***addInteger.for*** and insert it in the Modules Table.

Now transfer all lines until "% "to this file. So the file addInteger.for will contain the following code here with the modules table as shown below:

Table 3.6: Example1: Modules table 2

File Name	Type	Category
Example.c	C	M
addInteger.for	F	S

addInteger.for

```
$>, int a, int b,< int sum$
     INTEGER  r
    sum=0
    do 10 r=1 , 10
    sum=sum+(a + b)
10 continue
```

List 3.8: addInteger.for

The same sequence is repeated for %CS output until %END. We get another file contains C code lines with name output.c as shown with the updated modules table:

Table 3.7: Example1: Modules table 3:

File Name	Type	Category
Example.c	C	M
addInteger.for	F	S
Output.c	C	S

output.c

```
    printf("For a=%d and b=%d ", a , b):
    printf("  the 10th sum is  %d\n", sum);
}
```

List 3.9: output.c

So we completed Phase I since we detect the %END directive. The result is three files: ***Example1.c*** ***addInteger.for*** ***ooutput.c*** and their information in the Modules Table. Note the emigration statement exist at the beginning of the FORTRAN 77 file.

Phase II: generating the Emigration Symbol Table (EST table) and building the required statements.

The main C program is **Example1.c** as the first entry from Module Table and it has no EST table**.**

```
#include <stdio.h>
main() {
        int a, b, sum;
        printf("Enter two integers: ");
        scanf("%d %d",&a, &b);
```

List 3.10: Example1.c

- Next entry from the Module Table is a F-section named **addInteger.for** and has an EST table as shown next**:**

Table 3.8: Example1: The EST table 1:

Name	Type	Source	Destination	Modelled name
a	int	C	F	a
b	int	C	F	b
sum	int	C	F	sum

Building the required statements:

call statement: **sum = addInteger_(&a, &b) ;**

struct statement: not needed since the parameters are passed as argument

extern statement: **extern int addInteher_(int *, int *) ;**

The emigration statement is full indicating that the parameters are passed as arguments to the FORTRAN 77 module. So no need for struct statement and the other two statements are inserted in the C module. Now Example1.c file looks like this :

Example1.c

```
extern int addInteger_(int *, int *);
#include <stdio.h>
main() {
        int a, b, sum;
        printf("Enter two integers: ");
        scanf("%d %d",&a, &b);
        sum=addInteger_(&a, &b) ;
```

List 3.11: Example1.c

FORTRAN 77 module encapsulation and adding common statement: Following the process from the flowchart we get the F module encapsulated as follows:

Encapsulation: use EST table to add declarations and naming conventions for the emigrated variables.

addInteger.for

```
Function addInteger(a, b)
INTEGER a
INTEGER b
INTEGER sum

 INTEGER  r
 sum=0
 do 10 r=1 , 10
   sum=sum+(a + b)
  10 continue
addInteger=sum
return

End
```

List 3.12: addInteger.for

- EST table contains the following variables:

Table 3.9: Example1: The EST table 2:

Variable-name /type	Source module	Modified name/type	Destination module
int a	Example1.c	INTEGER a	addInteger.for
int b	Example1.c	INTEGER b	addInteger.for
int sum	Example1.c	INTEGER sum	addInteger.for

- Next entry from Module table shows the next module is a C-code section. So append it to the last C-module file with no added information. Delete its file and entry from the Modules table. The results is as follows:

Table 3.10: Example 1: Modules table 4:

File Name	Type	Category
Example1.c	C	M
addInteger.for	F	M

Example1.c

```
extern int addInteger_(int *, int *);
#include <stdio.h>
main() {
        int a, b, sum;
        printf("Enter two integers: ");
      scanf("%d %d",&a, &b);
      sum=addInteger_(&a, &b) ;
      printf("For a=%d and b=%d ", a , b):
     printf("  the 10th sum is  %d\n", sum);
       }
```

List 3.13: Finale Example1.c

addInteger.for

```
Function addInteger(a, b)
INTEGER a
INTEGER b
INTEGER sum
 INTEGER  r
  sum=0
 do10 r=1 , 10
    sum=sum+(a + b)
 10 continue
addInteger=sum
return
end
```

List 3.14: Final addInteger.for

Phase III: now to compile the modules with the names listed in the Modules Table. The pre-processor sands each module to its language compiler:

Example1.c à **C compiler** à **Example1.o**

addInteger.for à **F77 compiler** à **addInteger.o**

Phase IV: now the pre-processor send all the object code modules (Example1.o, addInteger.o) to the common linker and get an executable code :

Example1.o, addInteger.o à **common linker** à **Example1.exe**

Phase V: run the program, Example1 and check its output !!!

Example2: The same example as above but here the C program calls F-module that performs the addition and a C-Module for outputting the result. Data exchange is done through the common area. C and FORTRAN 77 direct function call does not need much preprocessing since passing the parameters is done through the global area.

The preprocessing of the mixed language file generates the following files:

Example2.c

```
extern void addInteger_(void);
struct{ int a; int b , int sum;} abc_;
int a,b,sum;
#include <stdio.h>
void output(void);
main() {
   printf("Enter two integers: ");
   scanf("%d %d",&a, &b);
    abc_.a=a;
   abc_.b=b;
   AddInteger_();
  a=abc_.a;
  b=abc_.b;
   sum=abc_.sum;
   output();
}
```

List 3.15: Example2.c

AddInteger.for

```
Integer function AddInteger()
INTEGER a
INTEGER b
INTEGER sum
Common/abc/ a, b, sum
    INTEGER  r
   sum=0
   do 10  r=1 , 10
   sum=sum+(a + b)
10 continue
 return
end
```

List 3.16: AddInteger.for in Example2

Output.c

```
void  output()
{
   printf("For a=%d
and b=%d ", a , b):
   printf("  the 10th
sum is  %d\n", sum);
}
```

List 4.17: output.c in

The emigration statement for such a call in the original .mix file is as follows:

$^, int a, int b, int sum, < void $ where the symbol '^' means these variables are emigrated globally through the common area between the C module and the FORTRAN 77 function. There is no returned variables and that is why " < void" is added.

The Modules table during generating these files looks as follows:

Table 3.11: Example2: Modules table 1:

File Name	Type	Category
Example.c	C	M
addIntger.for	F	M
Output.c	C	M

Example 3: A C program calls a FORTRAN 77 function without passing any parameters. The FORTRAN 77 function does its task independently by computing the sum and printing it on the screen. Here only the FORTRAN 77 function has to be declared as extern entity for the C program and called using C convention:

example3.C

```
extern void addInteger_(void);
#include <stdio.h>
main() {
    printf("Now Calling
addInteger ");
        AddInteger_();
    }
```

List 3.18: Example3.c

addInteger.for

```
integer function AddInteger()
   INTEGER  r
   INTEGER  a
   INTEGER  b
   INTEGER  sun
   sum=0
 read(*,*) a,b
   do 10  r=1 , 10
   sum=sum+(a + b)
10 continue
   wite(*,5) sum
5 format('sum=',sum)
 return
end
```

List 3.19: AddInteger in example3

The variable emigration statement for such a call in the original .mix programis as follows: ***$! $*** this means an empty emigration statement!. No variables are emigrated.

All these steps will be shown practically in the next chapter with the evaluation results.

3.7 Summary

As a result from this chapter the following points are concluded:

- Mixing C and FORTRAN 77 77as a one source file or separated source files is possible by the proposed method.
- Variables from a C module are emigrated to a FORTRAN 77 module in three ways: as arguments parameters in the call statement, as global parameters through a common area and no passing at all.
- FORTRAN 77 modules are treated as external entities to C modules.
- A pre-processor is required to implement this method. The pre-processor partitions a mixed-language program written in both languages (C and FORTRAN 77) into separated modules.
- The preprocessor inserts an emigration statement before code sections written in FORTRAN 77. This statements identifies the variables to be emigrated and the way how they emigrate.
- The method utilizes two main features in C and FORTRAN 77 languages. These are C language call by reference is the same as FORTRAN 77 call by name and that extern struct in C language is equivalent to a named common block statement in FORTRAN 77.
- The method works practically if and only if two system programs exist with required conditions:

  1. The compiler for each language produces an object code with standard format as the other compiler does.

2. There exists a common linker that links the produced object codes for both languages and produces an executable machine code.

