

DATA STRUCTURES AND ALGORITHMS FOR GATE

-To All Hard Working GATE Aspirants

Copyright© by *CareerMonk.com*

All rights reserved.

Designed by *Narasimha Karumanchi*

Copyright© Career Monk Publications. All rights reserved.

All rights reserved. No part of this book may be reproduced in any form or by any electronic or mechanical means, including information storage and retrieval systems, without written permission from the publisher or author

Acknowledgements

I would like to express my gratitude to many people who saw me through this book, to all those who provided support, talked things over, read, wrote, offered comments, allowed me to quote their remarks and assisted in the editing, proofreading and design. In particular, I would like to thank the following individuals.

Prof. Girish P. Saraph [Founder, Vegayan Systems, IIT Bombay], for his encouragement during my stay at IIT Bombay.

Prof. Chintapalli Sobhan Babu [IIT, Hyderabad], *Prof. Meda Sreenivasa Rao* [JNTU, Hyderabad] for spending their valuable time in reviewing the book, suggestions and encouragement.

Kiran & Laxmi [Founders of *TheGATEMATE.com*], for approaching me for teaching *Data Structures and Algorithms* at their training centers.

My friends, colleagues and readers have contributed greatly to the quality of this book. I thank all of you for your help and suggestions.

Last but not least, I would like to thank *Directors of Guntur Vikas College, Prof. Y.V. Gopala Krishna Murthy & Prof. Ayub Khan* [ACE Engineering Academy], *T.R.C. Bose* [Ex. Director of APTransco], *Ch. Venkateswara Rao VNR Vignanjyothi* [Engineering College, Hyderabad], *Ch. Venkata Narasaiah* [IPS], *Yarapathineni Lakshmaiah* [Manchikallu, Gurazala] & all our well – wishers for helping me and my family during our studies.

-Narasimha Karumanchi
M-Tech, IIT Bombay
Founder of *CareerMonk.com*

Motivational and Inspirational Quotes

A stone is broken by the last stroke. This does not mean that first stroke was useless.

Success is a result of continuous daily effort. --*Unknown*

Stand up, be bold, be strong. Take the whole responsibility on your own shoulders,
and know that you are the creator of your own destiny. --*Swami Vivekananda*

Preface

Dear Reader,

Please Hold on! I know many people do not read preface. But I would like to strongly recommend reading preface of this book at least. This preface has *something different* from regular prefaces.

This book assumes you have some basic knowledge about computer science. Main objective of the book is not to give you the theorems and proofs about *Data Structures* and *Algorithms*. I have followed a pattern of improving the problem solutions with different complexities (for each problem, you observe multiple solutions with different improved complexities). Basically, it's an enumeration of possible solutions. With this approach, even if we get a new question it gives us a way to think about all possible solutions. For all other topics, it will act as a refreshment. This book is very much useful for interview preparation, competitive exams preparation, campus preparations.

This book is aimed for GATE students. We have tried to solve all problems related to *Data Structures* and *Algorithms* from the last twenty years papers. Each solution has explanation associated with it and this gives the confidence for readers about the correctness of the solutions. As a *job seeker* if you read complete book with good understanding, I am sure you will challenge the interviewers and that is the objective of this book.

This book is very much useful for the *students of Engineering Degree* and *Masters* during their academic preparations. All the chapters of this book contain theory and their related problems as many as possible. If you read as a *student* preparing for competition exams for Computer Science/Information Technology], the content of this book covers *all* the *required* topics in full details. While writing the book, an intense care has been taken to help students who are preparing for these kinds of exams.

In all the chapters you will see more importance given to *problems* and analyzing them instead of concentrating more on theory. For each chapter, first you will see the basic required theory and then followed by problems.

For many of the problems, *multiple* solutions are provided with different complexities. We start with *brute force* solution and slowly move towards the *best solution* possible for that problem. For each problem we will try to understand how much time the algorithm is taking and how much memory the algorithm is taking.

It is *recommended* that, at least one complete reading of this book is required to get full understanding of all the topics. In the subsequent readings, you can directly go to any chapter and refer. Even though, enough readings were given for correcting the errors, due to human tendency there could be some minor typos in the book. If any such typos found, they will be updated at www.CareerMonk.com. I request you to constantly monitor this site for any corrections, new problems and solutions. Also, please provide your valuable suggestions at: Info@CareerMonk.com.

Wish you all the best. Have a nice reading.

