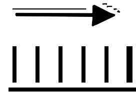
# Introduction to data structures



a[O] a[l] a[2] a[3] a[4]
3
memory representation
1.2
Figure 1.2
Storage Representation Of Data

THE NEED FOR DATA STRUCTURES

One of the tools that beginers often take for granted is the high-level language in which they write their programs. Since most of us first learn to program in a language like C. we do not appreciate its branching and looping structures and built-in data structures until we are later introduced to language that do not have these features.

In the first semester(C programming). we decided to use an array of structure to store our data. But what is an array? What is a structure? C. as well as many other hig-level programming langauges. provides arrays and structure as built-in data structures. As a C programming. you can use these tools without concern about their implementation. much as a car driver can use a car without knowing about automobile technology.

However, there are many interesting and useful ways of structuring data that are not provided in general-purpose programming languages. The programmer who wants to use these structures must build them. In this book, we will look in detail four useful data structures: stacks, queues, lists and binary trees. We will decribe each of these structures and design algorithms to manipulate them. We will build them using the tools that are available in the C language. Finally, we will examine applications where each is appropriate.

First,however, we will develop a definition of data structure and an approach that we can use to examine data structures. By way of example, we will apply our definition and approach to familiar C data structures: the one dimensional array, the two dimensional array, and the structure.

**Definition**

**Data Structure :** A collection of data elements whose organization is characterised by accessing functions that are used to store and retrieve individual data elements.

**OR**

The logical or mathematical model of a particular organization of data is called **data structure**.

The two important goals of data structures are first to identify the representation of abstract entities and then to identify the operations. which can be performed with them. The operations help us to determine the class of problems, which can be solved with these entities.



The choice of data model depends on two considerations. First. it must be rich enough in structure to show the actual relationships of data in real world. On the other hand, the structure should be simple enough so that one can efficiently process the data when necessary. Data structure is nothing but arrangement of data and their relationship and the allowed operations. One can use simple data structure to build complex data structures.

Data structures are fundamental to computer programming in any language. As progranuners work on algorithm development and problem analysis. they make crucial decisions about data structures. A data structure is a representation of the data in the program. The proper construction of a program is influenced by the choice of data structure. which is used. A data structure is a systematic way of organizing and accessing data, and an algorithm is a step- by-step procedure for perfonIling some task ina tillite amount of time. These concepts are central in computing.

## 1.3 GOALS OF DATA STRUCTURES

The goals of data structures can be designed to answer certain questions such as

1. Does the data structure do what it is supposed to do?
,,2. Does the representation work according to the requirement specification , of the task?
~ 3. Is there a proper description of the representation des;cribing how to use .it and how it works?

The above questions when answered create the- fundamental goals that are used in designing descriptions of data structures. some of them are

1. Correctness
i 2. Efficiency .,
., 3. Robustness ,; 4. Adaptability
5. Reusability

**By correctness.** we mean that a data structure is designed. to work correctly .for all possible inputs that one might encounter. For example. a data structure that is supposed to store a collection of numbers in order should never allow for elements i to be stored out of order. The precise meaning of correctness. will always depend I on the specific problem the data structure is intended to solve. but correctness should be a primary goal.

Useful data structure and their operations also need to be **efficient.** That is, they should be fast and not use more of the computer's resources. such as memory space. than required. In a real-time situation, the speed of a data structure ~ operation can make the difference between success and failure, a difference that can often be quite important.

Every good programmer wants to produce software that is **robust**, which means that a program produces the correct output for all inputs. For example, if a program is expecting a number to be input as an integer and instead it is input as a floating-point number, then the program should be able to recover from this error. A program that does not handle such unexpected-input errors can be embarrassing for the programmer.

Modern software projects, such as those for developing word processors, Web browsers, and Internet search engines, involve large software systems that are expected to last for many years. Software, therefore, needs to be able to evolve over time in response to changing conditions. These changes can be expected, such as the need to adapt to an increase in CPU speed. Software should also be able to adapt to unexpected events. Thus, another important goal of quality software is that it be **adaptable**.

Going hand-in-hand with adaptability is the desire that software be **reusable** that is the same code be a component of different systems in various application situations. Developing quality software can be expensive, and its cost can be reduced somewhat if the software is designed in a way that makes it easily reusable in future applications. Software reuse can be a significant cost-saving and timesaving technique.

## 1.4 NEED FOR ABSTRACTION

An abstraction is a powerful concept in computer science. The main idea of this concept is to distill a complicated system down to its most fundamental parts and describe these parts in a simple, precise language. Typically, describing the parts of a system involves naming the different parts and describing their functionality.

For example, a typical text-editor graphical user interface (GUI) provides an abstraction of an editor program that offers several specialized text-editing operations, including cutting and pasting portions of text or placing graphical objects at different locations in the text. With abstraction it is not necessary to go into the details about the various complicated ways in which a GUI represents and displays text and graphical objects.

The abstract functionality of an edit program and its cutting and pasting operations is specified in a language precise enough to be clear, but simple enough to "abstract away" unnecessary details. This combination of clarity and simplicity benefits robustness, since it leads to understandable and correct implementations.

## 1.5 CLASSIFICATION OF DATA STRUCTURES

The data structures are classified in the following categories:

1. Primitive data structures
2. Non-primitive data structures.

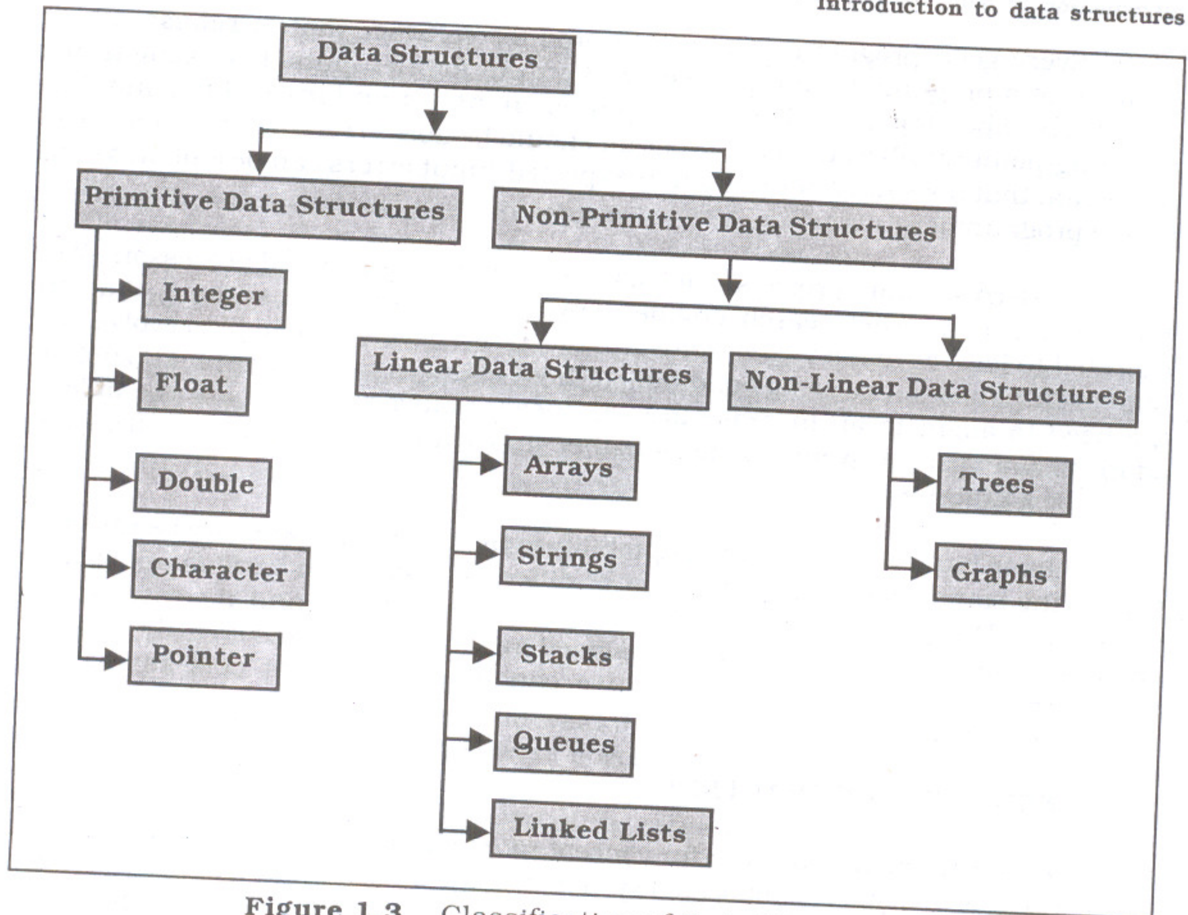



**Figure 1.3** Classification of Data Structures

### 1.5.1 Primitive Data structures

**Primitive data structures** are those structures, which are readily available in a programming language i.e., they can be directly operated upon by programming instructions. Here we are concerned with structuring of data at their most primitive level within a computer. The storage representation or structure (Memory representation) and the possible operations (arithmetic operations, relational operations) for these types of structures are predefined and the user cannot change this. The storage structure of these data structures may vary from one machine to another. The different primitive data structures are *integer, float, double, character* and *pointer.*

> **Definition**
>
> **Primitive data structures :** The data structures that are directly operated upon by machine-level instructions.

#### 1.5.1.1 Integers

We shall now look into the storing and using of integers. Integers as we know are whole numbers or natural numbers, which can be either positive or

**Example 1.1** : Consider the following program segment in C.

```
int    *ptr;
int    Info;

Info=*ptr;
```

Where **ptr** contains the address 0xff02 and the information stored at this location is 11, after this assignment we will get 11 in **Info**.

We discuss pointers and its operations in detail in the next chapter.

### 1.5.2 Operations on Primitive Data Structures

Some of the common operations on Primitive Data Structure are:

i. **Creation Operation :** This operation is used to create a storage representation for a particular data structure. This operation is normally performed with the help of a declaration statement available in the programming language.

**Example 1.2** : int n = 45;

causes memory space to be created for **n**.

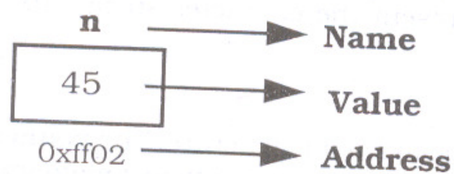



**Figure 1.5** Memory space refered by, name **n** where integer value can be stored.

ii. **Selection Operation :** This is most frequently used operation and is used to access data within a data structure. For complex structures method of access is one of the important property of a structure. In case of files the access can be sequential or random depending on the nature of files. This operation is normally performed using the name of the structure.

**Example 1.3 :** scanf ( "%d", &a);

iii. **Update Operation :** This operation is used to change or modify the data in a structure. An assignment operation is a good example of an update operation.

**Example 1.4 :** y = 5;

modifies the value of **y** to store the new value 5 in it.

iv. **Destroy Operation :** This operation is used to destroy or disassociate a particular data structure from its storage representation. In some languages this operation is not supported or it is automatically performed. In C one can destroy data structure by using the function called **free( )**. This aids in efficient use of memory.

### 1.5.3 Non-Primitive Data Structures

**Non-Primitive data structures** are those structures, which are not readily available in a programming language i.e., they cannot be directly operated upon by programming instructions. The storage representation and the possible operations for these types of structures are not predefined and the user has to define them. The different non-primitive data structures are **arrays**, **stacks**, **queues**, **files** and **linked lists**.

**Definition**

**Non-primitive data structures :** The data structures that are composed of primitive data structures.

Non-primitive data structures are further classified into two types.

1. Linear data structure and
2. Non-linear data structure

#### 1.5.3.1 Linear Data Structures

A linear data structures exhibit an important property called as adjacency between the elements. The concept of adjacency may indicate either a linear or sequential relationship i.e., if we are able to identify the position of an element we should be able to identify the position of the previous element and the next element. The different linear data structures are **arrays**, **strings**, **stacks**, **queues** and **linked list**. We discuss these data structures from chapter 3 to chapter 7.

**Definition**

**Linear data structure :** A data structure is said to be linear if there is a adjancency relationship between the elements.

#### 1.5.3.2 Non-Linear Data Structures

A non-linear data structures can exhibit any property other than adjacency between the elements. Non-linear data structure may exhibit either a hierarchical relationship or a parent child relationship. The different non-linear data structures are **trees** and **graphs**.

**Definition**

**Non-Linear data structure :** A data structure in which insertion and deletion is not possible in a linear fashion.

### 1.5.3.3 OPERATIONS ON NON-PRIMITIVE DATA STRUCTURES

Some of the common operations on Non Primitive Data Structure are:

i. **Traversing:** It is the process of visiting each element in the data structure exactly once to perform certain operations on it.

ii. **Sorting :** It is the process of arranging the elements of a particular data structure in some logical order. The order may be either ascending or descending or alphabetic order depending on the data item present.

iii. **Merging :** It is the process of combining the elements in two different structures into a single structure.

iv. **Searching :** It is the process of finding the location of the element with a given key value in a particular data structure or finding the location of an element, which satisfies the given condition.

v. **Insertion :** It is the process of adding a new element to the structure. Most of the times this operation is performed by identifying the position where the new element is to be inserted.

vi. **Deletion :** It is the process of removing an item from the structure.

## 1.6 ALGORITHM

The field of Computer Science can be defined as the study of algorithms because the main use of computers is to solve problems for us. A good algorithm is like a sharp knife, which does exactly what it is suppose to do with a minimum applied effort.

**Definition**

**ALGORITHM :** A set of ordered steps or procedures necessary to solve a problem.

Every problem as we understand can be solved using different methods. Thus each method may be represented using an algorithm. The important question to answer is How to choose the best algorithm? The design of a solution requires two goals to be looked at, most of the times they are conflicting. They are

1. Design an algorithm that is easy to understand, code and debug.
2. Design an algorithm that makes use of computer resources efficiently.

The first goal is concerned with a study in computer science called as Software Engineering and the second is concerned with the choice of data structures and the analysis of algorithms. Let us restrict our study to the second goal.

The main aim of using a computer is to transform data from one form to another. The algorithm describes the process of transforming data. That is why we often hear that a computer is referred to as a "Data Processing Machine". Raw data is the input to a computer and the algorithm shows the method used to transform the raw data into refined data or information. The organization of data thus plays an important role in the efficiency of algorithms since any organized data can be easily accessed and manipulated.

We have to organize the data either as a logical model or as a mathematical model as described by the definition of data structures.

The chosen model should reflect all the relationships and properties that exist between data and they can be accessed and implemented easily. While choosing a model we should also see to that as to what kind of operations can be performed on the data. Therefore Data Structures can also be defined as arrangement of data and their relationships.

Efficiency of algorithms depends upon the data structures that are selected for data representation. The data structure has to be finally represented in the memory. This is called as memory representation of data structures. While selecting the memory representation of data structures it should use less memory space and it should also be easy to access. Data Structures is also specification of how to represent information or data.

## 1.7 COMPLEXITY OF ALGORITHMS

Ever algorithm we write should be analyzed before it is implemented as a program. The process of analyzing algorithms forms a major field of study in computer science. There are two main criteria's or reasons upon which we can judge an algorithm. They are

1. The correctness of the algorithm and
2. The simplicity of the algorithm

The correctness of an algorithm can be analyzed by tracing the algorithm with certain sample data and by trying to answer certain questions such as.

1. Does the algorithm do what we want it to do?
2. Does the algorithm work when the data structure used is empty?
3. Does the algorithm work when the data structure used is full?
4. Does the algorithm work for all possibilities that can occur between a full structure and an empty structure?

The simplicity of an algorithm can be analyzed by trying to understand whether the algorithm can be implemented in a better and much simpler way. In order to analyze this we will have to consider the time requirements and the space requirements of the algorithm. These are the two parameters on which the efficiency

of the algorithms is measured. Space requirements are not a major problem today because memory is very cheap. So time is the only criteria for measuring efficiency of the algorithm. Consider the examples given below:

The time complexity of an algorithm is not measured by finding out how much time a particular algorithm requires for performing its task because the speed of different computers may be different (a slower computer may take more time whereas a faster computer may take less time for the same algorithm). The time is measured by counting the number of key operations, which are performed. Normally input and output operations are not considered as key operations. Comparison or assignment operations are considered as key operations. That is because key operations are so defined that the time for the other operations is much less than or at most proportional to the time for the key operation.

| Statement | Complexity |
| --- | --- |
| x=x+1; | We assume that the statement x=x+1 is not contained within any loop either explicitly or implicitly. Then, its frequency count is 1. |
| for (i=1; i<=n; i++)<br>x = x+1; | Now the same statement will executed **n** times. |
| for (i=1; i<=n; i++)<br>for (j=1; j<=n; j++)<br>x = x+1; | This will be executed $\mathbf{n^2}$ times. |

The complexity of algorithm M is the function of **f (n)**, which gives running time of the algorithm in terms of the number of key operations performed.

Let us take an example to understand how the analysis of time complexity helps us to decide the amount of work done by the algorithm based on which it is possible to select the best algorithm.

**Example 1.5** : Consider the task of finding the largest of three numbers. This problem can be solved using many methods, let us look at some of the methods and then analyze them.

**Method 1 :**

```
Step 1:    L = num1
Step 2:    If ( num2 > L ) L = num2
Step 3:    If ( num3 > L ) L = num3
```

**Method 2 :**

```
Step 1:    If ( num1 > num2 )
                   If ( num1 > num3 )  L = num1
                         else  L = num3
           else
                   If ( num2 > num3) L = num2
                         else  L = num3
```

**Method 3 :** Step 1: If ( num1 > num2 ) and ( num1 > num3 ) L = num1
Step 2: If ( num2 > num1 ) and ( num2 > num3 ) L = num2
Step 3: If ( num3 > num1 ) and ( num3 > num2 ) L = num3

Let us now look at the relative efficiency of the three methods. The first method requires us to perform two comparisons and three assignment operations. The second method requires three comparisons and four assignment operations. The last method though relatively simple requires six comparisons and three assignment operations.

From the above methods if we take the comparison operation as the key operation. Generalizing say to find the largest of "n" numbers we can say that the first method requires ( n – 1 ) comparisons. The second method may also require about ( n – 1 ) comparisons but it looks difficult to analyze. The third method would require each number to be compared with each other number thus it would require about n* ( n – 1 ) comparisons i.e., it would have to perform n times more work.

As a result of this analysis we see that method1 is the best method that is efficient and easy to generalize.

**Example 1.6 :** Let us now consider the example of Bubble Sort algorithm and analyze it.

Let A be an array of size N. This algorithm sorts the array in the ascending order.

```
Step 1:   For I =1 to N-1 do
Step 2:   For J =1 to N-I do
Step 3:   If a[J]  > a[J+1] then
Step 4:            Interchange a[J] and a[J+1]
          [End If ]
          [END For ( end of inner loop )]
          [END For ( end of outer loop )]
Step 5:   END
```

In this algorithm the key operation is the comparison operation. The number of comparisons can be easily computed. The first pass of the algorithm results in n-1 comparisons and in the worst case may result in n-1 interchanges also. The second pass results in n-2 comparisons and in the worst case may result in n-2 interchanges. Continuing the analysis we observe that as the iterations or passes increases the comparisons and exchanges decreases. Finally the total number of comparisons will be equal to

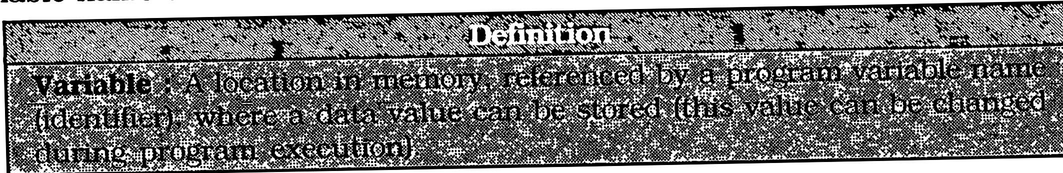
$$= (n-1) + (n-2) + (n-3) + \ldots\ldots\ldots\ldots + 2 + 1$$

$$= (n) * ( n-1) / 2$$

$$= O(n^2)$$

## 2.1 INTRODUCTION

A program operates on data. Data is stored in memory. While a program is executing, different values may be stored in the same memory location at different times. This kind of memory location is called a variable, and its content is the variable value. The symbolic name that we associate with a memory location is the variable name or variable identifier.



**Definition**

**Variable :** A location in memory, referenced by a program variable name (identifier), where a data value can be stored (this value can be changed during program execution).

Any variable, which is declared and initialized, has three things associated with it

1. A memory location with to hold the value of the variable,
2. The initialized value, which is stored in the location and
3. The address of that memory location.

All the three things are equally important. The name of a variable, which represents the memory location, is used to output the value stored in the variable and the address of the variable is used to input a value to the variable. Consider the following declaration in C.

```
int    a;
```

In this declaration, **a** is the name of the variable that is declared to be of type **int**. An integer variable a occupies two bytes of memory.

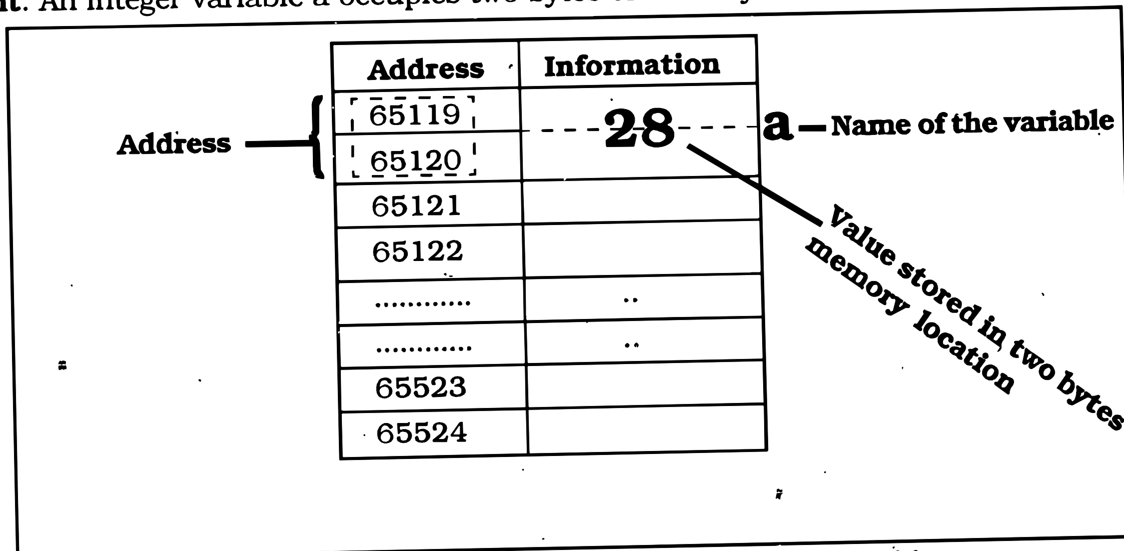



**Figure 2.1** Components associated with a variable

We have always used variables to store values, however a variable can als be used to store the address of another variable such variables are termed as pointe variables. Thus a pointer is a variable, which can contain the address of anothe variable. The program below highlights this fact very clearly.

**Program 2.1**: To show the creation of a pointer variable.

```
#include <stdio.h>
void main( )
{
        int  a = 28;
        int  *ptr;

        ptr =  &a;

        printf( " Address of the variable a = %u \n", &a);
        printf( " Value of the variable a = %d \n", a);
        printf( " Address present in the pointer variable ptr=%u\n", ptr);
}
```

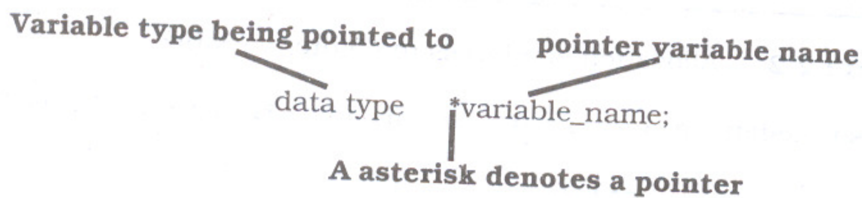
**OUTPUT**
Address of the variable a = 65119
Value of the variable a = 28
Address present in the pointer variable = 65119

Observing the program closely highlights the following points:

1. The address of a variable is accessed with the help of the "&" operator.
2. Using the name of the variable we can accesses the value of a variable.
3. A pointer variable is created by including the operator "*" when the variable is declared.
4. A pointer variable can hold only the address of another variable and not the value of another variable.

## 2.2 POINTER DECLARATION

We have already seen the use of a pointer and its usage, in this section we try to understand all the concepts clearly. A pointer is a variable that contains the address of the memory location of another variable. To create a pointer variable we use the syntax as shown in the Figure below. First we will have to specify the type of data stored in the location identified by the pointer. Then a variable is created along with an asterisk. The asterisk tells the compiler that you are creating a pointer variable. Finally, you give the name of the variable.



## 2.3 POINTER OPERATOR

A pointer operator is used to classify a variable as a pointer and not as a normal variable. Pointer variables can only store the address of another memory location, while a normal variable can only store a value and not an address. The classification is done by representing the variable with a combination of *(asterisk) with the variable name.

For example:

```
int   *ptr;
```

The base type of the pointer defines which type of variables the pointer is pointing to. Technically, any type of operator can point anywhere in memory. All pointer arithmetic is done relative to its base type. So it is important to declare the pointers correctly.

## 2.4 ADDRESS OPERATOR

Once we declare a pointer variable, we must make it to point to something. We can do this by assigning to the pointer the address of the variable you want to point to as in:

```
ptr = &a;
```

This places the memory address of the variable **a** into the pointer variable **ptr**. If **a** is stored in memory 65119 address, then the variable **ptr** has the value 65121. The figure below highlights the situation discussed above.

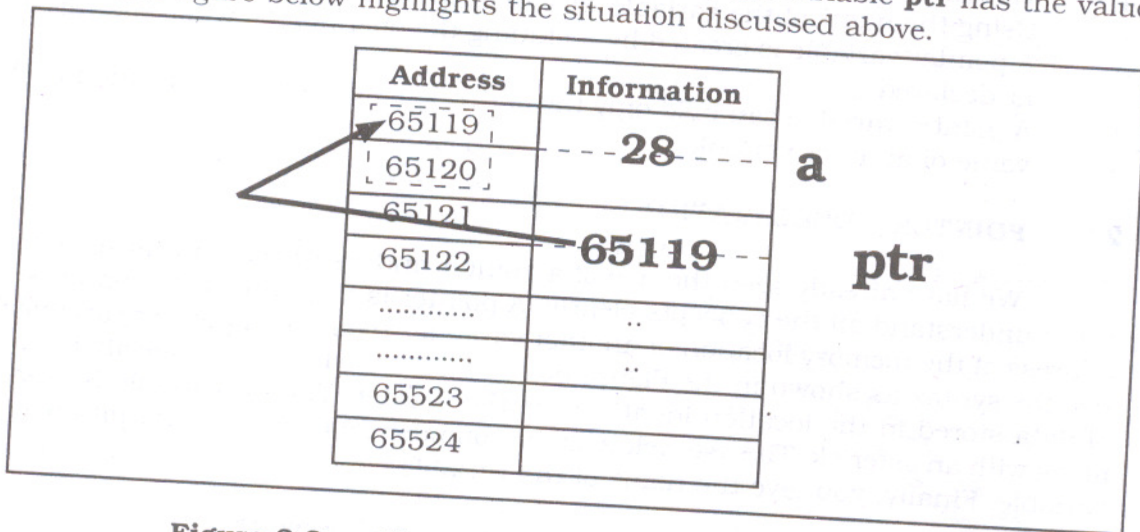



**Figure 2.2** Allocation of address to pointer variable

We can also assign an address to the variable **ptr** directly. Thus the instruction

```
ptr = 65119;
```

would generate a compiler error because it is attempting to assign an integer value to the pointer. The only assignment you can make to the variable **ptr** is the address of a variable, using the address operator, as

ptr = &a;

However, we can assign a value to the pointer ***ptr**, as in

*ptr = 28;

This means "Place the value 28 in the memory address pointed by the variable **ptr**." Since the pointer contains the address 65119, the value 28 is placed in that memory location. And since this is the location of the variable **a**, the value of **a** is also becomes 28. This shows how we can change the value of a variable indirectly using a pointer and the indirection operator.

**Program 2.2** : To display the contents of the variable and their address using a pointer variable.

```
#include <stdio.h>
void main( )
{
        int     num,*int_ptr;
        float   x,*float_ptr;
        char    ch, *char_ptr;

        num=123;
        x=12.34;
        ch='A';
        int_ptr = &num;
        float_ptr = &x;
        char_ptr = &ch;
        printf("Num %d is stored at the address %u\n", *int_ptr ,int_ptr);
        printf("Float %f is stored at the address %u\n",*float_ptr, float_ptr);
        printf("Character %c is stored at the address %u\n",*char_ptr,char_ptr);
}
```

```
OUTPUT
Num 123 stored at the address 65524
Value 12.340000000 stored at the address 65520
Character A stored at the address 65519
```

## 2.5 SPACE REQUIRED FOR POINTER VARIABLES

The amount of memory space allotted when a variable is created and compiled depends on the data type of the variable. The amount of space reserved for a variable has already been explained in the previous section. In this section

we try to understand how much space will be allotted to a pointer variable of different types of data. The program below highlights this point clearly.

**Program 2.3** : To show the amount of space required to store variables and space reserved for pointers.

```
#include <stdio.h>
void main( )
{
        int     a = 5, *int_ptr;
        char    b = 'w', *char_ptr;
        float   c = 17.53, *float_ptr;

        int_ptr = &a;  float_ptr = &b;  char_ptr = &c;

        printf( "Value of integer = %d\n", a);
        printf( "Value of char = %c\n", b);
        printf( "Value  of float = %f\n", c);
        printf( "Amount of space for int ptr = %u bytes\n", sizeof(int_ptr));
        printf( "Amount of space for char ptr = %u bytes\n", sizeof(char_ptr));
        printf( "Amount of space for float ptr = %u bytes\n", sizeof(float_ptr));
}
```

```
OUTPUT
Value of integer = 5
Value of char = w
Value of float =  17.530000
Amount of space for int ptr = 2 bytes
Amount of space for char ptr = 2 bytes
Amount of space for float ptr = 2 bytes
```

What is observed from the above described fact is a very important point. The amount of space to store different variables may vary. However the size of all the addresses available is the same and depends on the word length of the computer being used. Thus we observe that all the outputs are the same i.e., a pointer is created to hold the address of another memory location and the size of the address is the same immaterial of the type of data it holds, thus the outputs are the same.

## 2.6 POINTERS AND FUNCTIONS

Pointers are used very much with functions. Also sometimes complex function can easily be represented and accessed only with a pointer. Arguments can be passed to one of the following methods.

1. Passing values of the arguments (Call by Value)
2. Passing the addresses of the arguments (Call by Reference)

### 2.6.1 Call by value

We have seen that when a function is invoked, a correspondence is established between the formal and actual parameters. A temporary storage is created where the value of the actual parameter is stored. The formal parameter picks up its value from this storage area. This mechanism of data transfer, between the actual and formal parameters, allows the actual parameters to be an expression, arrays, etc. Such parameter is called value parameters and mechanism of data transfer is referred to as **Call-By-Value**. The corresponding formal parameter represents a local variable in the called function. The current value of the corresponding actual parameter becomes the initial value of the formal parameter. The value of formal parameter may then change in the body of the subprogram by assignment or input statements. This will not change the value of the actual parameter.

**Definition**

**Value parameter :** A parameter that receives a copy of the value of coresponding argument.

**Program 2.4 :** A C program to illustrate the function using call by value mechanism.

```
#include <stdio.h>
main( )
{
        int     a,b;
        void function(int , int );
        a=20;
        b=30;
        printf("a =%d b = %d before function call \n ",a,b);
        function(a,b);
        printf("a =%d b = %d after function call \n ",a,b);
}
/*      Call by value function      */
void function(int x, int y)
{
        x =  x + x;
        y =  y + y;
}
```

**OUTPUT**
a = 20 b = 30 before function call
a = 20 b = 30 after function call

### 2.6.2 Call by reference

Whenever a function call is made if we pass the address of a variable to a function, the parameters receiving the address should be pointers. The process of calling a function using pointers to pass the address of variable is known as **Call-By-Reference**. The function, which is called by 'reference', can change the

value of the variable used in the call i.e., any changes made to the copied variables will affect the original variables also.

**Definition**

**Reference parameter :** A parameter that receives the memory address of the caller's argument.

**Program 2.5 :** A C program to illustrate the function using call by reference mechanism.

```
#include <stdio.h>
main( )
{
        int     a,b;
        void function(int *, int *);
        a=20;
        b=30;
        printf("a =%d b = %d before function call \n ",a,b);
        function(&a, &b);
        printf("a =%d b = %d after function call \n ",a,b);
}
/*      Call by reference function      */
void function(int *x, int *y)
{
        *x  =  *x + *x;
        *y  =  *y + *y;
}
```

**OUTPUT**

a = 20 b = 30 before function call
a = 40 b = 60 after function call

When the function is called, the address of the variable **a**, not its value, is passed into the function. In the function the receiving variables are declared as a pointer thus the address of the variable is passed. Since x represents the address of **a**, the value of **a** is changed from 20 to 40. Therefore, the output of the above program segment will be as shown above.

**Program 2.6 :** A C program to exchange the contents of the two variables using call by value and call by reference.

```
#include <stdio.h>
main( )
{
        int     a,b;
        void swap_val(int , int );      /*  Function prototype  */
        void swap_ref(int *, int * );
        printf("Enter two numbers \n ");
```

```
        scanf("%d %d",&a,&b);
        printf("a =%d b = %d before function call  \n ",a,b);
        swap_val(a,b);
        printf("a =%d b = %d after function call using call by value \n ",a,b);
        swap_ref(&a,&b);
        printf("a=%d b=%d after function call using call by reference\n ",a,b);
}
/*      Function to exchange two values using call by value     */
void swap_val(int x, int y)
{
        int     temp;
        temp  =   x;
        x   =  y;
        y   =  temp;
}
/*      Function to exchange two values using call by reference      */
void swap_ref(int *x, int * y)
{
        int     temp;
        temp  =   *x;
        *x    =  *y;
        *y    =  temp;
}
```

**OUTPUT**
Enter two numbers
100 200
a = 100 b = 200 before function call
a = 100 b = 200 after function call using call by value
a = 200 b = 100 after function call using call by reference

Using the technique of call by reference in an intelligent manner it is possible for us to make a function return more than one value at any instant of time, whereas till now our function could return only one value.

**Program 2.7 :** A C program to make a function returning more than one value.

```
#include <stdio.h>
#define   PI   3.1415
main( )
{
        int     r;
        float   a,c;
        void calculate(int , float *, float *);
        printf("Enter the radius of a circle \n ");
        scanf("%f",&r);
        calculate(r,&a,&c);
```

```
        printf("Radius  =%d \n ",r);
        printf("Area  =%f  \n ",a);
        printf("Circumference  =%f  \n ",c);
}
/*      Function to calculate the area and circumference of a circle    */
void calculate(int x , float *y , float *z)
{
        *y =    PI * x * x;
         *z =  2 * PI * x;
}
```

```
OUTPUT
Radius  = 7
Area  = 153.933500
Circumference  = 43.981000
```

## 2.7 POINTER AND ARRAYS

An array is a name given to a set of memory locations of the same type of data. It is a very popular data type especially when we are working with large amounts of data. However discussing arrays and not discussing pointers and vice versa is not possible at all. In actuality all arrays make use of pointers internally.

Before we start studying pointers, let us note a few points about pointers:

1. An array is a collection of memory locations called by the same name and holding the same type of data and accessed with the help of a subscript.
2. Before using an array it must be declared.

**Example 2.1 :** char  name[20];

3. The accessing of elements at any location is done with the help of the name of the array. The name of the array always represents the address of the first location of the array. Thus data at any location is accessed with the help of the expression.

**LOC( i ) = Address of first location + Subscript value * size of data type**

### 2.7.1 One-dimensional array

Consider the following declaration

int a[ ], *ptr;

The pointer could be assigned the address of a[0]

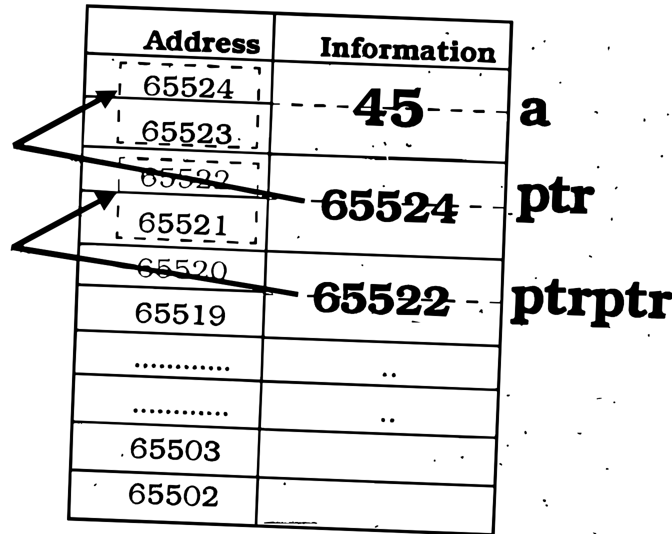



**Figure 2.3** To show the concept of a pointer to a pointer

Here, **a** is an ordinary *int*, **ptr** is a pointer to an *int*, whereas **ptrptr** is a pointer to a *int* pointer. In principle, there could be a pointer to pointer. There is no limit on how far can we go on extending this definition. Possibly, till the point we can comprehend it. And that point of comprehension is usually a pointer to pointer. Excess indirection is difficult to follow and process to pointer.

## 2.11 MEMORY ALLOCATION

In the previous section we have described the the storage classes which determined how memory for variables are allocated by the compiler. When a variable is defined in the source program, the type of the variable determines how much memory the compiler allocates. When the program executes, the variable consumes this amount of memory regardless of whether the program actually uses the memory allocated. This is particularly true for arrays. However, in many problems, it is not clear at the outset how much memory the program will actually need. Up to now, we have declared arrays to be "large enough" to hold the maximum number of elements we expect our application to handle. If too much memory is allocated and then not used, there is a waste of memory. If not enough memory is allocated, the program is not able to handle the input data.

Consider the following array declaration,

```
int     a[10][10];
```

This type of declaration would be generally used to write a generalized program to perform operations on matrices. The moment the declaration is made the system reserves a total of 100 locations each of two bytes for the array(200 bytes). However when we are running the program we may enter a matrix of just

three rows and three columns. This results in wastage of space of 91 (182 bytes) memory locations. This method of allocating space when the variable is created is called as **static allocation** of space.

| Definition |
|---|
| **Static allocation:** Creation of storage space in memory for a variable at compile-time(cannot be changed at run-time). |

### 2.11.1 Dynamic memory allocation

In programming we may come across situations where we may have to deal with data, which is dynamic in nature. The number of data items may change during the executions of a program. The number of customers in a queue can increase or decrease during the process at any time. When the list grows we need to allocate more memory space to accommodate additional data items. Such situations can be handled more easily by using dynamic techniques. This method of allocating space while running the program is called as **dynamic allocation** of space. Dynamic data items at run time, thus optimizing usage of storage space.

| Definition |
|---|
| **Dynamic allocation:** Creation of storage space in memory for a variable during run-time. |

The above said concept can be achieved with the help of the four standard library functions **malloc( ), calloc( ), realloc( )** and **free( ).**

| C function | Task to be performed by the function |
|---|---|
| malloc | Allocates memory requests size of bytes and returns a pointer to the first byte of allocated space. |
| calloc | Allocates space for an array of elements initializes them to zero and returns a pointer to the memory |
| free | Frees previously allocated space |
| realloc | Modifies the size of previously allocated space. |

### 2.11.2 Memory allocations process

The Figure 2.4 shows the conceptual view of storage of a C program. According to the conceptual view the program instructions and global and static variable in a *permanent storage* area and local area variables are stored in *stacks*. The memory space that is located between these two regions is available for dynamic allocation during the execution of the program. The *free* memory region is called the **heap**. The size of heap keeps changing when program is executed due to creation and death of variables that are local for functions and blocks. Therefore it is possible

to encounter memory overflow during dynamic allocation process. In such situations, the memory allocation functions mentioned above will return a NULL pointer.

**Definition**

**Free store (heap) :** A pool of memory locations reserved for allocation and deallocation of dynamic data.

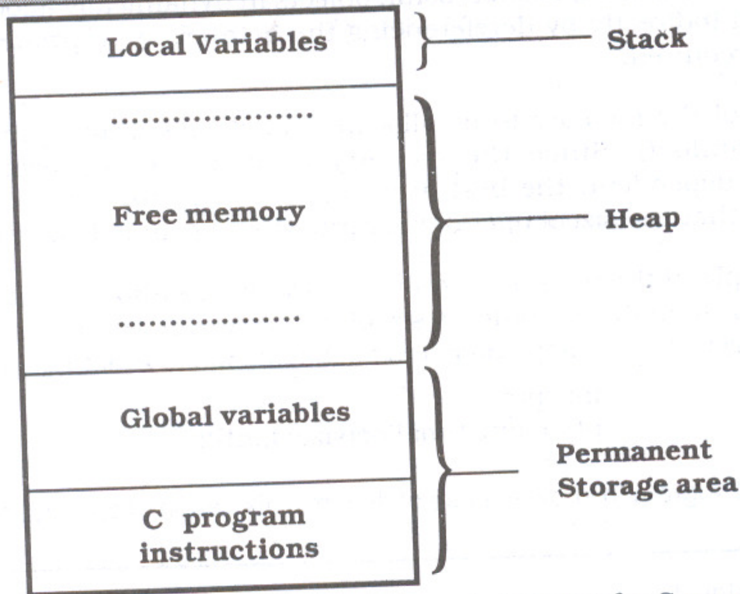



**Figure 2.4** Conceptual view of storage of a C program

### 2.11.2.1 Allocating a block of memory (malloc function)

A block of memory may be allocated using the function malloc. The malloc function reserves a block of memory of specified size and returns a pointer of type void. This means that we can assign it to any type of pointer. It takes the following form:

```
ptr=(cast-type*)malloc(byte-size);
```

**ptr** is a pointer of type **cast-type** the malloc returns a pointer (of cast type) to an area of memory with size byte-size.

**Example 2.3 :** ptr=(int *)malloc(100*sizeof(int));

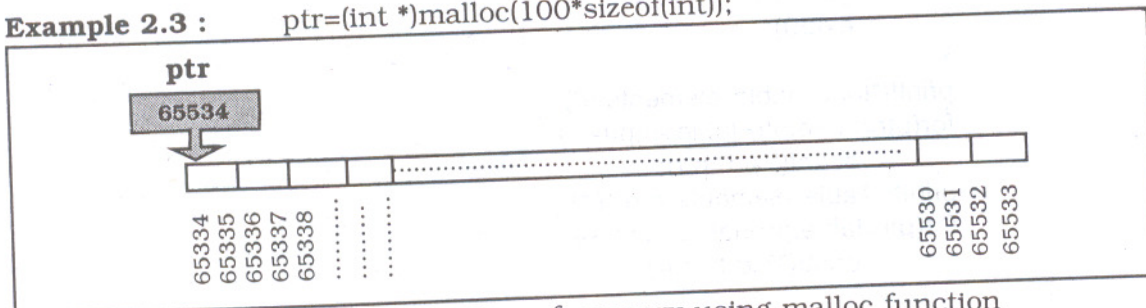



**Figure 2.5** Allocation of memory using malloc function

**Example 2.4 :** It is possible to allocate a block of memory for several elements of the same type by giving the appropriate value as an argument. Suppose, we wish to allocate memory for 100 **float** numbers. Then, if **fptr** is a float *, the following statement does the job:

```
fptr = (float *) malloc(100 * sizeof(float));
```

### 2.11.2.2 Allocatingmultiple blocks of memory (calloc function)

**calloc( )** is another memory allocation function that is normally used to request multiple blocks of storage each of the same size and then sets all bytes to zero. The general form of **calloc( )** is:

```
ptr=(cast-type*) calloc(n,elemment-size);
```

The above statement allocates contiguous space for **n** blocks each of size *element-size* bytes. All bytes are initialized to zero and a pointer to the first byte of the allocated region is returned. If there is not enough space a NULL pointer is returned.

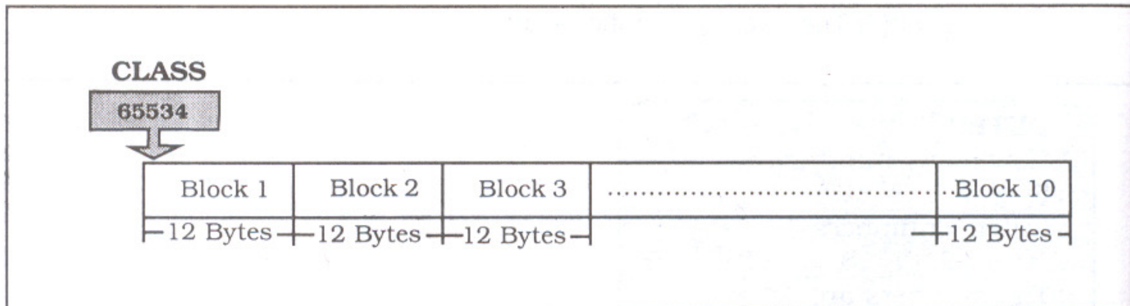
**Example 2.5 :** The following segement of a program allocates space for a structure variable:

```
struct std
{
        int     regno;
        char    name[10];
};
typedef  struct std STUDENT;

STUDENT  *CLASS;

CLASS=(STUDENT *)calloc(10,sizeof(STUDENT));
```

STUDENT is of type **struct std** having **regno** and **name**. The calloc( ) function allocates to hold for 10 students.



```
Regno 11111
Name Arun
Regno 22222
Name Babu
```

**Difference between the functions malloc( ) and calloc( )**

The definitions for the library functions malloc( ) and calloc( ) is present in the header file "**alloc.h**". malloc( ) and calloc( ) are similar in their works. However they differ in two important aspects. The function malloc( ) needs one argument which identifies the total space to be reserved and the function calloc( ) requires two arguments the first argument identifies the total locations required and the second argument identifies the space required for each location.

Another difference between the functions malloc( ) and calloc( ) is that the memory space created with malloc( ) contains garbage values whereas the memory space created with calloc( ) is initialized to zeros if it points to an integer and blank spaces if it points to a character.

### 2.11.2.3 Releasing the used space (free function)

Compile time storage of a variable is allocated and released by the system in accordance with its storage class. With the dynamic runtime allocation, it is our responsibility to release the space when it is not required. The release of storage space becomes important when the storage is limited. When we no longer need the data we stored in a block of memory and we do not intend to use that block for storing any other information, we may release that block of memory for future use, using the free function.

```
free(ptr);
```

**ptr** is a pointer that has been created by using malloc or calloc.

### 2.11.2.4 To alter the size of allocated memory (realloc function)

The memory allocated by using calloc or malloc might be insufficient or excess sometimes in both the situations we can change the memory size already allocated with the help of the function realloc. This process is called reallocation of memory. The general statement of reallocation of memory is :

```
ptr=realloc(ptr,newsize);
```

This function allocates new memory space of size *newsize* to the pointer variable **ptr** and returns a pointer to the first byte of the memory block. The allocated new block may be or may not be at the same region.