- The three examples were given only for illustration purpose and will be shown in the next chapter as they appear in practice with real output results and evaluation. There may be some modifications for practical requirements.
- Embedded FORTRAN 77 sections are treated as functions. Real separated functions/ subroutines are treated as modules.

==

Chapter Four: Evaluation and Results

Here are the results obtained from this research. A prototype of a software development environment is developed to evaluate the proposed method. Three program examples are presented here for results evaluation and discussions.

4.1 Evaluation Environment

A software package was developed in C language to be used as an evaluation environment of this research and as implementation of the proposed method. The package consists of a Turbo C editor, a pre-processor, C compiler , FORTRAN 77 compiler and a common linker as shown in Figure 4.1.

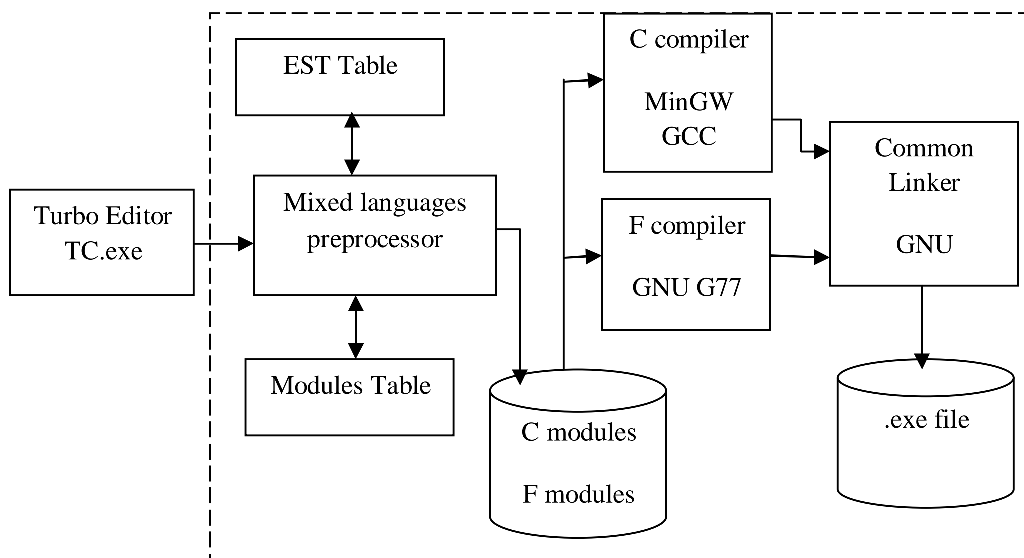


Figure 4.1: Evaluation Environment

The Turbo C editor is used just to enter the source program written in mixed languages C and FORTRAN 77 as a text file and saving it into a file with a given name

and extension .mix. The package calls this editor internally without the user intervention. The source program can be edited and its text can be processed to get a clean source program written in C and FORTRAN 77. The mixed language program should exist in one text file.

A pre-processor that implements the proposed method was developed in C language as given in appendix A. It was built in modules interacting between each other to perform the following tasks:

1. A main module that coordinates all tasks of the software development environment. It calls the Turbo C editor and then the preprocessing functions, the languages compilers and the common linker.

2. Implementing the following preprocessoing functions:

- File Splitting: this function splits the source program written in a mixed languages into C and FORTRAN 77 program Modules. This function utilizes the "%" directive statements. Each separated module is saved into a file with the name given in the directive statement and with extension of its module type. It generates modules table and EST table.

- Encapsulating C and FORTRAN 77 Modules: this function prepares the required extern , call and common statements and inserts them in the proper places in the spitted C and FORTRAN 77 modules. It also encapsulate each FORTRAN 77 module as a function by adding the required head and variables names mapping and declaration.

3. The package uses open source MinGW C comiler to translate C modules into object codes A minimized GNU Compilers Collection for windows), it is a version developed for researches, obtained from GNU Open Source development project group [23], . GNU FORTRAN 77 compiler for FORTRAN 77 modules translation. A common linker from GNU project is used to link all object modules.

The common linker used here is an Open Source GNU version adapted for this application. The original linker is obtained from GNU Open Source development project group [22]. This linker collects all EET tables and builds a common AEET table to resolve the symbols represent the emigrated variables between all modules. Any linking errors are signalled here and if no errors then a final executable code is generated under the name of the original program tag identifier with the extension .exe.

4.2 Examples and results

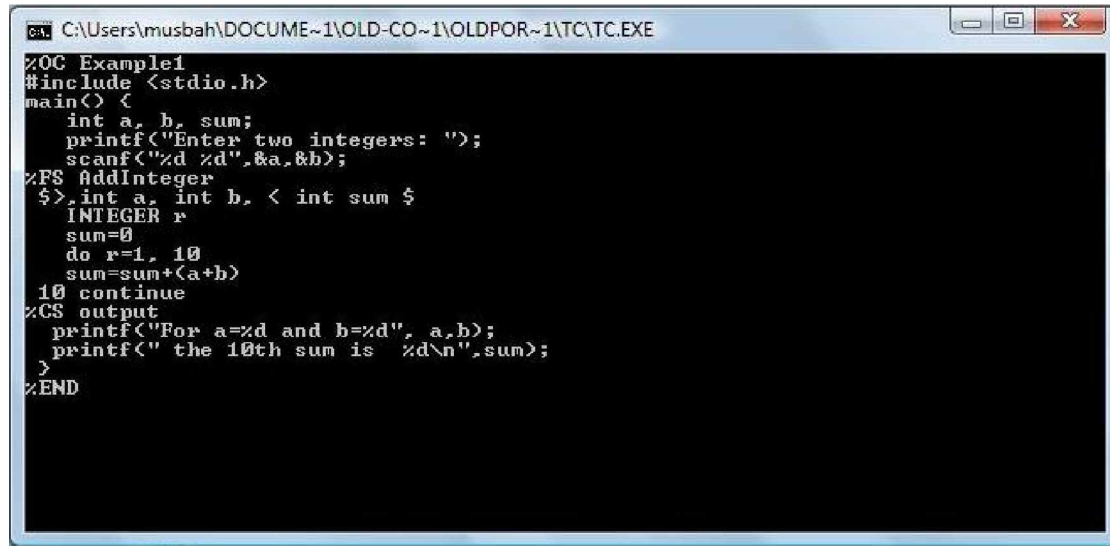
The evaluation environment as it does its main functions it should display the necessary information like the number and the names of generated files, modules table, EST table ... etc. Three examples are presented here to show the results obtained and the capability of the proposed method for programming in a mixed-language environment using C and FORTRAN 77.

Example 1:

Even though the task of the listed program in List 4.1 is not significant but it is taken as an image from the evaluation environment and has the following objectives:

1. How a mixed language program is organized.
2. How to use and where to place the variables emigration statement.
3. How FORTRAN 77 code lines are embedded into a C module.

4. How to pass the variables in the variables emigration statement to the FORTRAN 77 module as arguments.



List 4.1: Example1 Mixed-Language program

The result from the preprocessing of this mixed-language program is given in figure 4.1 next. The Modules table has three entries since the original program has three sections and splited into three files. The first file contains the original C program section and the second file contains the FORTRAN 77 section and the third file holds the rest of the C program.

