The *for* Statement contd.

- Another unique aspect of ***for*** loop is that one or more sections can be omitted, if necessary. e.g.

```
m = 5;
for ( ; m!=100; )
{printf("%d %d\n", m);m=m+5;}
```

- Both the initialization and increment sections are omitted in the ***for*** statement.
- Initialization has been done before the for statement and the control variable is incremented inside the loop. In such cases, the sections are ***left 'blank'. However, the semicolons separating the sections must remain.***
- If the test-condition is not present, the for statement sets up an ***infinite loop***.
- ***Infinite loops are broken using break or goto statements.***
- We can set up ***time delay*** loops using ***null statement*** as follows:

```
for(j= 1000; j > 0; j = j-1);
```

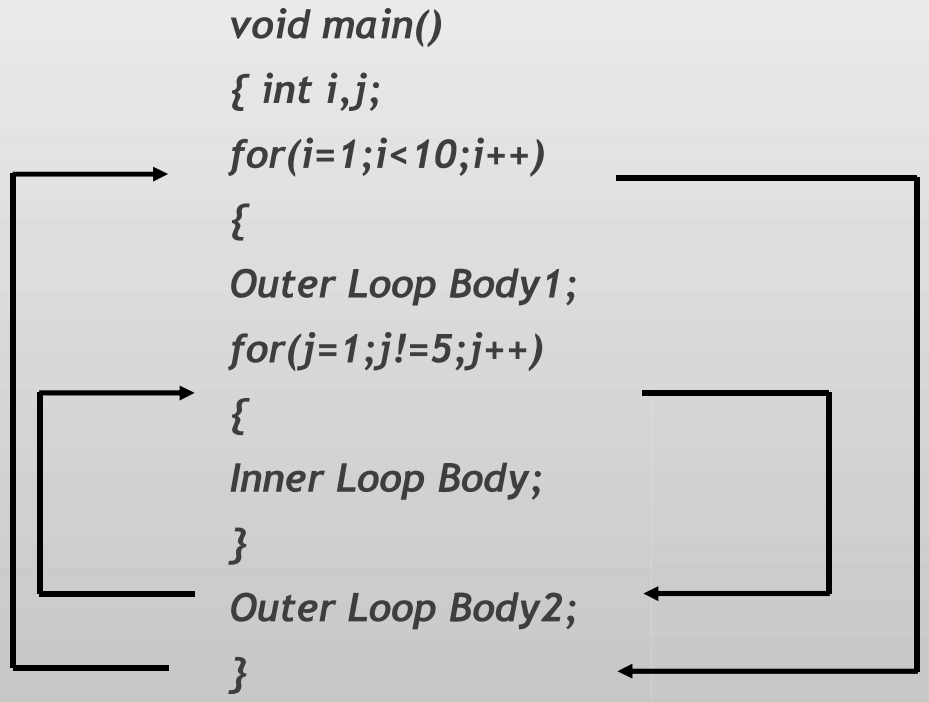
- Loop is ***executed 1000 times without producing any output; simply causes a time delay.*** Notice that the body of the loop contains only a semicolon, known as a null statement. Can also be written as:

```
for (j=1000; j > 0; j = j-1)
```

- This implies that the C compiler will not give an error message if we place a semicolon by mistake at the end of a ***for*** statement. The semicolon will be considered as a ***null statement***.

Nesting of *for* Loops

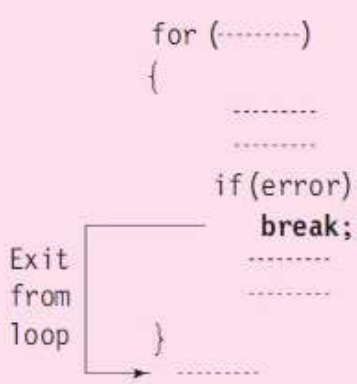
- Nesting of loops, that is, one for statement within another for statement, is allowed in C. For example, two loops can be nested as follows:

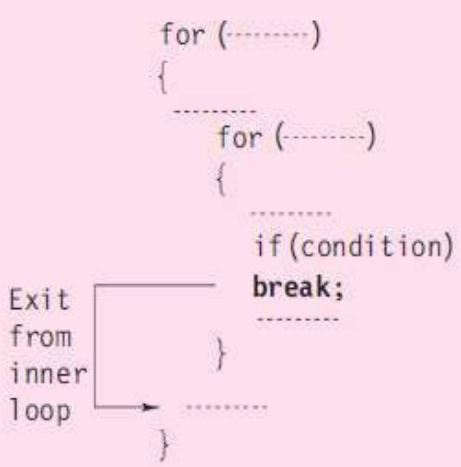


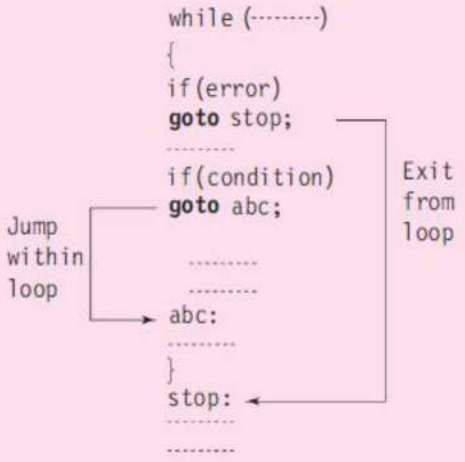
- The nesting may continue up to any desired level. The loops should be properly indented so as to enable the reader to easily determine which statements are contained within each ***for*** statement.
- ***Note: ANSI C allows up to 15 levels of nesting. However, some advanced compilers permit more nesting levels.***

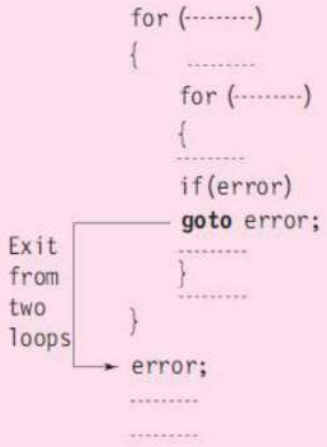
Jumps in Loops

- Loops perform set of operations repeatedly until the control variable fails the test condition.
- ***Sometimes when some part of the loop body is to be skipped then jump control statements are utilized.***
- Early exit from loop is accomplished using ***break or goto*** statements.
- When ***break*** statement is encountered inside a loop, the loop is immediately excited and the program continues with the next block of statements. In nested the loop containing break statement would only exit from containing it.
- ***goto*** statement can transfer the control to any place in a program, it is useful to provide branching in loop.
- ***goto*** is utilized to exit from deeply nested loops when an error occurs.