-Narasimha Karumanchi
M-Tech, IIT Bombay
Founder of CareerMonk.com

Foreword

With the availability of different books on Data Structures and/or Algorithms, their design and analysis with a general or language based orientation; one would be in gross error to think that this book is one more to the list. Keeping in view the GATE and other competitive exams perspective in mind and critical time management requirement, the author has taken the approach of embedding theoretical concepts through efficient problem solving.

The different topics related to data structures and algorithms and their intricate interdependency have been dealt with comprehensive eloquence and subtle clarity.

The topics coverage in the book spans from primitive to advanced data structures, their characteristics, their space and time complexity analysis, the different algorithmic paradigms/classifications, application areas and problems and their analysis.

Each topic is explained through a problem solving approach so that at the end, the reader is well equipped to be able to apply to any type of problem solving requirement and distinctly choose one strategy or type from the other.

The book also includes solved GATE questions of last twenty years which works as a mock exam practice plane for the reader. It is a very handy and must have book for any reader aiming for competitive exams of all universities and boards and as well as for campus recruitment. It can also serve as a reference text-book for software developers, who wanted to design and implement efficient algorithms.

*Dr. Khaleel Ur Rahman Khan,
Prof. of C.S.E and Dean
ACE Engineering College
Hyderabad*

Table of Contents

1. Programming Basics -----	9
2. Introduction-----	30
3. Recursion and Backtracking-----	52
4. Linked Lists -----	57
5. Stacks -----	93
6. Queues-----	113
7. Trees-----	123
8. Priority Queue and Heaps -----	193
9. Disjoint Sets ADT -----	211
10. Graph Algorithms-----	220
11. Sorting-----	259
12. Searching -----	280
13. Selection Algorithms-----	304
14. Symbol Tables-----	311
15. Hashing-----	313
16. String Algorithms -----	329
17. Algorithms Design Techniques-----	350
18. Greedy Algorithms-----	353
19. Divide and Conquer Algorithms -----	363
20. Dynamic Programming-----	377
21. Complexity Classes-----	415
22. Miscellaneous Concepts -----	422

Other Titles by *Narasimha Karumanchi*

Success keys for Big Job Hunters

- 🔪 Data Structures and Algorithms Made Easy (C/C++)
- 🔪 Data Structures and Algorithms Made Easy in Java
- 🔪 Coding Interview Questions
- 🔪 Peeling Design Patterns



PROGRAMMING BASICS



Chapter-1

The objective of this chapter is to explain the importance of analysis of algorithms, their notations, relationships and solving as many problems as possible. We first concentrate on understanding the basic elements of algorithms, importance of analysis and then slowly move towards analyzing the algorithms with different notations and finally the problems. After completion of this chapter you should be able to find the complexity of any given algorithm (especially recursive functions).

1.1 Variables

Before going to the definition of variables, let us relate them to old mathematical equations. All of us have solved many mathematical equations since childhood. As an example, consider the below equation:

$$x^2 + 2y - 2 = 1$$

We don't have to worry about the use of above equation. The important thing that we need to understand is, the equation has some names (x and y) which hold values (data). That means, the *names* (x and y) are the place holders for representing data. Similarly, in computer science we need something for holding data and *variables* are the facility for doing that.

1.2 Data types

In the above equation, the variables x and y can take any values like integral numbers (10, 20 etc...), real numbers (0.23, 5.5 etc...) or just 0 and 1. To solve the equation, we need to relate them to kind of values they can take and *data type* is the name being used in computer science for this purpose.

A *data type* in a programming language is a set of data with values having predefined characteristics. Examples of data types are: integer, floating point number, character, string etc...

Computer memory is all filled with zeros and ones. If we have a problem and wanted to code it, it's very difficult to provide the solution in terms of zeros and ones. To help users, programming languages and compilers are providing the facility of data types.

For example, *integer* takes 2 bytes (actual value depends on compiler), *float* takes 4 bytes etc... This says that, in memory we are combining 2 bytes (16 bits) and calling it as *integer*. Similarly, combining 4 bytes (32 bits) and calling it as *float*. A data type reduces the coding effort. Basically, at the top level, there are two types of data types:

- System defined data types (also called *Primitive* data types)
- User defined data types

System defined data types (Primitive data types)

Data types which are defined by system are called *primitive* data types. The primitive data types which are provided by many programming languages are: int, float, char, double, bool, etc... The number of bits allocated for each primitive data type depends on the programming languages, compiler and operating system. For the same primitive data type, different languages may use different sizes. Depending on the size of the data types the total available values (domain) will also changes. For example, "*int*" may take 2 bytes or 4 bytes. If it takes 2 bytes (16 bits) then the total

possible values are $-32,768$ to $+32,767$ (-2^{15} to $2^{15}-1$). If it takes, 4 bytes (32 bits), then the possible values are between $-2,147,483,648$ to $+2,147,483,648$ (-2^{31} to $2^{31}-1$). Same is the case with remaining data types too.