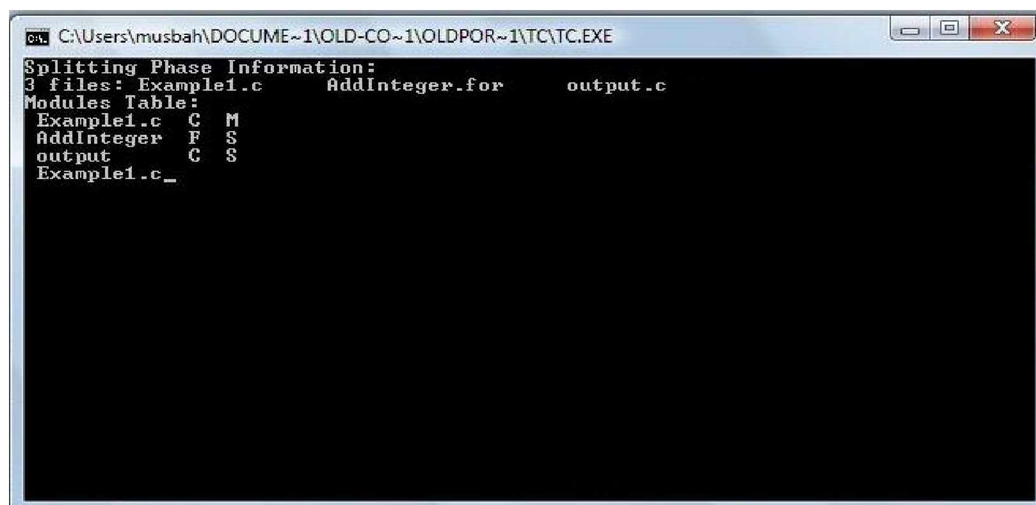


Figure 4.1: Spliting Phase Information

In Figure 4.2 the outputs from the evaluation environment for the second phase are shown. Here the extern statement is build and listed where the FORTRAN 77 module is encapsulated as shown by adding the required haed , variables mapping and declaration. Also the two secition of the C files are combined to form one complete C function. A call statement to the FORTRAN 77 module with the addresses of the emigrated variables is inserted.

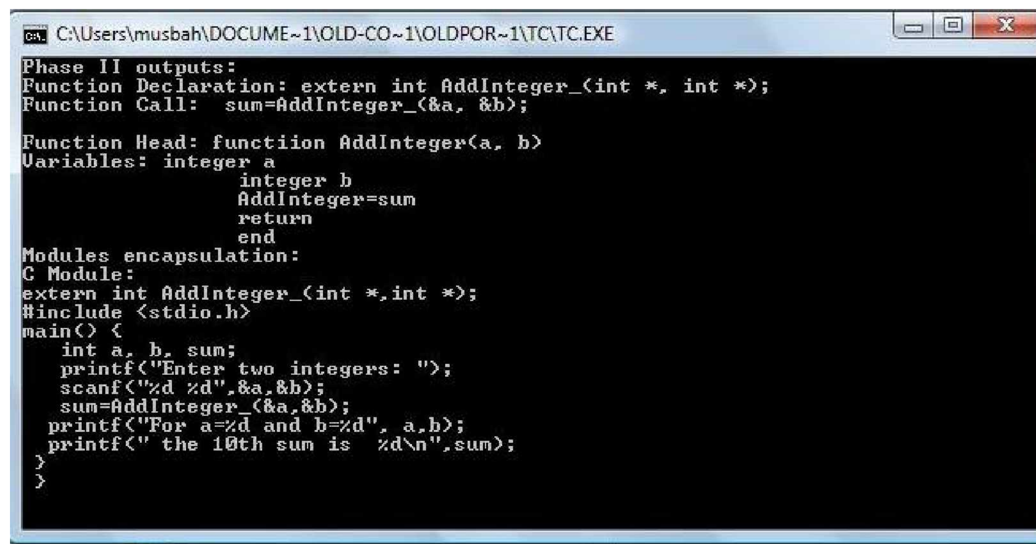


Figure 4.2: Phase II outputs

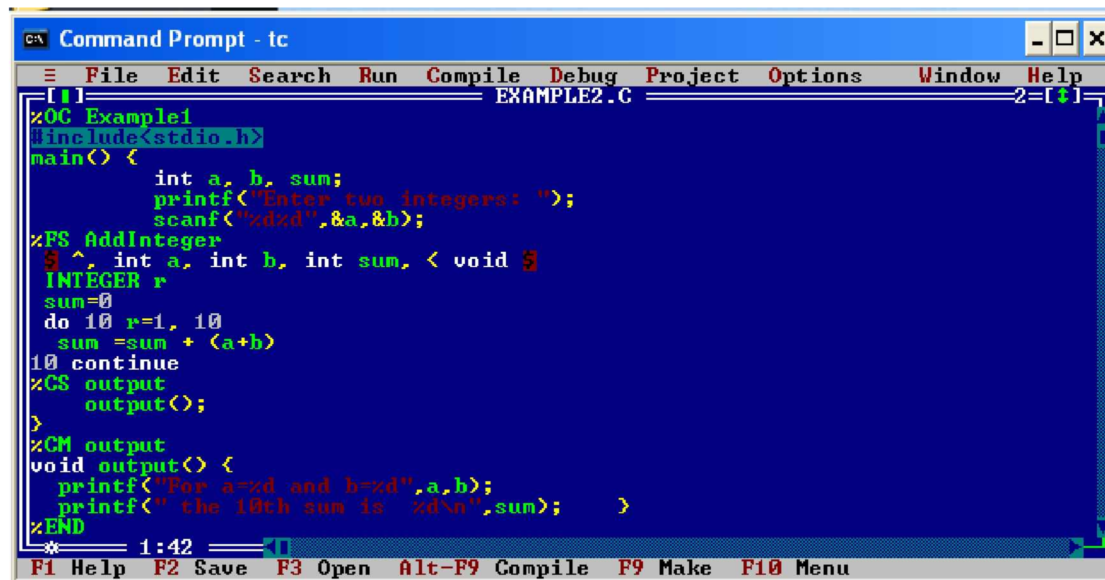
Now for the third phase, each module is passed to its corresponding compiler the response of each compiler is shown in Figure 4.3.The modules table information is used to select which compiler to use for which module. At this stage two object code files are obtained as denoted in the figure.

Figure 4.3: Compilation and Linking

Figure 4.3 also shows the common linker output. The two object code files are linked and one executable file is generated. The package prompts the user whether to run this executable program or not. Here the program is run and the results are as shown. Exactly what is expected from tracing the original program.

Example 2:

The same original mixed language program is used here but the parameters are emigrated through the global area. As shown in List 4.2, the variables emigration statement shows that the listed variables are to be passed globally to the next FORTRAN 77 code section. The program still retains its original format. Therefore this example has the same objective and i addition it illustrated the second strategy of parameters emigration.



List 4.2: Example2 Mixed-Language program

Figure 4.4 presents the same outputs as for example1 and we get three files. The interesting output for this example is the construction of the extern struct for the C module and the common statement for the FORTRAN 77 module as presented in Figure 4.5. The call statement uses no parameters since all parameters are passed between the extern struct and the common area.

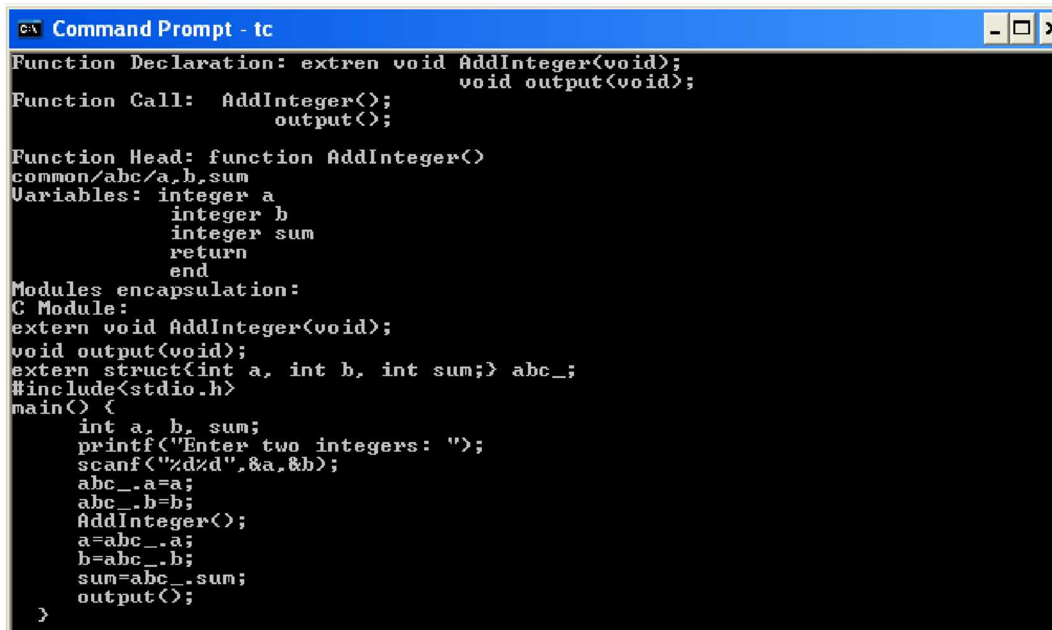
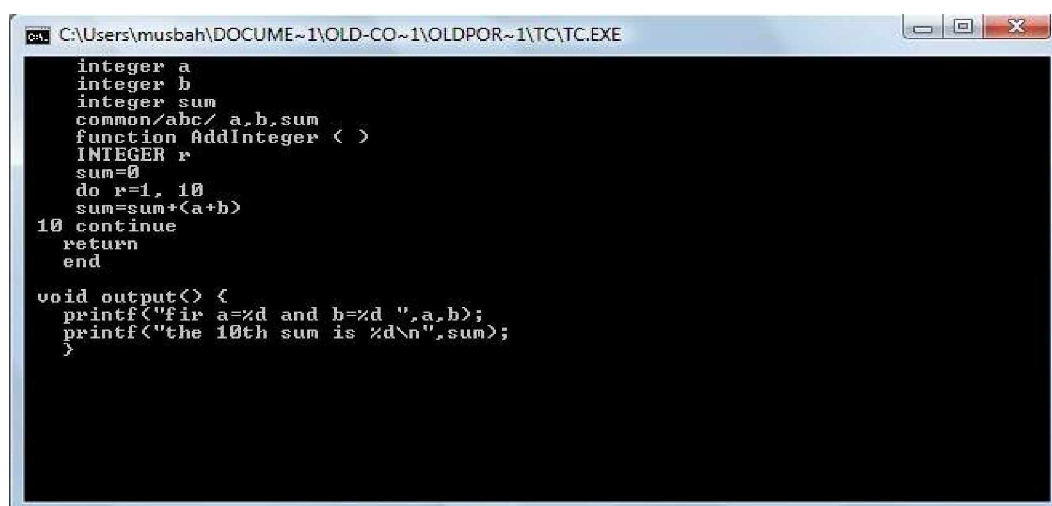