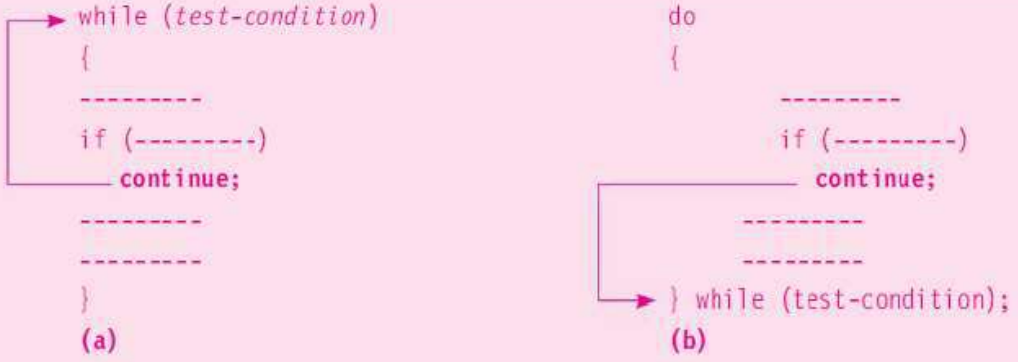


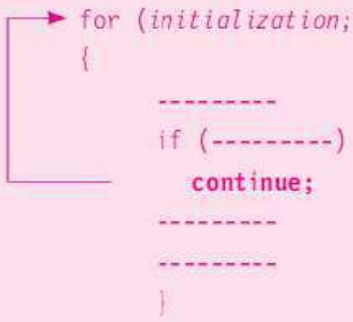




Skipping a Part of a Loop

- During the loop operations, it may be necessary to skip a part of the looping body under certain conditions. e.g., in job application processing, we might like to exclude the processing of data of applicants belonging to a certain category.
- On reading the category code of an applicant, a test is made to see whether his application should be considered or not. If it is not to be considered, the part of the program loop that processes the application details is skipped and the execution continues with the next loop operation.
- ***C supports continue*** statement responsible for skipping a set of instruction in loop.
- Unlike **break** which causes the loop to be terminated, ***continue***, causes the loop to be continued with the next iteration after skipping any statements in between.
- ***Note: continue tell C compiler "SKIP THE FOLLOWING STATEMENTS AND CONTINUE WITH THE NEXT ITERATION".***





Jumping out of the Program

- ***break*** and ***goto*** statements shifts the compilation control by jumps out of loop.
- In a similar way, in order to jump out of a program we use ***exit()*** library function.
- In certain case, we wish to break out of a program and return to the operating system, we can use ***exit( )***.
- ***Syntax:***

```
. . . . . .
. . . . . .
if(test-condition) exit(0);
. . . . . .
. . . . . .
```

- The ***exit( )*** function takes an integer value as its argument. Normally zero is used to indicate normal termination and a nonzero value to indicate termination due to some error or abnormal condition.
- The use of ***exit( )*** function requires the inclusion of the header file **<stdlib.h>**.

Module3

Array, String, Structures and Union

Arrays

- We used only the fundamental data types, namely ***char***, ***int***, ***float***, ***double***, and variations of ***int*** and ***double***.
- Although these types are very useful, they are constrained by the fact that a variable of these types can store only one value at any given time.
- They can be used only to handle limited amounts of data. In many applications, however, we need to handle a large volume of data in terms of reading, processing and printing.
- ***To process large data sets, a derived data type known as array is used for efficient storing, accessing and manipulation of data items.***
- An array is a ***fixed-size*** sequenced collection of elements of the same data type. It is simply a grouping of like-type data. In its simplest form, an array can be used to represent a list of numbers, or a list of names. Some examples where the concept of an array can be used:

1. List of temperatures recorded every hour in a day, or a month, or a year.
2. List of employees in an organization.
3. List of products and their cost sold by a store.
4. Test scores of a class of students.
5. List of customers and their telephone numbers.
6. Table of daily rainfall data.

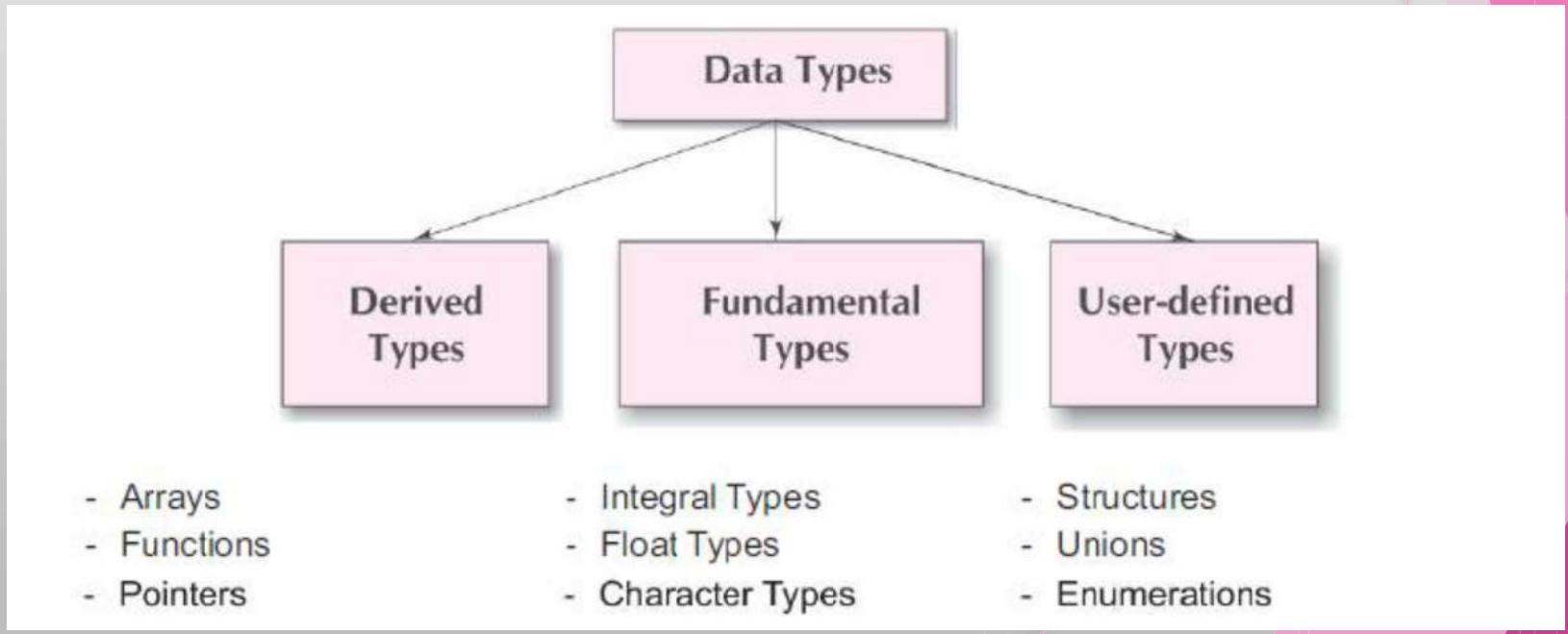
Arrays contd.