User defined data types

If the system defined data types are not enough then most programming languages allows the users to define their own data types called as user defined data types. Good example of user defined data types are: structures in *C/C++* and classes in *Java*.

For example, in the below case, we are combining many system defined data types and called it as user defined data type with name “*newType*”. This gives more flexibility and comfort in dealing with computer memory.

```
struct newType {  
    int data1;  
    float data 2;  
    ...  
    char data;  
};
```

1.3 Data Structure

Based on the above discussion, once we have data in variables, we need some mechanism for manipulating that data to solve problems. *Data structure* is a particular way of storing and organizing data in a computer so that it can be used efficiently. That means, a *data structure* is a specialized format for organizing and storing data. General data structure types include arrays, files, linked lists, stacks, queues, trees, graphs and so on. Depending on the organization of the elements, data structures are classified into two types:

- 1) *Linear data structures*: Elements are accessed in a sequential order but it is not compulsory to store all elements sequentially (say, Linked Lists). *Examples*: Linked Lists, Stacks and Queues.
- 2) *Non – linear data structures*: Elements of this data structure are stored/accessed in a non-linear order. *Examples*: Trees and graphs.

1.4 Abstract Data Types (ADTs)

Before defining abstract data types, let us consider the different view of system defined data types. We all know that, by default, all primitive data types (int, float, etc..) supports basic operations like addition, subtraction etc... The system is providing the implementations for the primitive data types. For user defined data types also we need to define operations. The implementation for these operations can be done when we want to actually use them. That means, in general user defined data types are defined along with their operations.

To simplify the process of solving the problems, we generally combine the data structures along with their operations and are called *Abstract Data Types* (ADTs). An ADT consists of *two* parts:

1. Declaration of data
2. Declaration of operations

Commonly used ADTs *include*: Linked Lists, Stacks, Queues, Priority Queues, Binary Trees, Dictionaries, Disjoint Sets (Union and Find), Hash Tables, Graphs, and many other. For example, stack uses LIFO (Last-In-First-Out) mechanism while storing the data in data structures. The last element inserted into the stack is the first element that gets deleted. Common operations of it are: creating the stack, pushing an element onto the stack, popping an element from stack, finding the current top of the stack, finding number of elements in the stack etc...

While defining the ADTs do not care about implementation details. They come in to picture only when we want to use them. Different kinds of ADTs are suited to different kinds of applications, and some are highly specialized to specific tasks. By the end of this book, we will go through many of them and you will be in a position to relate the data structures to the kind of problems they solve.


1.5 Memory and Variables

Address	Memory Value
0	
1	
2	
...	...
...	
...	
$2^n - 1$	

First let's understand the way memory is organized in a computer. We can treat the memory as an array of bytes. Each location is identified by an address (index to array). In general, the address 0 is not a valid memory location. It is important to understand that the address of any byte (location) in memory is an integer. In the below diagram n value depends on the main memory size of the system.

To read or write any location, CPU accesses it by sending its address to the memory controller. When we create a variable (for example, in C: `int X`), the compiler allocates a block of contiguous memory locations and its size depends on the size of the variable. The compiler also keeps an internal tag that associates the variable name X with the address of the first byte allocated to it (sometimes called as *symbol table*). So when we want to access that variable like this: $X = 10$, the compiler knows where that variable is located and it writes the value 10.