Figure 4.5: Extern struct and Common area statements

In the second phase, the extern struct is added to the C module with a definition of the FORTRAN 77 module as an external entity. For the FORTRAN 77 module the common statement with the required parameters is added at the begging of the FORTRAN 77 module. List 4.3 presents the complete encapsulated C and FORTRAN 77 modules.



List 4.3: Encapsulated C and FORTRAN 77 Modules

The output from running the generated executable code is given in Figure 4.6. The same results are obtained which verifies the capability of the proposed method.

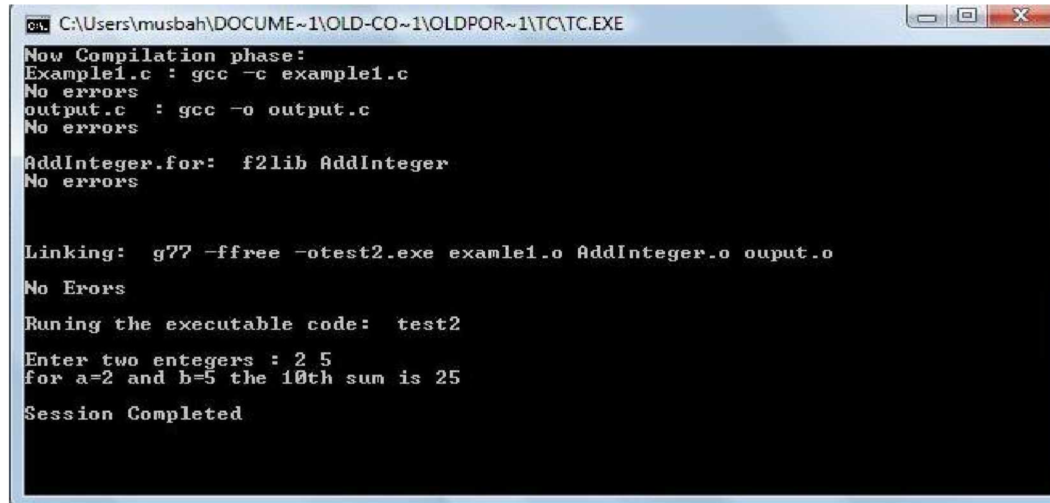


Figure 4.6: Run output of Example2

Example3

Here the third way to pass the parameters is tested. The C module call the FORTRAN 77 module with its name without passing any parameters. The FORTRAN 77 module does its task independently. Thus the mixed-language source program, it has just a call. Here still a variable emigration statement and the processor directives are needed. The emigration statement is empty indicating no parameters are passed. List 4.4 presents the original program showing two parts but in one file: the first part has the original program in the mixed-language format, while the second part gives the standalone FORTRAN 77 function.

```
C:\Users\musbah\DOCUME~1\OLD-CO~1\OLDPOR~1\TC\TC.EXE
#include <stdio.h>
main() {
    int a, b, sum;
    printf("Enter two integers: ");
    scanf("%d %d",&a,&b);
%FS AddInteger
 $!$
    AddInteger();
%CS output
   printf("For a=%d and b=%d", a,b);
   printf(" the 10th sum is  %d\n",sum);
 }
%FM
   function AddInteger ()
   INTEGER r,a,b,sum
    sum=0
    read(*,*) a,b
    do r=1, 10
    sum=sum+(a+b)
 10 continue
    write(*,5) sum
 5  format('sum=',sum)
     return
     end
%END_
```

List 4.4: Example3 Modules

Since the emigration statement is empty, the pre-processor treats the FORTRAN 77 module as an external function to the C module. Thus it just moves the FORTRAN 77 module into a separate file as it is without modification. In the C module the pre-processor adds the extern declaration without any parameters for the FORTRAN 77 module. The call statement remains as its and the pre-processor directives are removed.

By preprocessing this mixed-language program the outputs shown in Figure 4.7 are obtained.

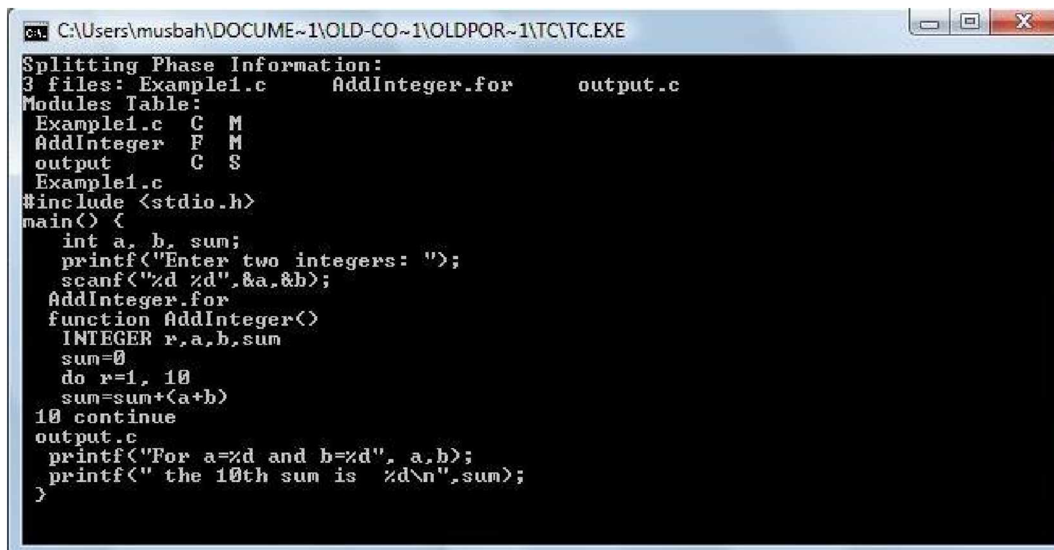


Figure 4.7: Preprocessor outputs for Phase I

The interpretation of the given output is the same as for the previous examples. When both files are compiled and linked an executable file is obtained and it produced the shown results in Figure 4.8 when it was executed. The same results are obtained as in the previous examples.