- An array provide a convenient structure for data representation, and is classified as one of the data structure in C. Other data structures are ***structures, lists, queues, and trees***.
- An array is a sequenced collection of related data items that share a common name. For instance, we can use an array name ***salary*** to represent a ***set of salaries*** of a group of employees in an organization. We can refer to the individual salaries by writing a number called ***index*** or ***subscript*** in brackets after the array name. For example, ***salary [10]***. While the ***complete set of values is referred to as an array***, individual values are called ***elements***.
- The ability to use a single name to represent a collection of items and to refer to an item by specifying the item number enables us to develop concise and efficient programs. e.g. We use a loop construct, discussed earlier, with the subscript as the control variable to read the entire array, perform calculations, and print out the results.
- We can use arrays to represent not only simple lists of values but also tables of data in two, three or more dimensions. In this chapter, we introduce the concept of an array and discuss how to use it to create and apply the following types of arrays.

1. One-dimensional arrays
2. Two-dimensional arrays
3. Multidimensional arrays

Data Structures

- C supports a rich set of derived and user-defined data types in addition to a variety of fundamental types.
- ***Arrays and structures*** are referred to as ***structured data types*** because they can be used to represent data values that have a structure of some sort. Structured data types provide an organizational scheme that shows the relationships among the individual elements and facilitate efficient data manipulations.
- ***In programming parlance, such data types are known as data structures.***
- In addition to arrays and structures C support : ***Linked lists, Stacks, Queues, Trees***



One-Dimensional Array

- A list of items can be given one variable name using only one subscript and such a variable is called a ***single-subscripted variable or a one-dimensional array.*** In mathematics, we often deal with variables that are single-subscripted. e.g. To calculate sum of n values we have

$$S = \sum_{i=1}^{n} x_i$$

- ***In C, single-subscripted variable*** x_i can be expressed as ***x[1], x[2], x[3],.....x[n].*** The subscript can begin with number 0. That is ***x[0] is allowed. e.g.*** if we want to represent a set of five numbers, say (35,40,20,57,19)

by an array variable number, then we may declare the variable number as

int number[5]; and the computer reserves five storage locations as

- The values to the array elements can be assigned as ***number[0] = 35;***

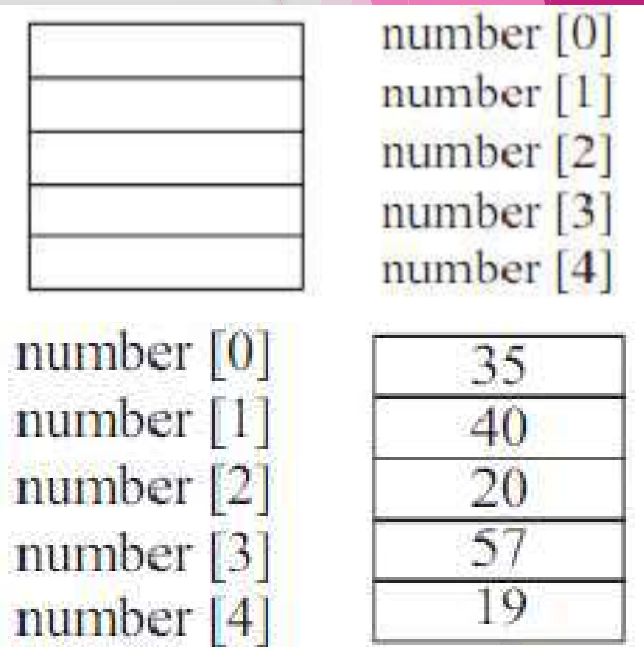
number[1] = 40; number[2] = 20; number[3] = 57; number[4] = 19;

- The array number in memory is stored as shown. The array elements

can be used in program like any other C variable. e.g. a=number[0]+10;

number[4]=number[0]+number[2];

- ***Note: C does not perform bounds checking and therefore care should be Exercised to ensure that the array indices are within the declared limits.***



One-Dimensional Array Declaration

- Arrays must be declared before they are used so that the compiler can allocate space for them in memory. The general form of array declaration is ***type variable-name[size];***
- The ***type*** specifies the ***type of element*** that will be contained in array, such as ***int***, ***float***, ***or char*** and the ***size*** indicates the maximum number of elements that can be stored inside the array. e.g. ***float height[50];*** declares the height to be an array containing 50 real elements. Subscripts 0 to 49 are valid. ***int group[10];*** declares the ***group*** as an array to contain a maximum of 10 integer constants.
- ***Note: 1. Any reference to the arrays outside the declared limits would not necessarily cause an error. Rather, it might result in unpredictable program results.***

2.The size should be either a numeric constant or a symbolic constant.

- ***C treats character strings simply as arrays of characters.*** The size in character string represents the maximum number of characters that the string can hold.

e.g. char name[10]; declares ***name as a character array (string) that hold a maximum of 10 characters.***

- When the ***compiler sees a character string,***

it ***terminates array with an additional null character.***

Thus, the element ***name[10] holds the null character '\0'.***

One-Dimensional Array Initialization

- Array declaration leads to array initialization, otherwise it will have garbage value. ***Array is initialized at either Compile Time or at Run Time.***
- ***Compile Time Initialization:*** We initialize the array elements in the same way as the ordinary variables when they are declared. The general form of array initialization is: ***type array-name[size] = { list of values };***
- The values in the list are separated by commas. e.g.,

int number[3] = { 0,0,0 };

will declare ***number*** as an array of size 3 and will assign zero to each element.