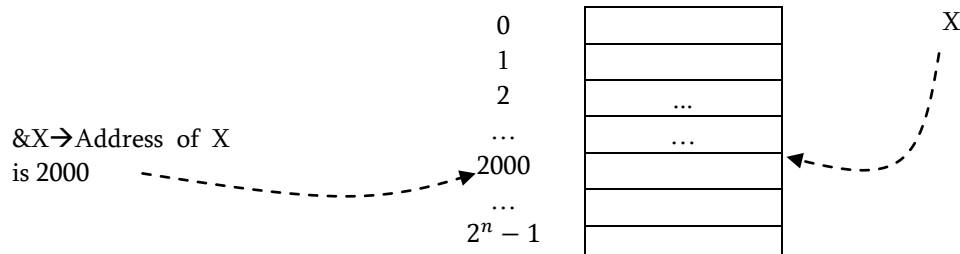
Address	Memory Value
0	
1	
2	
...	...
2000	
...	
$2^n - 1$	



Size of a Variable: *sizeof* operator is used to find size of the variable (how much memory a variable occupies). For example, on some computers, *sizeof*(X) gives the value 4. This means an integer needs 4 contiguous bytes in memory. If the address of X would be 2000, then the actual memory locations used by X are: 2001, 2002, 2003, and 2004.

Address of a Variable: In C language, we can get the address of a variable using *address-of* operator (&). The below code prints the address of X variable. In general, addresses are printed in hexadecimal as they are compact and also easy to understand if the addresses are big.

Address	Memory Value
0	
1	
2	
...	...
...	...
2000	
...	
$2^n - 1$	



```

int X;
printf("The address is: %u\n", &X);

```

1.6 Pointers

Pointers are also variables which can hold the address of another variable.

Declaration of Pointers: To declare a pointer, we have to specify the type of the variable it will point to. That means we need to specify the type of the variable whose address it's going to hold. The declaration is very simple. Let us see some examples of pointer declarations below:

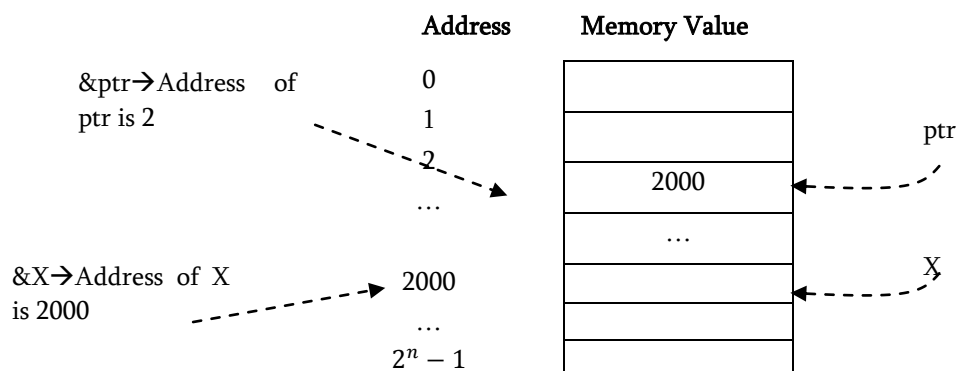
```
int *ptr1;
float *ptr2;
unsigned int *ptr3;
char *ptr4; void *ptr5;
```

Here *ptr1* is a pointer that can point to an *int* variable, *ptr2* can point to a *float*, *ptr3* to an *unsigned int*, and *ptr4* to a *char*. Finally *ptr5* is a pointer that can point to anything. These pointers are called *void* pointers, and there are some restrictions on what we can do with void pointers.

Pointers Usage: As we said, pointers hold addresses. That means, we can assign the address of a variable to it. Let us consider the below sample code:

```
int X = 10;
int *ptr = &X;
```

Here, we first declare an integer named *X* and initialize it to the value 10. Then we create a pointer to *int* named *ptr* and assign the address of *X* to it. This is called, “making the pointer *ptr* point to *X*.” A common operation we can do with a pointer is what is called *indirection* and it is a way to access the contents of the memory that it points to. The indirection operator is represented by the asterisk symbol. Do not confuse this operator with the use of the same asterisk symbol in the declaration of pointers, they are not the same thing.



If we would like to access the contents of the memory where *ptr* points to, we would do it like this: **ptr*. Let's see a small code that shows pointer indirections.

```
int X = 10;
int *ptr = &X;
printf("X contains the value %d\n", X);
printf("ptr points to %p\n", ptr);
printf("there lies the value %d\n", *ptr);
*ptr = 25;
printf("now X contains the value %d\n", X);
```

Here we first declare *X* and *ptr* just like before. Then we print *X* (which is 10), followed by *ptr*, i.e., the contents of the variable *ptr* which is an address; the address of *X*. And finally we print **ptr*, which is the value of the memory location where *ptr* points to (again 10, since it points to the location occupied by the variable *X* in memory).