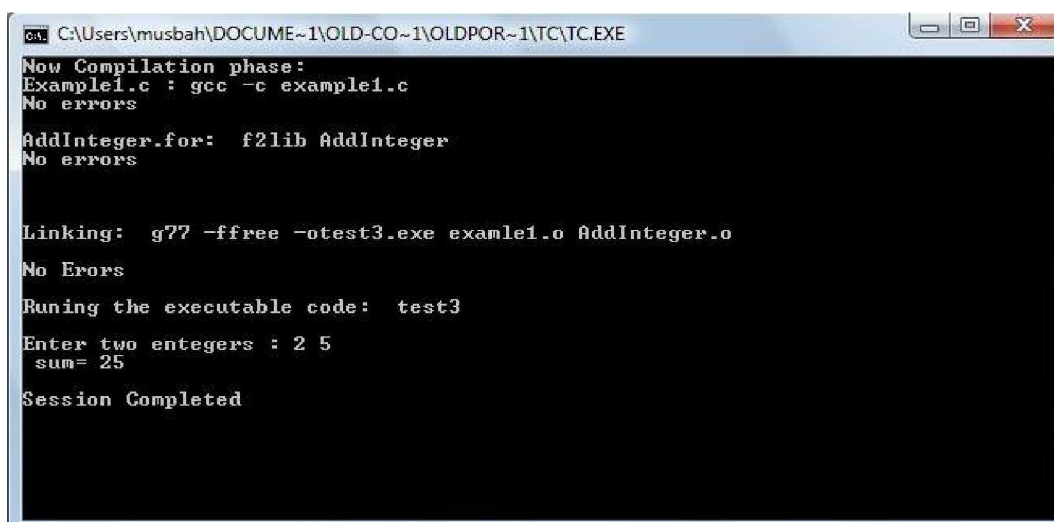


Figure 4.8: Example3 results

4.3 Method performance

By analyzing the results obtained from the evaluation environment using the three examples presented in section 4.2, the following points summarizes the capabilities of the proposed method and the performance of its pre-processor:

1. Using the evaluation environment a programmer can develop a mixed-language programs written in C and FORTRAN 77 freely.
2. The outputs for the three strategies of parameters emigration is the same, but their usage depends on the programmer and the program organization.
3. Three ways of variables emigration can be used and mixed. These are : passing parameters as arguments in the call statement, passing parameters as global arguments through a common area and no parameters are passed where a FORTRAN 77 function does its task independently.
4. In practice these three options of parameters passing may be mixed or selecting one type for a certain module and other type for another module. They are presented here separately to show how they are used and how the environment implements them.

4.4 Applications of the method

Mixed-Languages Programming is a field of challenge today. Many researches and many methods are developed for this purpose. They share the idea to get the benefits from different programming languages for an application and to overcome the gape of programs conversions from one language to another. The method presented in this research is one of the attempts in this direction of programming due to the fact that C and FORTRAN 77 are widely used past and present languages.

The applications of such a mixed-language programming methodology are summarized as follows:

1. The method enables programmers to get use of the data types and procedures from more than one language.

2. In critical applications N-version programming is often used to get more reliable outputs. By this methodology more reliable results are obtained from one program. The critical section may be written in the language that provides reliable procedures and accurate data types that best models the real world data.

3. Parallel processing of different sections written in different programming languages and can be executed on different hardware platforms.

4. For educational process it gives a sense of the differences and similarities between different programming languages and how data is represented in memory and passed between program modules.

5. Verification and acceptance test modules can be built using this method since this method can embed the testing module where ever it is necessary.

6. Normal programmers do not know how to mix C and FORTRAN 77 . So by this method simply a programmer develops his programs using C and FORTRAN 77 as if he were programming with a homogeneous programming language that supports both C and FORTRAN 77 data types. The programming goes smoothly with no need from the programmer to know how to map these data types or how to organize the program different modules. Thereby this method increase programmer productivity and enhances software reliability and efficiency.

7. Old developed FORTRAN 77 modules are reused at the source and object code levels using this method.

8. This method can be used for both Modules interfacing (interacting modules) and mixed modules.

9. By mixing C and FORTRAN 77 based on data variable emigration, a new direction in programming has been produced. The software development environment built in this research is simple, effective and has the benefit of original languages compilers so it did not change any language's syntax.

==

Chapter Five: Conclusion and Future Work

As the end of this work, some conclusions and remarks on this research are presented. The conclusions present what have been achieved and the results obtained from this research. Whereas the future work presents what have been left for further investigation and implementation.

5.1 Conclusion

In this research, The focus was on the data types supported by C and Fortran-77 programming languages since they are selected as the platform of this research. Both languages support almost similar data types with different sizes, binding and memory allocation. Data exchange between such languages led to investigate their program organization. The outcome was there is an intersection in this direction. Both languages use functions as sub programs. Also both languages can pass parameters by references and names which is equivalent. On the other hand passing the parameters between program modules globally is equivalent in both languages for extern struct in C and common block statement in Fortran-77.

As a result from the above investigation a method was proposed and developed to mix C and Fortran-77 programming in one source program. The method solves many problems of programs conversions and interfacing problems between the two selected languages. It was described fully with demonstration examples. A peprocessor was developed with special directives to implement this method. To keep the programmer comfortable with the proposed method, the normal language syntax is unchanged and the languages normal compilers are used. A common linker is imported and used from the GNU group.

A complete evaluation environment is integrated that includes a Turbo C editor, the pre-processor, the languages compilers and the common linker. The environment was then used to evaluate the capabilities of the proposed method. Three examples each using a different strategy for variables emigration as parameters passing were developed and tested on this environment. The results obtained verified the performance and proofed the capabilities of the proposed method.

5.2 Future Work

Since any work can not be completely established, here are some future work activities to enhance or to add some capabilities for the proposed method:

1. More data types including compound and user defined have to be added especially those compound ones.

2. Considering Fortran Subroutines as modules. This requires a little addition in the pre-processor to build the required head and considering the subroutine organization and calling requirements.

3. Making the environment to call C functions from Fortran modules. The same idea is to be applied and the same added programming constructs are to be used reversely.

4. The same approach can be followed to add more languages in the environment.

==

References

1. **-"A Fortran-to-C Converter"** *S. I. Feldman, David M. Gay, Mark W. Maimone and N. L. Schryer* issued May 16, 1990. Bell Communications Research,
2. " **Data representation for mixed-language program development**" Niklas Gustafsson, John Hamby- Issue date: May 23, 2000- Assignee: Instantations, Inc.
3. "C -Programming Tutorial"" *Alfred Park 2003 - Georgia Institute of Technology.*
4. **"cfortran.h , An interface between C and Fortran 77"** - Experiments Division Programming Group, *European Synchrotron Radiation Facility*.
5. "**Concepts of Programming Languages**" Robert W. Sebesta 4th Edition (2006) Publisher*:* Addison-Wesley.
6. "**Constant values in mixed language programming environments**" Lawrence R. Schwarcz - Issue date: Nov 8, 2005- -Assignee: Hewlett-Packard Development Company, L.P.
7. " **FORTRAN 77 Language Fundamentals and style"** Walter S. Brainerd, Charles H. Goldberg and Jonathan L. Gross **-** (July 1985)- Heinle & Heinle Pub.
8. GNU Open Source The Free Software Foundation's software.
9. The minimal GNU list for windows, GNU Open Source 26\10\2008.
10. "**Mixed Language Programming – Fortran**"" Mike Eddy - 2000 - Bell Communications Research, Morristown.
11. "**MIXED LANGUAGE PROGRAMMING REALIZATION AND THE PROVISION OF DATA TYPES**" Bo Einarsson -Computer Centre LIDAC - University of Linkoping.