- The number of values in the list is less than the number of elements, then only that many elements will be initialized. The remaining elements will be set to zero automatically. e.g.

float total[5] = {0.0,15.75,-10};

will initialize the first three elements to ***0.0, 15.75, -10.0 and the remaining two elements are declared to be zero.***

- Similarly, in char array declaration

char city[5] = {'B'};

will initialize the first element to ***'B' and the remaining elements are declared NULL.***

- ***Note: If we have more initializers than the declared size, the compiler will produce an error. e.g. int a[2]={10,20,30}; leads to initialization error.***

Run-Time Initialization

- ***Run-Time Initialization:*** An array can be explicitly initialized at run time. This approach is usually applied for initializing large arrays. e.g.

```
--------
--------
for (i = 0; i < 100; i = i+1)
{
  if    i < 50
        sum[i] = 0.0;          /* assignment statement */
  else
        sum[i] = 1.0;
}
--------
--------
```

- The first 50 elements of the array ***sum*** are initialized to ***0.0*** while the remaining ***50*** elements are initialized to ***1.0*** at run time.

Two-Dimensional Array

- We have discussed the array variables that can store a list of values. There could be situations where a table of values will have to be stored. e.g. Consider the following data table for the sales of three items by four sales girls:

	Item1	*Item2*	*Item3*
Salesgirl #1	310	275	365
Salesgirl #2	210	190	325
Salesgirl #3	405	235	240
Salesgirl #4	260	300	380

- The tables contains total 12 values, three in each line. The table behaves as a 4x3 matrix having 4 rows and 3 columns. In mathematics this particular representation is carried out using two subscript i.e. a_{ij}, where a denotes the entire matrix and a_{ij} refers to the value in the i^{th} row and j^{th} column.

Two-Dimensional Array: Initialization

- Two-dimensional arrays are declared as follows: ***type array_name [row_size][column_size];***
- Like the one-dimensional arrays, two-dimensional arrays may be initialized by following their declaration with a list of initial values enclosed in braces. e.g., ***int table[2][3] = { 0,0,0,1,1,1};*** initializes the elements of the first row to zero and the second row to one.
- Initialization is done row by row. The above statement can be equivalently written as ***int table[2][3] = {{0,0,0}, {1,1,1}};*** by surrounding the elements of each row by braces.
- A 2-D array in the form of a matrix as shown below: ***int table[2][3] = {{0,0,0},{1,1,1}};***

***h*ere commas are required after each brace that closes off a row, except in the case of the last row.**

- When the array is completely initialized with all values, explicitly, we need not specify the size of the first dimension, i.e.,

int table [ ] [3] = {{ 0, 0, 0},{ 1, 1, 1}};

is permitted.

- If the values are missing in an initializer, they are automatically set to zero. For instance, the statement

int table[2][3] = {{1,1},{2}};

Will initialize the first 2 elements of the 1st row to 1, the 1st element of the 2nd row to 2, and all other elements to zero.

- When all the elements are to be initialized to zero, the following short-cut method may be used. e.g.

int m[3][5] = { {0}, {0}, {0}};

The first element of each row is explicitly initialized to zero while other elements are automatically initialized to zero.

Multi-Dimensional Array

- C allows arrays of three or more dimensions. The exact limit is determined by the compiler. The general form of a multi-dimensional array is

type array_name[s1][s2][s3]....[sm];

where s_i is the size of the i^{th} dimension. Some example are:

int survey[2][5][12];

float table[5][4][5][3];

survey is a ***3-dimensional array*** declared to contain 120 ***integer type elements***. ***Table*** is a ***4-dimensional array*** containing 300 ***floating-type elements***.

- ***Note: ANSI C does not specify any limit for array dimension. However, most compilers permit 7 to 10 dimensions.***

Year 1

month city	1	2	.....	12
1				
.				
.				
.				
.				
5				

Year 2

month city	1	2	.....	12
1				
.				
.				
.				
.				
5				

Dynamic Arrays

- An array created at compile time by specifying size in the source code has a fixed size and cannot be modified at run time.
- The process of allocating memory at compile time is known as ***static memory allocation*** and the arrays that receive static memory allocation are called ***static arrays***.
- ***The static memory allocation is fine as long as we know exactly what our data requirements are.***
- In certain environments where we want to use an array that can vary greatly in size. In C it is possible to allocate memory to arrays at run time. This feature is known as ***dynamic memory allocation*** and the arrays created at run time are called ***dynamic arrays***.
- In ***dynamic array*** the ***array definition*** is done during run time.
- Dynamic arrays are created using what are known as ***pointer variables*** and ***memory management functions malloc, calloc, and realloc***. These functions are included in the header file ***<stdlib.h>***.
- ***Note: The concept of dynamic arrays is used in creating and manipulating data structures such as linked lists, stacks and queues.***

Strings

- A string is a sequence of characters that is treated as a single data item. Any group of characters (except double quote sign) defined between double quotation marks is a string constant. e.g. “Man is obviously made to think.”
- To print *“”* in string we may use it ***with a back slash***. This is now expressed as

  “\” Man is obviously made to think,\” said Pascal.”
- e.g.

  printf(“\” Well Done !”\”);
- will show output as

  “ Well Done !”
- While the statement

  printf(“Well Done !”);
- will provide output as

  Well Done !
- Character strings are often used to build meaningful and readable programs. The common operations performed on character strings include:

1. Reading and writing strings.
2. Combining strings together.
3. Copying one string to another.
4. Comparing strings for equality.
5. Extracting a portion of a string.