Finally we change the contents of the location where *ptr* points to by writing **ptr = 25*, this means assign the value 25 to wherever *ptr* is pointing to. Note that when we do that, what actually happening is modifying the value of *X*.

This is because, *ptr* holds the address of *X*, and changing the contents of the memory at that address changes the value of *X*.

One limitation of *void* pointers, i.e. pointers that can point to any type, is that they cannot be *dereferenced*. This is because of the fact that each variable type takes different amount of memory. On a 32-bit computer for example usually an int needs 4 bytes, while a short 2 bytes. So in order to read the actual value stored there, the compiler has to know how many consecutive memory locations to read in order to get the full value.

Pointer Manipulation: Another very useful thing we can do with pointers is to perform arithmetic operations on them. This might be obvious to the careful reader, since we said that pointers are just integers. However, there are a few small differences on pointer arithmetic that make their use even more intuitive and easy. Try the following code:

```
char *cptr = (char*)2;
printf("cptr before: %p ", cptr);
cptr++;
printf("and after: %p\n", cptr);
```

We declare a pointer named *cptr* and assign the address 2 to it. We print the contents of the pointer (i.e. the address 2), increment it, and print again. Sure enough the first time it prints 2 and then 3, and that was exactly what we expected. However try this one as well:

```
int *iptr = (int*)2;
printf("iptr before: %p ", iptr);
iptr++;
printf("and after: %p\n", iptr);
```

Now the output, on my computer, is *iptr* before: 2 and after: 6! Why does this pointer point to the address 6 after incremented it by one and not to 3 as the previous pointer? The answer lies with what we said about the *size of* variables. An int is 4 bytes on my computer. This means that if we have an int at the address 2, then that int occupies the memory locations 2,3,4 and 5. So in order to access to the next int we have to look at the address 6,7,8 and 9. Thus when we add one to a pointer, it is not the same as adding one to any integer, it means give me a pointer to the next variable which for variables of type int, in this case, is 4 bytes ahead.

The reason that in the first example with the char pointer, the actual address after incrementing, was one more than the previous address is because the size of char is exactly 1. So the next char can indeed be found on the next address. Another limitation of void pointers is that we cannot perform arithmetic on them, since the compiler cannot know how many bytes ahead is the next variable located. So void pointers can only be used to keep addresses that we have to convert later on to a specific pointer type, before using them.

Arrays and Pointers: There is a strong connection between arrays and pointers. So strong in fact, that most of the time we can treat them as the same thing. The name of an array can be considered just as a pointer to the beginning of a memory block as big as the array. So for example, making a pointer point to the beginning of an array is done in exactly the same way as assigning the contents of a pointer to another:

```
short *ptr;
short array[10];
ptr = array;
```

and then we can access the contents of the array through the pointer as if the pointer itself was that array. For example this: *ptr*[2] = 25 is perfectly legal. Furthermore, we can treat the array itself as a pointer, for example, **array* = 4 is equivalent to *array*[0] = 4. In general **(array + n)* is equivalent to *array*[*n*].

The only difference between an array, and a pointer to the beginning of an array, is that the compiler keeps some extra information for the arrays, to keep track of their storage requirements. For example if we get the size of both an array and a pointer using the *sizeof* operator, *sizeof(ptr)* will give us how much space does the pointer itself occupies (4 on my computer), while *sizeof array* will give us, the amount of space occupied by the whole array (on my computer 20, 10 elements of 2 bytes each).

Dynamic Memory Allocation: In the earlier sections, we have seen that pointers can hold addressed of another variables. There is another use of pointers: pointers can hold addresses of memory locations that do not have a specific compile-time variable name, but are allocated dynamically while the program runs (sometimes such memory is called *heap*).

To allocate memory during runtime, *C* language provides us the facility interns of *malloc* function. This function allocates the requested amount of memory, and returns a pointer to that memory. To deallocate that block of memory, *C* supports it by providing *free* function. This function takes the pointer as an argument. For example, in the below code, an array of 5 integers is allocated dynamically, and then deleted.

```
int count = 5;
int *A = malloc(count * sizeof(int));
.....
free(arr);
```

In this example, the *count * sizeof(int)* calculates the amount of bytes we need to allocate for the array, by multiplying the number of elements, to the size of each element (i.e. the size of one integer).

Function Pointers: Like data, executable code (including functions) also stored in memory. We can get the address of a function. But the question is what type of pointers we use for that purpose?