12. “**Programming languages design and implementation**” Terrence W. Pratt and Marvin V. Zelkowitz - second edition 1999 – Publisher: prentice hall.
13. "**PROGRAMMING LANGUAGE DESIGN CONCEPTS** " David A. Watt, University of Glasgow with contributions by William Findlay, 2004 Publisher: John Wiley & Sons Ltd,
14. **"Professional Programmer's Guide to Fortran**", Clive G. Page, 2005 - University of Leicester UK .
15. *"The C Book*" Mike Banahan, Declan Brady and Mark Doran, second edition, published by Addison Wesley in1991.
16. **"Using C and C++ with Fortran"** - *Nelson H. F. Beebe* – **2001**
17. "**Variable Scope for New Programmers"** – Jone Dely - Digital Web Magazine - www.digital-web.com/articles/variable_scope_for_new_programmers/ - 19\5\2007

===

Appendix A:

"The Preprocessor Program List"

```
/*********************************************************************/

 */A Mixed-Language Preprocessor for M.Sc. Thesis requirement/*

 */Developed by: Dr. Musbah M. Elahresh as a supervisor of the Thesis and Rehab Abdalla as
a student/*

 */Date of Development:  May 2009/*

 */Last Date Modified:   04/08/2009/*

/*********************************************************************/

#include <stdio.h<

#include <string.h<

#include <process.h<

#include <io.h<

struct}

    char fname[20؛[

    char type[3]; /* type and category merged here/*

 {    ModTab[20؛ [

char EmgSt[20][20]; /* holds current Emigra   on statement/*

int Emgnum=0; /* index counter for EmgSt/*

char call[80]; /* call statement/*

char retst[25];  /* assignement return value statement in F module/*

char strct[80]="struct {\n"; /* exten struct statement/*

char strname[10]; /* shared name of struct and common statements/*

char comon[80]="common /"; /* common satement/*

char Head[80]="func   on؛"

char DecSt[20][20]; int Decnum=0؛

char Fdec[80]="extern؛"

int tp=0؛

char varlist[20][20];   /* variables list table/*

int varnum=0؛

int split(FILE *); /* implements Phase I/*
```

```
void encapslt( int );  /* implements Phase II recieves number of modules/*


void complink(int); // compile and link

char DcTab[8][20]={"int","integer","float","real","char","character","double","double
precision؛{"

int Tabnum=8;  /* number of entries in DcTab/*

int checkname(char؛(*

void ErrorMsg(int؛(

void merg(int؛(


/*******************************************************************/

  */Main program coordinated PhaseI, Phase II, Phase III, Phase IV/*

/*                                                                  */

/*******************************************************************/

main} ()

  FILE *foc؛

  char OCname[15]="", mask[25]="TC؛"

  int flag, numfiles,i؛

  clrscr؛()

  printf(" WELCOME TO MIXED-LANGUAGES PROGRAMING ENVIRONMET\n\n؛("

  printf("Enter Mixed-Language file name؛(" :

  scanf("%s", OCname؛(

*/   flag=checkname(OCname);  check for %OC signature/*


  strcat(mask,OCname؛(

*/   system(mask/*؛(

  if(flag==1) ErrorMsg(flag؛(

  foc=fopen(OCname,"r؛("

  numfiles=split(foc؛(

  printf(" \n\nModules Table: \n؛("
```

```
   for(i=0;i<tp;i(++
      printf("%s   %s  \n", ModTab[i].fname, ModTab[i].type؛(

   printf("\n Split complete. Check [%d] files\n",numfiles؛(
   getch؛()
   printf("\n\n Now Encapsulation: \n؛("
   encapslt(numfiles؛(
   printf("Encapsulation completed. Press any Key "); getch؛()
   merg(numfiles);  /* combines separated C sections/*
   printf(" Merge is completed. Press any key\n"); getch؛()


-------//
   complink(numfiles؛(
   printf(" compile and link are completed, run . Press any key\n"); getch؛()

{

/**********************************************************/
 */split: implements Phase I. It splits the mixed languages/*
 */program into C and Fortran modules. It saves each module/*
 */min a separate file. Builds the Modules Table andr returns/*
 */number of modules/*
/**********************************************************/

int split(FILE *fp(
}
   char line[80],c,type[3],name[20؛[
   FILE *mp؛
   int number=0؛
```

```
   printf(" Now Splitting: press:\n؛("
   while(!feof(fp} ((
     fgets(line,80,fp؛(
     printf("%s",line؛(

    if(line[0} ('%'==[
          fclose(mp؛(
          sscanf(line,"%c %s %s",&c,type,name؛(
          if(strcmp(type,"END")==0} (
               fclose(mp);fclose(fp؛(
               break{؛
          */printf("Tag=%c type=%s  Name=%s\n",c,type,name/*؛(
          if(type[0]=='C') strcat(name,".c؛("
             else strcat(name,".for؛("
          strcpy(ModTab[tp].fname, name؛(
          strcpy(ModTab[tp++].type,type؛(
          */printf("name=%s \n",name/*؛(
          mp=fopen(name,"w"); number؛++
          fgets(line,80,fp؛(
          printf("%s",line؛(
*/{       if/*
   if(feof(fp)) line[strlen(line)-1]='\0؛'

      fputs(line,mp؛(
*/{   while/*

    //line[strlen(line)-1]='\0؛'
     //  fputs(line,mp؛(

   printf("\n \aEND OF SPLIT"); getch؛()
```

```
   return number؛
{
/************************************************************/
*/encapsIt: utilizes the Emigration sataement and builds the/*
 */struct  common and call satements. It then encapsulates/*
 */each module to be ready for Phase III/*
/************************************************************/
char make(char *, char *); /* help function to build the required sts/*
void InStrct(char *,char);  /* insert external struct and call statments/*
void InsCmn(char *,char); /* inserts common statement/*

void encapsIt(int nmod} (
  int m؛
  FILE *tmp؛
  char line[80], name[20],pass؛
  for(m=0;m<nmod;m++) {    /* scan the ModTab and encapsulate each module/*
    if(strcmp(ModTab[m].type,"FS")==0} (
     tmp=fopen(ModTab[m].fname,"r؛("
     fgets(line,80,tmp؛(
     printf("Now preparaing call, extern struct, common statements: \n؛("
     pass=make(line,ModTab[m].fname); /* build the required statements/*

     printf("\n\nNow inserting into C module: \n");getch؛()
     InStrct(ModTab[m-1].fname,pass); /* to be modifed for any CM uses FS/*

     printf("\n\nNow Inserting into Fortran Module:\n");getch؛()
     InsCmn(ModTab[m].fname,pass؛(
*/{    end if/*
```

```
*/ {  end for m/*

{

/*********************************************************************/
 */make: a help function constructs the extern struct, call, common/*
        */statements. Called from encapsulate function/*
/********************************************************************/
char *MapDc(char؛(*
char *MapVr(char؛(*
char make(char stat[80], char mname[20([
}
*/  NOTE: no syntax cheking for the Emigration Statement is made here/*
  char c, var[20]="",dec[20]="",trm[20؛""=[
  int k,j=0,L=strlen(stat؛(

*/  printf("LINE====%s \n ",stat/*؛(
  mname[strlen(mname)-4]='\0؛'

  for(k=0;k<L;k} (++
   c=stat[k]; /*printf("c=%c ",c/*؛(
    if((c=='$')) continue؛
     if(c} ('،'==
       strcpy(EmgSt[Emgnum++],var);strset(var,'\0');j=0;con  nue{؛
    var[j++]=c؛
*/{  for k/*
  strcpy(EmgSt[Emgnum++],var؛(
  printf("\n Emigration Statement Entries: %d \n ",Emgnum؛(
  for(k=0;k<=Emgnum;k(++
    printf("%s  ",EmgSt[k؛([
```

```
*/   printf("\nMMMMM=%s",EmgSt[0/*؛([
  strcpy(var,EmgSt[0؛([

  for(k=0;k<strlen(var) ;k++)  if(var[k]==' ') continue؛
                                                          else { c=var[k]; break{ ؛
*/  printf("\a\a\n C=%c",c); getch();*/getch؛()
  switch(c} (
          case '>': /* passing parameters as arguments in function call/*
                    */    eventhough the same is repeated for each case, leave it/*

                    */  now prepare call statement/*
                     strcpy(trm,EmgSt[Emgnum-1]);sscanf(trm,"%s %s",dec,var؛(

                     if(strcmp(dec,"<")==0) {sscanf(trm,"%s %s %s",dec,dec,var{ ؛(

                if(strcmp(dec,"void")!=0} (
                       strcat(call,var);strcat(call{  ؛("="،
                     else {sscanf(dec,"%s",var);strcat(call{؛(""،



               puts(trm؛(
                     puts(dec؛(
                     puts(var؛(

                     strcat(call,mname);strcat(call؛(")_"،
                     for(k=1;k<Emgnum-1;k}(++
                       sscanf(EmgSt[k],"%s %s",var,var؛(
                       strcat(call,"&"); strcat(call,var);strcat(call؛("، "،
                  {
                call[strlen(call)-2]='\0؛'
```

```
   strcat(call؛("؛("،
   printf("Call st: %s \n",call); getch؛()

        */  prepare return assignment statement/*
strcpy(retst؛(""،
          strcpy(trm,EmgSt[Emgnum-1]);sscanf(trm,"%s %s",dec,var؛(
         if(strcmp(dec,"<")==0) {sscanf(trm,"%s %s %s",dec,dec,var{ ؛(

if(strcmp(dec,"void")!=0} (
                    strcat(retst,mname);strcat(retst؛("="،
                    strcat(retst,var); strcat(retst,"\n\0{ ؛("
          printf("retst: %s ", retst);getch؛()

         */  now declare F-module as externl function with parameters/*
          strcpy(trm,EmgSt[Emgnum-1؛([
          sscanf(trm,"%s %s",dec,var؛(
          if(strcmp(dec,"<")==0) {sscanf(trm,"%s %s %s",dec,dec,var{ ؛(
          strcat(Fdec,dec);  /*returned type/*
   strcat(Fdec؛(" "،
   strcat(Fdec,mname); strcat(Fdec؛(") _"،
         for(k=1;k<Emgnum-1;k++){ /*arguments list as pointers/*
           sscanf(EmgSt[k],"%s %s",dec,var؛(
           strcat(Fdec,dec؛(
           strcat(Fdec؛("* "،
           strcat(Fdec؛("، "،
        {
          Fdec[strlen(Fdec)-2]='\0؛'
          strcat(Fdec,");\n؛("
          printf("Fdec st: %s \n",Fdec); getch؛()
```

```
*/  now prepare the function head and declaration statement/*
        strcat(Head,mname);strcat(Head؛(")"،
            for(k=1;k<Emgnum-1;k}(++
              sscanf(EmgSt[k],"%s %s",dec,var؛(
              strcat(Head,var); strcat(Head؛("،"،
           {
        Head[strlen(Head)-1]='\0؛'
        strcat(Head,")\n؛("
        printf("Head st: %s \n",Head؛(
            getch؛()

 */  declaration staements/*
            for(k=1;k<Emgnum;k++){ /*arguments list as pointers/*
              sscanf(EmgSt[k],"%s %s",dec,var؛(
              if(strcmp(dec,"<")==0) {sscanf(trm,"%s %s %s",dec,dec,var{ ؛(
              strcpy(dec,MapDc(dec)); /* map variables declaration/*
           */    strcpy(var,MapVr(var);  Map variable name:   later/*
              strcat(DecSt[k-1],dec);strcat(DecSt[k-1],"  ");strcat(DecSt[k-1],var؛(
              strcat(DecSt[k-1],"\n؛("
           {
           Decnum=k-1؛
           printf("DecSt=    \n؛("
           for(k=0;k<Decnum;k(++
           printf("      %s",DecSt[k؛([
      getch؛()
           break؛

case '^': /*  passing parameters as global variables/*


                       */now prepare call statement/*
```

```
strcpy(trm,EmgSt[Emgnum-1]);sscanf(trm,"%s %s",dec,var؛(
if(strcmp(dec,"void")!=0} (
  strcat(call,var);strcat(call{  ؛("="،
else {sscanf(dec,"%s",var);strcat(call{؛(""،
strcat(call,mname);strcat(call,"_();\0؛("
printf("call st: %s ",call);getch؛()

*/   prepare assigmnets/*
for(k=1;k<Emgnum;k}(++
  sscanf(EmgSt[k],"%s %s",dec,var؛(
  if(strcmp(dec,"void")==0) con  nue؛
  strcpy(varlist[k-1],var);varnum=k؛
{

*/   now declare F-module as externl function with No parameters/*
strcat(Fdec,dec);  /*returned type/*
strcat(Fdec؛(" "،
strcat(Fdec,mname); strcat(Fdec,"_();\n\0؛("
printf("\n Fdec: %s",Fdec؛(
getch؛()

*/  now prepare extern struct and assinments of global variables/*
for(k=1;k<Emgnum;k}(++
  sscanf(EmgSt[k],"%s %s",dec,var؛(
  if(k==1) {strcpy(strname,var);strcat(strname,"_\0{؛("؛
  if(strcmp(dec,"void")==0) con  nue؛
  strcat(strct,dec؛(
  strcat(strct؛(" "،
  strcat(strct,var؛(
  strcat(strct,";\n؛("
```

```
{
strcat(strct؛(" {"،
strcat(strct,strname);strcat(strct," ;\n\0؛("
printf("\n\nstrct= %s\n",strct؛(
getch؛()

*/now prepare the function head, common and declaration statement/*
strcat(Head,mname);strcat(Head,"( )\n\0؛("
printf("Head st: %s \n",Head؛(
getch؛()
*/ now prepare common statements/*
*/ and map the conventions/*
strname[1]='\0؛'
strcat(comon,strname); strcat(comon؛(" /"،
for(k=1;k<Emgnum;k}(++
sscanf(EmgSt[k],"%s %s",dec,var؛(
if(k<Emgnum-1) strcat(comon,var؛(
if(k<Emgnum-2)strcat(comon؛(" ،"،
{
strcat(comon,"\n\0؛("
printf("comon=%s \n",comon); getch؛()

for(k=1;k<Emgnum;k++){ /* declara ons/*
sscanf(EmgSt[k],"%s %s",dec,var؛(
if(strcmp(dec,"void")==0) con nue؛
strcpy(dec,MapDc(dec)); /* map variables declaration/*
*/ strcpy(var,MapVr(var); Map variable name: later/*
strcat(DecSt[k-1],dec);strcat(DecSt[k-1]," ");strcat(DecSt[k-1],var؛(
strcat(DecSt[k-1],"\n؛("
{
```

```
        Decnum=k-1؛
        printf("DecSt=   \n؛("
        for(k=0;k<Decnum;k(++
        printf("      %s",DecSt[k؛([
    getch؛()
         break؛

case '!': /* no parameters are passed. Standalone function/*
                  */ now prepare call statement/*
                   strcpy(trm,EmgSt[Emgnum-1]);sscanf(trm,"%s %s",dec,var؛(
                  if(strcmp(dec,"void")!=0} (
                    strcat(call,var);strcat(call{  ؛("="،
                  else {sscanf(dec,"%s",var);strcat(call{؛(""،
                  strcat(call,mname);strcat(call,"_();\0؛("

                  */ now declare F-module as externl function with No parameters/*
                  strcat(Fdec,dec);  /*returned type must be void/*
                  strcat(Fdec؛(" "،
                  strcat(Fdec,mname); strcat(Fdec,"_ ();\n\0؛("
                  printf("\n %s",Fdec؛(
                  getch؛()

        */    now prepare the function head and declaration statement/*
                 strcat(Head,mname);strcat(Head؛(")"،
                 strcat(Head,")\n\0؛("
     printf("Head st: %s \n",Head؛(
     getch؛()
          break؛
default: printf("\n\a  Error in: Variables Emigration Statement\a؛("
          prin   ("press any key to exit"); getch(); exit(0); break؛
```

```
*/ {  switch/*

 return c؛

{

/*****************************************************************/
 */Mapping variables names and declarations to Fortran conventions/*
/*****************************************************************/

char *MapDc(char *v(
}
  int i؛
  for(i=0;i<Tabnum;i=i+2(
  if(strcmp(DcTab[i],v)==0) return DcTab[i+1؛[
  printf("\a\a\n Error in Emigration Statement: %s \n",v);exit(0؛(
  return NULL؛
{
char *MapVr(char *Vr(
}

{

/********************************************************************/
 */InStrct: a help function inserts the extern, struct, call staements/*
```

```
*/and assignment at bigen and midle of C module/*
/*********************************************************************/

void InStrct(char *name,char c(
}
  FILE *tmp, *fp؛
  char line[80؛[
  int k؛
  fp=fopen(name,"r؛("
  tmp=fopen("tmp.c","w؛("
  switch(c} (

          case؛    :'!'

          case '>':  fprintf(tmp,"%s",Fdec؛(
                  while(!feof(fp} ((
                    fgets(line,80,fp؛(
                    if(feof(fp)) break؛
                    fprintf(tmp,"%s",line؛(
                  {
                  fprintf(tmp,"%s",call؛(
                  fclose(fp); fclose(tmp؛(
                  tmp=fopen("tmp.c","r؛("
                  fp=fopen(name,"w؛("
                  while(!feof(tmp} ((
                    fgets(line,80,tmp؛(
                 */   if(feof(tmp)) break/*؛
                    fprintf(fp,"%s",line؛(
                  {
                  fclose(fp); fclose(tmp؛(
```

```
                break؛

            case '^': fprintf(tmp,"%s",Fdec؛(
                            fprintf(tmp,"%s",strct؛(
                     while(!feof(fp} ((
                            fgets(line,80,fp؛(
                            if(feof(fp)) break؛
                            fprintf(tmp,"%s",line؛(
                           {
            for(k=0;k<varnum;k} (++
                     fprintf(tmp,"     %s_.%s=%s;\n", strname,varlist[k],varlist[k؛([
                 {
                    fprintf(tmp,"%s",call؛(
                    for(k=0;k<varnum;k} (++
                     fprintf(tmp,"\n     %s=%s_.%s;",varlist[k],strname,varlist[k؛([
                 {
                    fclose(fp); fclose(tmp؛(
                    tmp=fopen("tmp.c","r؛("
                    fp=fopen(name,"w؛("
                    while(!feof(tmp} ((
                           fgets(line,80,tmp؛(
                           fprintf(fp,"%s",line؛(
                 {
                    fclose(fp); fclose(tmp؛(
                    break؛

*/{    switch/*

{
```

```
/*******************************************************************/
 */InCmn: a help function inserts common, return assigmnt  statemnt/*
            */at begin of  fortran module/*
/*******************************************************************/
void InsCmn(char *name,char c(
}

   FILE *tmp, *fp؛
   char line[80؛[
   int k=0؛
   strcat(name,".for؛("
   fp=fopen(name,"r؛("
   tmp=fopen("tmp.c","w؛("
   switch(c} (

    case؛      :'!'

    case '>': fprintf(tmp,"\t%s",Head؛(
                 for(k=0;k<Decnum;k(++
                          fprintf(tmp,"\t%s",DecSt[k؛([
                        fgets(line,80,fp); /* to skip emg. statemnet/*
                 while(!feof(fp} ((
                       fgets(line,80,fp؛(
                       if(feof(fp)) break؛
                       fprintf(tmp,"%s",line؛(
                       {
                       fprintf(tmp,"\t%s",retst؛(
                fprintf(tmp,"\treturn\n؛("
                fprintf(tmp,"\tend؛("
```

```
fclose(fp); fclose(tmp؛(
tmp=fopen("tmp.c","r؛("
fp=fopen(name,"w؛("
while(!feof(tmp} ((
  fgets(line,80,tmp؛(
  printf("%s",line);/*getch/*؛()
  fprintf(fp,"%s",line؛(
{
   fclose(fp); fclose(tmp؛(
   printf(" \n\n\n\a Insert into F-module  finish. Press any key ");getch؛()
   break؛

case '^': fprintf(tmp,"\t%s",Head؛(
            for(k=0;k<Decnum;k(++
                  fprintf(tmp,"\t%s",DecSt[k؛([
            fprintf(tmp,"\t%s",comon؛(

          fgets(line,80,fp); /* to skip emg. statemnet/*
    while(!feof(fp} ((
             fgets(line,80,fp؛(
             if(feof(fp)) break؛
             fprintf(tmp,"%s",line؛(
{
   fprintf(tmp,"\treturn\n؛("
   fprintf(tmp,"\tend؛("
   fclose(fp); fclose(tmp؛(
   tmp=fopen("tmp.c","r؛("
   fp=fopen(name,"w؛("
   while(!feof(tmp} ((
     fgets(line,80,tmp؛(
```

```
            printf("%s",line);/*getch/*؛()
            fprintf(fp,"%s",line؛(
        {
            fclose(fp); fclose(tmp؛(
            printf(" \n\n\n\a Insert into F-module  finish. Press any key ");getch؛()
            break؛
*/{   switch/*
{
/*********************************************************************/

int  checkname(char *name(
}
  printf("Now Check file name:< %s >   . Press any key \n",name؛(
  return 0؛
{
void ErrorMsg(int n(
}
     switch(n} (
          case 1: prin   ("Error: Input File format Error \n"); exit(0؛(
{
{

/**********************************************************************/
 */merg: combines the splitted C sections to OC program/*
/**********************************************************************/

void merg(int nmod} (
 int i؛
 FILE *fp,*tp؛
 char name[25], tmp[25؛[
```

```
 char line[80؛[
 strcpy(name,ModTab[0].fname؛(
 fp=fopen(name,"a؛("
 for(i=0;i<nmod;i} (++
   if(strcmp(ModTab[i].type,"CS")==0} (
    strcpy(tmp,ModTab[i].fname؛(
    tp=fopen(tmp,"r؛("
    fprintf(fp,"\n؛("
    while(!feof(tp} ((
            fgets(line,80,tp؛(
            if(!feof(tp}((
            fprintf(fp,"%s",line؛(
            puts(line{   ؛(
{
   fclose(tp؛(
{
{
 fclose(fp؛(
{



*/
Compile and link function


/*
void complink(int nom(
}


  int i,j; // nom= number of modules
  char mask[80],fn[30],ofn[30][10],lnk[80];     //fn= file name, ofn= object file names array
```

```
      //for(i=0;i<nom;i(++
     //   puts(ModTab[i].fname؛(

   puts("Now compiling؛(" :
    for(i=0;i<nom-1;i(++
}
     //fflush(stdiomask؛(
    if((ModTab[i].type[0])=='F('
}
==========//copy the FORTRAN files to f\york and compile them
      strcpy(mask,"copy؛("
      strcat(mask,ModTab[i].fname؛(
      strcat(mask," f\\york؛("
      system(mask؛(
      system("cd f؛("
     system("cd york؛("
      //puts(ModTab[i].fname؛(
      for(j=0;j<strlen(ModTab[i].fname);j(++
       if(ModTab[i].fname[j]!='.') fn[j]=ModTab[i].fname[j؛[
       else
}
       fn[j]='\0؛'
       break؛
{

       strcpy(mask,"f2lib؛("
       strcat(mask,fn؛(
             puts(fn؛(
      getch؛()
```

```
        system(mask؛(
{
    else
}
===//     copy the C files to minGW\bin and compile them
    system("cd؛("\\
      strcpy(mask,"copy؛("
      strcat(mask,ModTab[i].fname؛(
      strcat(mask," mingw\\bin؛("
      system(mask؛(
            //     puts(mask؛(
      getch؛()
*/              system("cd؛("\\
      system("cd \\mingw؛("
                system("cd \\bin/*؛("

      strcpy(mask,"c:\\mingw\\bin\\gcc -c؛("
      strcat(mask,ModTab[i].fname؛(
      system(mask؛(
              puts(ModTab[i].fname؛(
      getch؛()

{

{

==========//linke the  .o files========
   system("cd؛("\\

     for(j=0;j<strlen(ModTab[0].fname);j(++
```

```
        if(ModTab[0].fname[j]!='.') fn[j]=ModTab[0].fname[j؛[
        else
}
        fn[j]='\0؛'
        break؛
{
    strcpy(lnk,"g77 -ffree -form -o؛("
    strcat(lnk,fn؛(
    strcat(lnk,".exe؛("
    for(i=0;i<nom-1;i(++
}

===//      construct the object files (.o) names
        for(j=0;j<strlen(ModTab[i].fname);j(++
        if(ModTab[i].fname[j]!='.') fn[j]=ModTab[i].fname[j؛[
        else
}
        fn[j]=ModTab[i].fname[j؛[
        fn[++j]='o؛'
        fn[++j]='\0؛'
   //    puts(fn؛(
        break؛
{

            ========//  copy the object files to the mine directory
        strcpy(mask,"copy؛("
       strcat(mask,fn؛(
       strcat(mask," f\\mine؛("
//              puts(mask؛(
//      getch؛()
```

```
      system(mask؛(

      strcat(lnk," ..\\mine؛("\\
      //puts(fn؛(
      strcat(lnk,fn؛(
 //     puts(lnk؛(
{

   puts("Now linking the object files؛(":

      getch؛()
      system("cd؛("\\
              system("cd f؛("
     system("cd york؛("
     puts(lnk؛(
    system(lnk؛(
{

 ************************/E N   D/*********************************
```