Strings Declaration and Initialization

- C does not support strings as a data type, but character arrays are used to represent strings.
- ***A string variable is any valid C variable name and is always declared as an array of characters. The general form of declaration of a string variable is:***

  char string_name[size];

- The size determines the number of characters in the ***string_name***. Some examples are: ***char city[10]; char name[30];***
- ***When the compiler assigns a character string to a character array, it automatically supplies a null character ('\0') at the end of the string. Therefore, the size should be equal to the maximum number of characters in the string plus one.***
- Like numeric arrays, character arrays may be initialized when they are declared. C permits character array initialization as:

  char city [9] = " NEW YORK ";

  char city [9]={'N','E','W',' ','Y','O','R','K','\0'};

- The reason that ***city*** had to be ***9 elements long*** is that the ***string NEW YORK contains 8 characters*** and ***one element space*** is ***provided for null terminator.***
- ***Note: While initializing a character array by listing individual elements, the null terminator (\0) should be explicitly introduced in the character array.***

Strings Declaration and Initialization contd.

- C permits character array initialization without specifying total number of elements.

  char string [ ] = {'G','O','O','D','\0'};

- The character array declaration size can be more than the string size in the initializer. e.g. ***char str[10] = "GOOD"; here the computer creates a character array of size 10, and places the value "GOOD" in it, and initializes all other elements to NULL.***
- The storage will now look like:

- ***Note: We cannot separate the initialization from declaration. e.g. char str2[10];str2="HELLO";***
- The familiar input function ***scanf()*** with ***%s as format specifier*** to read in a string of characters. e.g.: ***char address[10]; scanf("%s", address);***
- The problem with ***scanf*** is that it terminates its input on the first white space. A whitespace includes blanks, tabs, carriage returns, form feeds, and new lines. e.g. if ***NEW YORK*** is typed in the in the terminal then only string "***NEW"*** will be read into the array address.

Using *getchar, gets, putchar, puts* Functions

- ***getchar()*** function is used to read successive single characters from the input and place them into a character array. An entire line of text can be read and stored in an array.
- The reading is terminated when the newline character (‘***n**’) is entered and the null character is then inserted at the end of the string. ***getchar()*** is ***char ch; ch = getchar( );***
- ***Note: getchar() function has no parameters.***
- Another method of reading a string of text containing whitespaces is to use the library function ***gets*** available in the ***<stdio.h>*** header file. ***gets is a simple function with one string parameter*** and called as under:

gets (str);

- where ***str*** is a string variable declared properly. It reads characters into ***str*** from keyboard until a ***\n*** character is encountered and then appends a null character to the string. ***Unlike scanf(), gets() does not skip the whitespaces. e.g.***

```
char line [80];
gets (line);
printf (“%s”, line);
```

- ***Note: Be careful not to input more character that can be stored in the string variable used. Since C does not check array-bounds, it may cause problems.***

Using *getchar, gets, putchar, puts* Functions contd.

- Like ***getchar()***, C supports another character handling function ***putchar()*** to output the values of character variables. Putchar() is written as:

```
char ch = 'A';
putchar (ch);
```

- ***putchar()*** requires one parameter. This statement is equivalent to: ***printf("%c", ch);***
- ***putchar()*** function can be used repeatedly to output a string of characters stored in an array using a loop.

```
char name[6] = "PARIS"
for (i=0, i<5; i++)
putchar(name[i];
putchar('\n');
```

- Another method of printing string values is by ***puts()*** declared in the header file <stdio.h>. This is a one parameter function and invoked as: ***puts(str);*** where ***str*** is a string variable containing a string value.

Character Arithmetic Operations & String Handling Functions

- C allows character manipulation just like the integer manipulation. The C library supports a function that converts a string of digits into their integer values. The function takes the form:

  x = atoi(string);

- ***x*** is an integer variable and string is a character array containing a string of digits. e.g. ***number = "1988"; year = atoi(number); number*** is a string variable which is assigned the string constant ***"1988"***. ***atoi()*** function converts the string "1988" (contained in ***number***) to its numeric equivalent ***1988*** and is assigned to the integer variable year.
- ***Arithmetic operation are done on the ASCII equivalent numeric representation of the character value. e.g.***

  x = ASCII value of '7' - ASCII value of '0' =55-48 = 7

- ***strcat(string1, string2) concatenates two strings; strcmp(string1, string2) compares two strings identified by the argument and has value 0 if they are equal; strcpy(string1, string 2) almost work like string-assignment operator with string2 may be a character array variable or a string constant.***

Function	*Action*
strcat()	concatenates two strings
strcmp()	compares two strings
strcpy()	copies one string over another
strlen()	finds the length of a string

Structures

- Array are used to represent a group of data items that belong to same type, such as int or float. However, arrays can't be used to represent a collection of data items of different types using single name.
- C supports a constructed data type known as structures, a mechanism for packing data of different types.
- Structure is a convenient tool for handling group of logically related data items such as student_name, roll_number, and marks.

Arrays Vs Structures

Both the arrays and structures are classified as structured data types as they provide a mechanism that enable us to access and manipulate data in a relatively easy manner. But they differ in a number of ways.

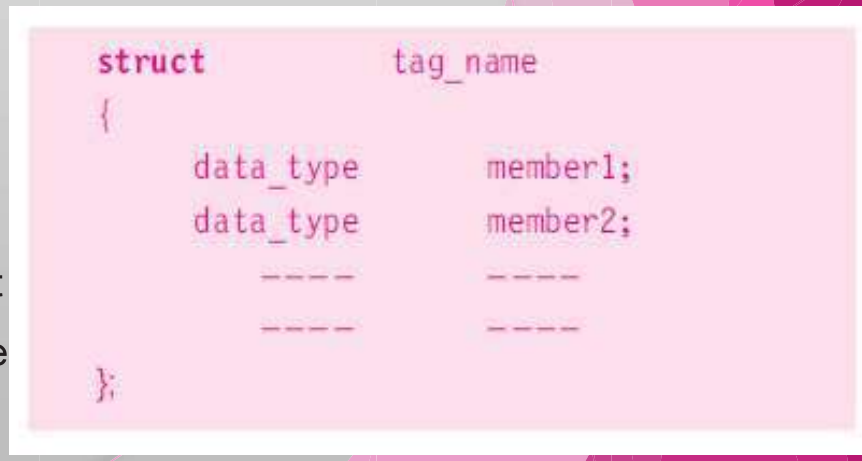
1. An array is a collection of related data elements of same type. Structure can have elements of different types.
2. An array is derived data type whereas a structure is a programmer-defined one.
3. Any array behaves like a built-in data type. All we have to do is to declare an array variable and use it. But in the case of a structure, first we have to design and declare a data structure before the variables of that type are declared and used.

Structures Definition

- Structures must be defined first for their format that may be used later to declare structure variables.
- Consider a book database containing book name, author, number of pages, and price. The structure variable is defined as:

```
struct book_bank
{
char title[20];
char author[15];
int pages;
float price;
};
```

- ***struct*** keyword declares a structure to hold the details of four data fields, namely **title**, **author**, **pages**, and **price.** These fields are called ***structure elements or members***. Each member belong to a different type of data. ***book_bank*** is the name of the structure and is called the ***structure tag.***



Structure Declaration

- After defining structure format we declare variable of that type.
- Structure variable declaration is similar to any data type variable declaration. Structure declaration include

1. ***Keyword struct***
2. ***Structure tag name***
3. ***A terminating semicolon***

- e.g. ***struct book_bank, book1, book2, book3;*** declares ***book1,book2,*** and ***book 3*** as variables of ***struct book_bank***
- Each one of these variables has four members as specified by the template. The complete declaration is

```
struct book_bank
{
char title[20];
char author[15];
int pages;
float price;
};
struct book_bank book1, book2, book3;
```

- ***Remember the members of a structure themselves are not variables. They don't occupy any memory until they are associated with the structure variables such as book1. When compiler comes across a declaration statement, it reserves memory space for the structure variables.***

Accessing Structure Members

- Structure Members themselves are not variables and should be linked to the structure variables to make them meaningful members.
- e.g. the word ***title*** has no meaning whereas "title of book3" has a certain meaning.
- The link between a member and a variable is established using ***the member operator '.' also known as "dot operator" or "period operator"***.
- ***e.g. book1.price*** is the variable representing the price of ***book1*** and can be treated like any other ordinary variable. The syntax for assigning values to the members of ***book1*** is:

```
strcpy(book1.title, "ANSI");
strcpy(book1.author, "Dennis");
book1.pages = 150;
book1.price = 420.50;
```

- We can also use ***scanf()*** to give values as: *scanf("%s\n", book1.title);scanf("%d\n", &book1.pages);*

Structure Initialization

- Like any other data type a structure variable can be initialized at compile time. e.g.
- Assigns value 60 to ***student.weight*** and 180.75 to ***student.height***.
- ***There is one-to-one correspondence between the structure members and their initializing values.***
- A lot of variation is possible in structure initialization. ***Initializing Two structure variable requires an essential use of tag name.***
- ***Third way to initialize a structure variable is to initialize outside the function.***

```
main()
{
    struct
    {
       int weight;
       float height;
    }
    student = {60, 180.75};
    .....
    .....
    }
```

```
struct book_bank
{
    char   title[20];
    char   author[15];
    int    pages;
    float  price;
};
struct book_bank book1, book2, book3;
```

```
struct st_record
{
    int weight;
    float height;
}   student1 = {60, 180.75};
main()
{
    struct st_record student2 = {53, 170.60};
    .....
    .....
}
```

Structure Initialization contd.

- C does not permit the initialization of individual structure members within the template.
- Initialization must be done only in the declaration of the actual variables.
- Compile-time initialization of structure variable must have the following elements:

1. ***Keyword struct***
2. ***Structure tag name***
3. ***Name of the variable to be declared***
4. ***Assignment operator =***
5. ***Set of values for the members of the structure variable, separated by commas and enclosed in braces***
6. ***Terminating semicolon***

Rules for Initializing Structures

There are a few rules to keep in mind while initializing structure variables at compile-time.

1. We cannot initialize individual members inside the structure template.
2. The order of values enclosed in braces must match the order of members in the structure definition.
3. It is permitted to have a partial initialization. We can initialize only the first few members and leave the remaining blank. The uninitialized members should be only at the end of the list.
4. The uninitialized members will be assigned default values as follows:
   - Zero for integer and floating point numbers.
   - '\0' for characters and strings.

Copying and Comparing Structure Variables

- Two variables of same structure type can be copied the same way as ordinary variables.
- For example if ***person1*** and ***person2*** belong to the same structure, then the following statements are valid:

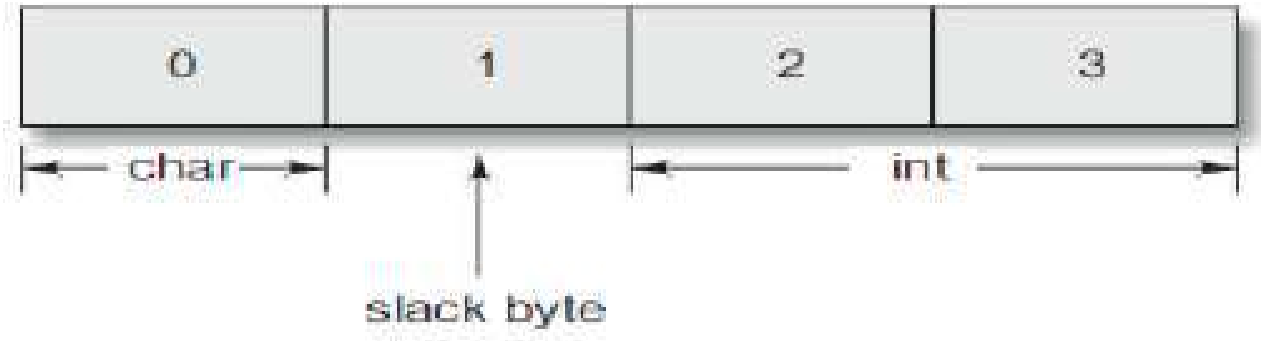
  person1 = person2; person2 = person1;

- However, the statements such as

  person1 == person2; person1 != person2;

are not allowed in C, since C does not provide any logical operations on structure variables.

- In case we need to compare the members we may do so by comparing members individually.
- Computer stores structures using the concept of "***word boundary***". The ***size of word boundary is machine dependent***.
- In a computer with ***two bytes word boundary***, the members of a structure are stored left aligned on the word boundary.
- A ***char*** data-type takes ***1 byte*** and ***int*** takes ***2 bytes***. ***1 byte*** between them is left unoccupied. This unoccupied byte is known as the ***slack byte***.



Operations on Individual Members

- ***Individual members*** identified using member operator, the ***dot***.
- A member with the ***dot operator*** along with its structure variable can be treated like any other variable name and therefore can be manipulated using expressions and operators.

if (student1.number == 111)

student1.marks += 10.00;

float sum = student1.marks + student2.marks;

student2.marks * = 0.5;

- We can also apply increment and decrement operators to numeric type members.

student1.number ++;

++ student1.number;

- ***Note: Precedence of the member operator is higher than all arithmetic and relational operators and therefore no parentheses are required.***
- Three ways to access members:

1. Using dot notation: v.x
2. Using indirection notation: (*ptr).x
3. Using selection notation: ptr->x

Array of Structures

- Structures used to describe the format of ***a number of related variables***. e.g. In analyzing marks obtained by a class of students we may use a template to describe student name and marks obtained in various subjects and then declare all the students as structure variables.
- For such cases we may declare an array of structures, with each element of the array representing a structure variable. e.g. ***struct class student[100];*** defines an array called ***student,*** that consists of 100 elements, with each element defined to be the type ***struct class.***

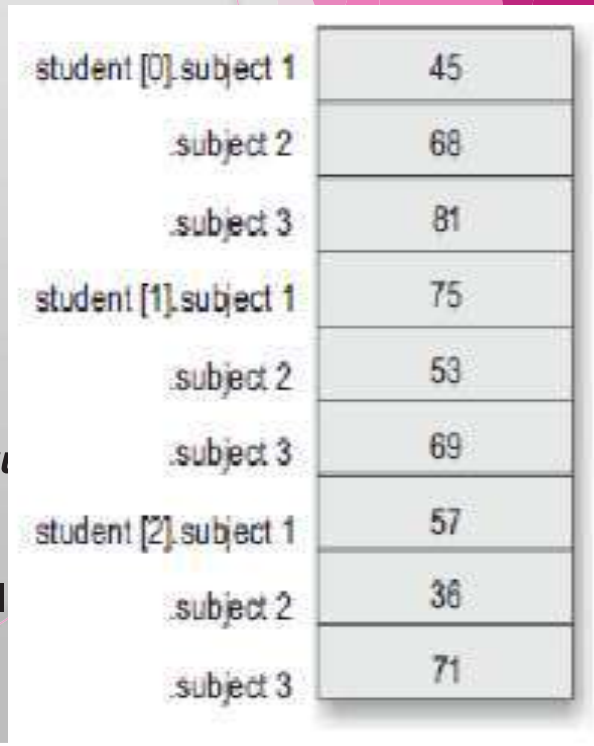
```
struct marks
{int subject1;int subject2;int subject3;};
main()
{
struct marks student[3] ={{45,68,81}, {75,53,69}, {57,36,71}};
}
```

- Declares the ***student*** as an array with elements ***student[0], student[1],*** and ***st***

with member initialization as ***student[0].subject1 = 45; student[0].subject2 = 65;***

- C permits the use of arrays as structure members. Single and Multi-dimensional

can be used inside the structure declaration as



Array Within Structures

- C permits the use of arrays as structure members.
- Single and Multi-dimensional array of ***char, int*** and ***float*** data type can be used inside the structure declaration as

```
struct marks
{
int number;
float subject[3];
} student[2];
```

- Here the member subject contains three elements, ***subject[0], subject[1],*** and ***subject[2]*** and can be accessed using appropriate subscripts.

Nested Structures

- Structure within a structure is known as nested structure. In C structure nesting is permitted. e.g.

```
struct salary
{char name;char department;int basic_pay;int dearness_allowance;
int house_rent_allowance;int city_allowance;}
employee;
```

- This structure defines name, department, basic pay and three kinds of allowances. We can group all the items related to allowance together and declare them under a ***substructure*** as

```
struct salary
{char name;char department;
struct
{int dearness;int house_rent;int city;}
allowance;}
employee;
```

- ***salary*** structure contain member named ***allowance***, which is a inner structure with three members namely ***dearness***, ***house_rent*** and ***city*** are referred as: ***employee.allowance.dearness***, ***employee.allowance.house_rent***, and ***employee.allowance.city***

Array in Nested Structures

- An inner structure can contain an array. The syntax for declaration is:

```
struct salary
{
struct
{
int dearness;
}
allowance,
arrears;
}
employee[100];
```

- Inner structure has two variables, ***allowance*** and ***arrears***, implying that both of them have same structure template.
- ***Note: The comma operator is present after allowance .***
- The base member can be accessed as: ***employee[1].allowance.dearness***

Structures and Functions

- C supports the passing of structure values as arguments to functions. Three methods by which the values of a structure can be transferred from one function to another.

1. ***1st method is to pass each member of the structure as an argument of the function call. The actual arguments are then treated independently like ordinary variables. This is the basic method and becomes unmanageable and inefficient for large structures.***
2. ***2nd method involves passing a copy of entire structure to the called function. Since the function is working on a copy of the structure, any changes to structure members within the function is not reflected in the original structure (in calling function). So it is necessary for the function to return the entire structure back to the calling function. All compilers may not support this method of passing the entire structure as a parameter.***
3. *3rd approach employs pointer concept to pass structure as argument. In this the address location of the structure is passed to called function. The function can access indirectly the entire structure and work on it. This process is same as the passing of array to function.*

- General syntax is:

function_name(structure_variable_name);

```
data_type function_name(struct_type st_name)
{
    ......
    ......
  return(expression);
}
```

Structures and Functions contd.

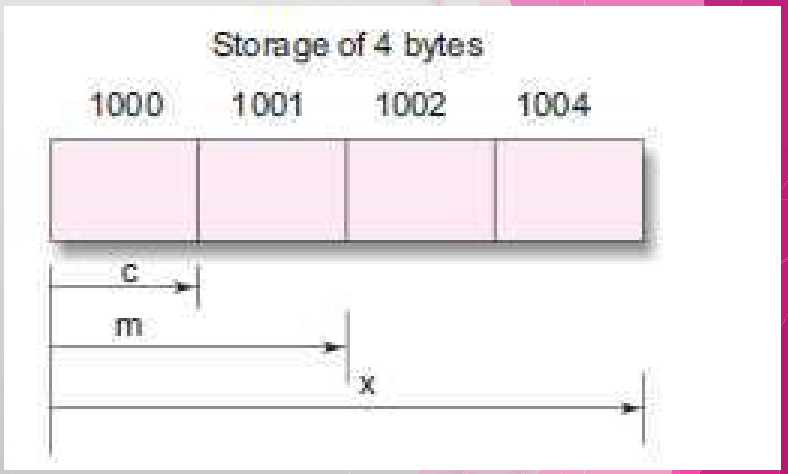
The following points are important to note:

1. The called function must be declared for its type, appropriate to the data type it is expected to return. For example, if it is returning a copy of the entire structure, then it must be declared as **struct** with an appropriate tag name.
2. The structure variable used as the actual argument and the corresponding formal argument in the called function must be of the same **struct** type.
3. The **return** statement is necessary only when the function is returning some data back to the calling function. The *expression* may be any simple variable or structure variable or an expression using simple variables.
4. When a function returns a structure, it must be assigned to a structure of identical type in the calling function.
5. The called functions must be declared in the calling function appropriately.

Unions

- ***Unions*** are concept borrowed from structures and therefore follow the same syntax as structures.
- ***Major distinction between Unions and Structures are in terms of storage.***
- In ***structures, each member has its own storage location***, whereas ***all the members of a union use the same location***.
- This implies that, although a ***union may contain many members of different types, it can handle only one member at a time***.
- ***Union*** can be declared using keyword **union** and is given as:

```
union item
{
int m;
float x;
char c;
} code;
```



- Variable ***code*** of type ***union item***. Union contains three members, each with a different data type, however we can use only one of them at a time. ***Only one location is allocated for a union variable irrespective of its size.***
- ***Note the compiler allocates a piece of storage that is large enough to hold the largest variable type in the union.***
- ***Union members can be accessed using dot operator and same as the structure member. e.g. code.m, code.x***

Module4

Functions: Scope of variables, call by value, call by reference, Recursion, Pointers

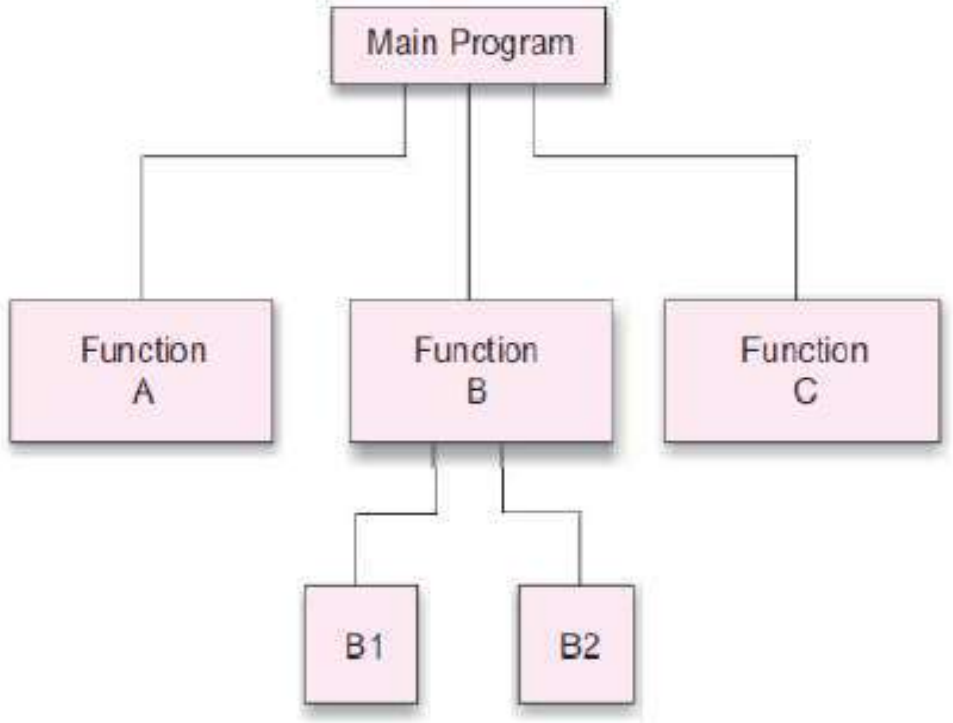
Functions

- ***C uses modular programming approach by utilizing one of its strengths which are known as the user-defined functions.***
- Function being the one of the powerful tool of the C language helps creating programs which are less complex, easily debuggable, and understandable.
- C functions are classified into two categories, namely, library functions and user-defined functions.
- ***main()*** is an example of ***user-defined functions***. ***printf()*** and ***scanf()*** *ibelong to the category of* ***library functions.*** Other mathematics and string functions such as ***sqrt()***, ***cos()***, ***strcat()*** belong to ***library functions*** category.
- ***Main distinction between*** these two categories is that **library functions are not required to be written by the end-user** whereas ***the user-defined functions*** has to be developed by the user at the time of writing a program.
- The user-defined function can also become the library function.
- ***main()*** function a specially recognized function which marks the beginning of program execution. While it is possible to code any program utilizing only main function, it leads to a number of problems.
- First program may become too large and complex.
- Second the debugging, testing, and maintaining tasks becomes difficult.
- So if a program is divided into functional parts, then each part may be independently coded and later combined into a single unit called the ***subprograms*** that are much easier to understand, debug, and test.
- Further, there are times when certain operations or calculations are repeated at many points throughout a program.

Functions contd.

- E.g. Consider the use case of a factorial of a number at several locations in the main program.
- Instead of repeating the factorial program statement at every location, we need to create a function named ***factorial()*** that can be called and used whenever required.
- Utilizing functions helps in saving both ***memory space*** and ***time***.
- The division approach clearly results in a number of advantages.

1. Facilitates top-down modular programming as shown in the figure. In this programming style, the high level logic of the overall problem is solved first while the details of each lower-level function are addressed later.
2. Length of a source program can be reduced by using functions at appropriate places. This factor is particularly critical with microcomputers with limited memory.
3. Location and Isolation of faulty function becomes easy.
4. Multiple-usage of a function. This means C programmer can build on what others have done instead of starting from beginning.



Multi-Function Program

- Function is a self-contained block of code that performs a particular task.
- Once created the function is treated as a ***black box*** that takes data from main program and returns a value.
- Inner details of operation are invisible to the rest of the program. All the program knows about a function is : what goes in and what comes out. e.g.

```
main( )
{printline( );
printf("This illustrates the use of C functions\n");
printline();}
void printline(void)
{int i;
for (i=1; i<40; i++)
printf("-");
printf("\n");}
```

- The module defines a function ***printline()*** to print a line of 39 character length. The program use main() and printline() functions.