In general, we use function pointers and they store the address of functions. Using function pointers we can call the function indirectly. But, function pointers manipulation has limitation: limited to assignment and indirection and cannot do arithmetic on function pointers. Because for function pointers there no ordering (functions can be stored anywhere in memory). The following example illustrates how to create and use function pointers.

```
int (*fptr)(int);
fptr = function1;
printf("function1 of 0 is: %d\n", fptr(5));
fptr = function2;
printf("function2 of 0 is: %d\n", fptr(10));
```

First we create a pointer that can point to functions accepting an *int* as an argument and returning *int*, named *fptr*. Then we make *fptr* point to the *function1*, and proceed to call it through the *fptr* pointer, to print the *function1* of 5. Finally, we change *fptr* to point to *function2*, and call it again in exactly the same manner, to print the *function2* of 10.

1.7 Parameter Passing Techniques

Before starting our discussion on parameter passing techniques, let us concentrate on the terminology we use.

Actual and Formal Parameters: Let us assume that a function *B()* is called from another function *A()*. In this case, *A* is called the “caller function” and *B* is called the “called function or callee function”. Also, the arguments which *A* sends to *B* are called actual arguments and the parameters of *B* function are called formal arguments. In the below example, the *func* is called from *main* function. *main* is the caller function and *func* is the called function. Also, the *func* arguments *param1* and *param2* are formal arguments and *i, j* of *main* function are actual arguments.

```
int main() {
    long i = 1;
    double j = 2;
    // Call func with actual arguments i and j.
    func( i, j );
}
// func with formal parameters param1 and param2.
void func( long param1, double param2 ){
}
```

Semantics of Parameter Passing: Logically, parameter passing uses the following semantics:

- **IN:** Passes info from caller to *callee*. Formal arguments can take values from actual arguments, but cannot send values to actual arguments.
- **OUT:** Callee writes values in the caller. Formal arguments can send values from actual arguments, but cannot take values from actual arguments.
- **IN/OUT:** Caller tells callee value of variable, which may be updated by callee. Formal arguments can send values from actual arguments, and also can take values from actual arguments.

Language Support for Parameter Passing Techniques

Passing Technique	Supported by Languages
Pass by value	<i>C, Pascal, Ada, Scheme, Algol68</i>
Pass by result	<i>Ada</i>
Pass by value-result	<i>Fortran</i> , sometimes <i>Ada</i>
Pass by reference	<i>C</i> (achieves through pointers), <i>Fortran</i> , <i>Pascal var params</i> , <i>Cobol</i>
Pass by name	<i>Algol60</i>

Pass by Value: This method uses in-mode semantics. A formal parameter is like a new local variable that exists within the scope of the procedure/function/subprogram. Value of actual parameter is used to initialize the formal parameter. Changes made to formal parameter *do not* get transmitted back to the caller. If pass by value is used then the formal parameters will be allocated on stack like a normal local variable. This method is sometimes called as *call by value*. Advantage of this method is that, it allows the actual arguments not to modify.

In general the pass by value technique is implemented by copy and it has the following disadvantages:

- Inefficiency in storage allocation
- Inefficiency in copying value
- For objects and arrays, the copy semantics are costly

Example: In the following example, main passes func two values: 5 and 7. The function func receives copies of these values and accesses them by the identifiers a and b. The function func changes the value of a. When control passes back to main, the actual values of x and y are not changed.

```
void func (int a, int b) {
    a += b;
    printf("In func, a = %d   b = %d\n", a, b);
}
int main(void) {
    int x = 5, y = 7;
    func(x, y);
    printf("In main, x = %d   y = %d\n", x, y);
    return 0;
}
```

The output of the program is: In func, a = 12 b = 7. In main, x = 5 y = 7

Pass by Result: This method uses out-mode semantics. A formal parameter is a new local variable that exists within the scope of the function. No value is transmitted from actual arguments to formal arguments. Just before control is transferred back to the caller, the value of the formal parameter is transmitted back to the actual parameter. This method is sometimes called as *call by result*.

Actual parameter *must* be a variable. *foo(x)* and *foo(a[1])* are fine but not *foo(3)* or *foo(x * y)*.

Parameter collisions can occur: That means, suppose let us assume that there is a function *write(p1, p1)*. If the two formal parameters in write had different names, which value should go into p1? Order in which actual parameters are copied determines their value. In general the pass by result technique is implemented by copy and it has the following disadvantages: