

Infosys®

POWERED BY INTELLECT  
DRIVEN BY VALUES



## Programming Fundamentals Day 2



## Session Plan - Day 2

- Arrays
- Pointers
- Pointers and Arrays
- String Functions
- Selectional Control Structures
- Iterational Control Structures – while Loop



Today's session will focus on the need for programming and how to get started with programming.

Infosys®

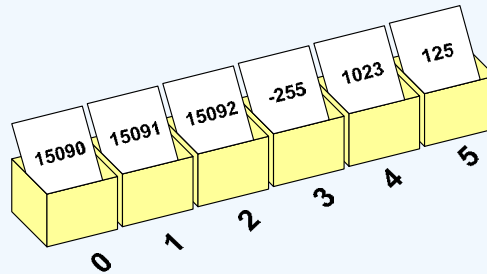
POWERED BY INTELLECT  
DRIVEN BY VALUES



## Arrays

## Arrays

- An array is a series of variables, all being same type and size
- Each variable in an array is called an *array elements*
- All the elements are of same type, but may contain different values
- The entire array is contiguously stored in memory
- The position of each array element is known as array index or subscript
- An integer array looks like this:



An array (also called a subscripted variable) is a collection of similar elements stored in adjacent memory locations.

Arrays are "a set of items which are randomly accessible by numeric index." Arrays are typically laid out in memory as a contiguous chunk of blocks, where each block is of the same data type. Because the memory is laid out as a contiguous set of memory chunks, arrays are very fast for accessing items by index. Arrays are a great choice of data structure when the number of elements it contains is known ahead of time.

An array is a fixed collection of same data-type that are stored contiguously and are accessible by an index. What this means is that an array consists of a set of physically contiguous memory locations. These memory locations can all be addressed with one name – the name of the array. Each location in the array is referred to by the subscript of the array. In C, the syntax to declare an array is as follows.

```
data-type array-name [size];
```

Arrays must be explicitly declared so that the compiler can allocate space for them in memory. Here, data-type declares the base type of the array, which is the type of each element in the array and size defines how many elements the array will hold.

e.g. `int aiArrayOfIntegers[10];`

This statement declares an integer array named `aiArrayOfIntegers`, consisting of 10 elements. Each of these 10 elements can only contain an integer value. The index of the first element of an array in C is 0. That is, the ten elements of the array can be referenced as

`aiArrayOfIntegers[0], aiArrayOfIntegers[1], aiArrayOfIntegers[2], ... , aiArrayOfIntegers[9].`

These values may be accessed as shown below.

```
iVar1 = aiArrayOfIntegers[0];
```

Note that in the declaration statement, `aiArrayOfIntegers[10]` means that there are 10 elements in the array; but in an assignment statement, `aiArrayOfIntegers[10]` refers to the 11th element of a 10-element array. That is, if there are ten elements in an array, the subscript of the last element of the array is 9, and **not 10**. This is a common programming mistake and results in some indeterminate value being returned by `aiArrayOfIntegers[10]`, which is very likely to cause program failure.

In the declaration statement, the subscript can be a named constant; however, it cannot be a variable.

The amount of storage required to hold an array is directly related to its type and size. For a single dimensional array, the total size in bytes is computed as:

`total-bytes = sizeof( data-type) * length of array.`

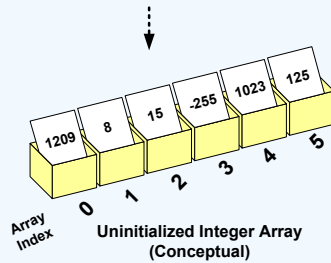
## Declaring Arrays

- In C, an array can be of any basic data type
- **Example:**  
int aiEmployeeNumbers[6];  
float afSalary[6];
- The array index starts with zero
- The valid array indexes for the above declared array is 0 to 5
- When an array is declared without initializing it, the elements have unknown (garbage) values



# Memory Representation of an Integer Array

```
/* Declare array */
int aiEmployeeNumbers [6];
```



How an integer array looks in memory?

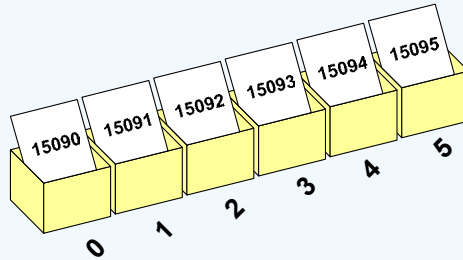
Memory Address      Contents of Memory Location

2A 3014	00	Integer (0)
2A 3015	00	
2A 3016	04	
2A 3017	B9	Integer (1)
2A 3018	00	
2A 3019	00	
2A 301A	00	Integer (2)
2A 301B	08	
2A 301C	00	
2A 301D	00	Integer (3)
2A 301E	00	
2A 301F	0F	
2A3020	80	Integer (4)
2A 3021	00	
2A 3022	00	
2A 3023	7C	Integer (5)
2A 3024	00	
2A 3025	00	
2A 3026	03	
2A 3027	FF	
2A3028	00	
2A 3029	00	
2A 302A	00	
2A 302B	7D	



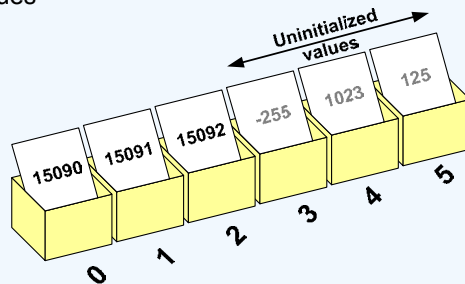
## Declaring and Initializing arrays (1 of 2)

- Arrays can be initialized as they are declared
- **Example:**  
`int aiEmployeeNumbers[ ] = {15090, 15091, 15092, 15093, 15094, 15095};`
- The size in the above case is optional and it is automatically computed
- In the above example size of the array is 6 and it occupies  $6 * 4 = 24$  bytes



## Declaring and Initializing arrays (2 of 2)

- When an array is partially initialized the remaining elements will be zero or garbage values
- **Example:**  
`int aiEmployeeNumbers[6] = {15090, 15091, 15092};`
- In the above example, the array indexes from 3 to 5 may contain zero or garbage values





## Using Array Elements

- An array can be accessed by giving the respective index

- **Syntax:**

ArrayName[index]

- **Example:**

```
float afSalaries[6];           /* Declare an array of six floats */
afSalaries[0] = 12500.00;      /* Assign a value to element at index 0 */
afSalaries[1] = 15000.00;      /* Assign a value to element at index 1 */
...
afSalaries[5] = 25000.00;      /* Assign a value to element at index 5 */

/* Print the salary at array index 0 */
printf ("Salary at Index 0 is %f \n", afSalaries[0]);
```

- Arrays can also be referenced as index[ArrayName].
- The statement 5[afSalaries] is a valid statement



## A Summary on Arrays (1 of 2)

- Before using an array, its type and size must be declared
- The first element in the array is numbered 0, so the last element is 1 less than the size of the array
- The elements of the array are always stored in contiguous memory locations
- An array can be initialized at the same place where it is declared.

**Example:** `int num[6] = {2,4,12,5,45,5}`



C has no bounds checking on arrays. One could overwrite either end of an array and write into some other variable's data or even into the program's code.

E.g.

```
int count[10],i;  
for (i=0;i<100;i++)  
count[i] = i;
```

This code will compile without error, but it is incorrect because the for loop will cause the array count to be overrun.

## A Summary on Arrays (2 of 2)

- if the array is initialized at the time of declaration, mentioning the dimension of array is optional.

**Example:** `double dNum[] = {12.3, 34.2, -23.4, -11.3};`

- If the array elements are not given any specific values, they are supposed to contain garbage values.
- In C there is no check to see if the subscript used for an array exceeds the size of the array. Data entered with a subscript exceeding the array size will simply be placed in memory outside the array. This will lead to unpredictable results, to say the least, and there will be no error messages to warn the programmer.



Infosys®

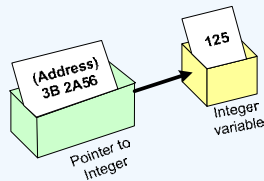
POWERED BY INTELLECT  
DRIVEN BY VALUES



## Pointers

## Pointers (1 of 4)

- A pointer is a data type which stores the address of a variable
- A variable is a *name* given to a set of memory locations allocated to it.
- For every variable there is an address, the starting address of its set of memory locations.
- If a variable called **p** holds the address of another variable **i** then **p** is called as a **pointer variable** and **p is said to point to i**.



## Pointers (2 of 4)

- To Declare a pointer variable, use the following syntax
  - `data_type *pointer_var_name;`

### Example:

1. `int *piCount;`

This declaration tells the compiler that `piCount` will be used to store the address of an integer value – in other words `piCount` points to an integer variable.

2. `float *pfBasic;`

This statement declares `pfBasic` as a pointer variable which can contain the address of a float variable

**Note:** The size of pointer variable on Windows platform is 4 bytes. However it may vary from one platform to another.



‘\*’ stands for ‘value at address’.

Thus `int *p` would mean, the value at address contained in `p` is an `int`.

## Pointers (3 of 4)

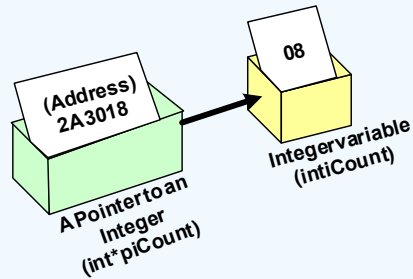
- To store the address of an integer into an integer pointer, use the following syntax

```
int_pointer = & int_var;
```

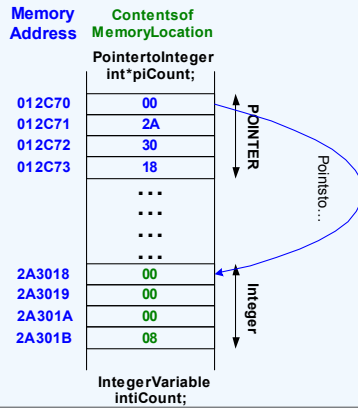
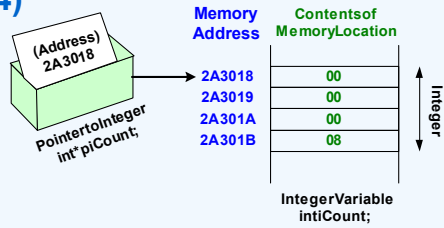
**Example:**

```
int iCount = 8;  
int *piCount;  
piCount = & iCount;
```

- **iCount**: an integer variable
- **piCount**: an integer pointer.
- **&**: the “address of” operator



## Pointers (4 of 4)





## Reading Contents of Variable using Pointers

- To access the **value at the address** stored in the pointer, use the following syntax `*pointervariable`
- Example:  
`printf("%d", *piCount);`
  - Here the `*` operator preceding `piCount` will fetch the value at the address stored in `piCount`
- Using `'=='` operator (Equal to) on pointers, will check whether both pointers being compared are pointing to the same address or not
- Uninitialized pointers may point to any memory location
- Using `*` (indirection operator) on uninitialized pointers may result in program throwing a run time error



### •Program to illustrate pointers.

```
int main(int argc, char **argv) {  
    int iCount=3;  
    int *piCount;  
    piCount=&iCount;  
    printf("\n Address of iCount = %u",&iCount);  
    printf("\n Address of iCount = %u",piCount);  
    printf("\n Value of iCount = %d",iCount);  
    printf("\n Value of iCount = %d",*piCount);  
    return 0;  
}
```

In the above program variable **iCount** is declared as a integer and **piCount** is declared as a pointer to integer. Consider the statement **piCount=&iCount**; Here the **&** operator will extract the address of **iCount** and store its value in **piCount**. Consider the statement **printf("\n Value of iCount = %d",\*piCount)**; Here the value of **iCount** is displayed using the pointer notation. The operator **\*** preceding **piCount** will retrieve the value at the address stored in **piCount**.

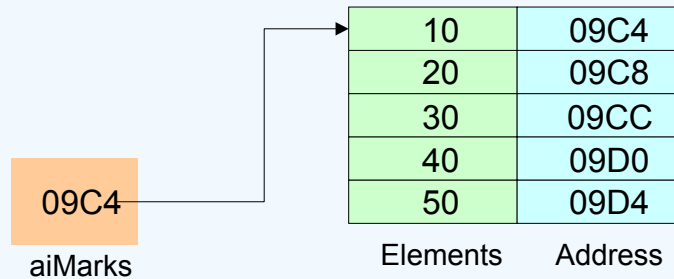
## NULL Pointers

- Using a pointer without initializing it to any data may result in data corruption or even program crash
- To differentiate between initialized and uninitialized pointers, we usually set an uninitialized pointer to NULL
- **Example:**  
/\* Initializing pointer to NULL \*/  
`int *piCount = NULL;`
- A pointer can be checked whether it is pointing to NULL using the '==' (Equal to) operator



## Pointers and Arrays (1 of 2)

- Pointers & Arrays are closely related to each other.
- The name of the array is the address of the 0<sup>th</sup> element of the array. Hence it implies that the name of the array acts like a pointer to the first element in the array
- Array elements are allocated memory in a contiguous fashion.



scanf() revisited:

Consider the example:

```
char s[10];  
printf("Enter a string");  
scanf("%s",s);
```

In the above code, we need not mention & in the scanf statement. This is because, the name of the array is nothing but the address of the first element of the array. Hence, only the name of the array variable is mentioned in the scanf() function.

## Pointers and Arrays (2 of 2)

- Arrays make use of pointers internally to navigate through the array.

```
int aiMarks[10];           /* Declare an integer array */
int *piMarks;              /* Declare an integer pointer */

aiMarks[0] = 76;           /* Initialize the array elements */
aiMarks[1] = 89;
... ..
... ..
aiMarks[9] = 90;
/* Point the pointer to the array. Array name is nothing but a pointer to the
first element in the array */
piMarks = aiMarks;
/* Accessing array elements using the pointer INSTEAD OF ARRAY NAME */
printf ("The array element at position 1 is %d \n", piMarks[1]);
```

**NOTE:** The above can also be written as `*(piMarks+1)` or `*(aiMarks+1)`.



Copyright © 2004, 20  
Infosys Technologies Ltd

ER/CORP/CRS/LA06/003  
Version no: 2.0

Infosys®

### •Accessing Array elements using a pointer

Array name gives the base address of array. if we extract this address in a pointer we can use it to access the elements of the array. Following Program illustrates this concept.

```
void main(){
    int i;
    int a[5]={1,2,3,4,5};/*array of integers */
    int *q=a;
    for(i=0;i<5;i++)      {
        printf("\n%d",*q);
        q++;
    }
}
```

In the above program the base address of the array is extracted in a pointer **q**. Each iteration of the for loop accesses the element in the array and increments **q** by 1 so that it points to the next element in the array.

## A brief recap on Pointers (1 of 2)

- A variable is declared by giving it a type and a name (e.g. **int iVar;**)
- A pointer variable is declared by giving it a type and a name (e.g. **int \*piPtr**) where the asterisk tells the compiler that the variable named **piPtr** is a pointer variable and the type tells the compiler what type of variable, the pointer will point to (integer in this case).
- Once a variable is declared, we can get its address by preceding its name with the unary **&** operator, as in **&iVar**.
- We can "dereference" a pointer, i.e. refer to the value of that which it points to, by using the unary **\*** operator as in **\*piPtr**.



## A brief recap on Pointers (2 of 2)

- NULL pointers are those pointers which contain a value NULL. It is a good practice to initialize pointers to NULL as they are declared
- Using uninitialized pointers may result in program crash
- The name of an array is nothing but a pointer to the first element of the array
  - Array elements can be accessed using a pointer by providing an index



Infosys®

POWERED BY INTELLECT  
DRIVEN BY VALUES



Strings

## Strings

- A string is a series of characters in a group that occupy contiguous memory

- **Example:**

`"My Training"`

`"Programming Fundamentals"`

- A string should always be enclosed with in double quotes ("")
- In memory, a string ends with a null character `'\0'`
- Space should be allocated to store `'\0'` as part of the string
- A null character (`\0`) occupies **1 byte** of memory



A group of characters (Alphabets, digits and special characters) is called as a **string**. For example in order to store the name of a person or to store the name of an item a string have to be used. That is any attribute that involves characters other than digits is considered as string. In computing applications where processing of textual data are involved, strings are useful in representing these textual data. When strings are written it is important that they are enclosed with in **double quotes**. If a single quote is used it is treated as **character constants**. For example 'M' is different from "M". The former one is called as a character constant and the latter is called as **string constant**.

Each and every string ends with a **null character \0**. This designates the end of the string. This `\0` is a **single character**. So when it has to be explicitly assigned to a string, it has to be written with in single quotes as `'\0'`. When the size of the string is decided, `'\0'` should also be considered. For example if the Department Name has at most 15 characters the size of the string should be 15+1 which is 16.



## Declaration of Strings (1 of 2)

```
char variablename [Number_of_characters ];
```

- **Example:**

```
char acEmployeeName[20];
```

Here 20 implies that the maximum number of characters can be 19 and one position is reserved for '\0'

Since a character occupies one byte, the above array occupies 20 bytes (19 bytes for the employee name and one byte for '\0')

```
/* Declaring a string as a character pointer */
```

```
char *pcInfy = "Infosys Tech";
```

```
/* Declaring a string as a character array */
```

```
char acInfy[ ] = "Infosys Tech";
```

- Think why '\0' requires only one byte!!



The null character '\0' has so much of importance with respect to strings. The built in string functions considers the characters in the string only till the first null character is encountered. So, if an array contains the string "Easy to\0 go\0" then, only "Easy to" is considered by the string functions.

## Declaration of Strings (2 of 2)

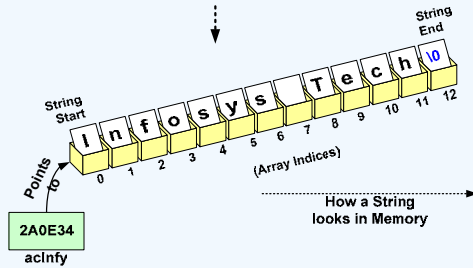
```
char acInfy[] = "Infosys Tech";
```

(OR)

```
char *pcInfy = "Infosys Tech";
```

(OR)

```
char acInfy[13] = "Infosys Tech";
```



acInfy	Memory Address	Data Decimal (Hex)	Character/ Glyph
Points to	2A0E34	073 (0x49)	I
	2A0E35	110 (0x6E)	n
	2A0E36	102 (0x66)	f
	2A0E37	111 (0x6F)	o
	2A0E38	115 (0x73)	s
	2A0E39	121 (0x79)	y
	2A0E3A	115 (0x73)	s
	2A0E3B	032 (0x20)	(Space)
	2A0E3C	084 (0x54)	T
	2A0E3D	101 (0x65)	e
	2A0E3E	099 (0x63)	c
	2A0E3F	104 (0x68)	h
	2A0E40	000 (0x00)	\0



## Storage of strings in memory

```
char acItemCategory[15]= "Books";
```

acItemCategory

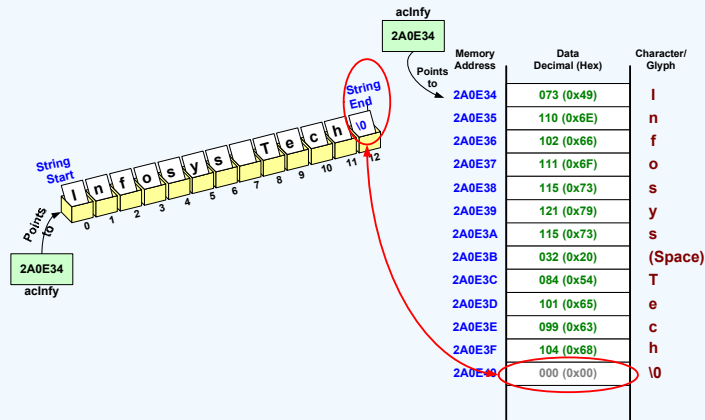
66 (B)	8000
111 (o)	8001
111 (o)	8002
107 (k)	8003
115 (s)	8004
0 (\0)	8005
Garbage value	8006
Garbage value	8007
Garbage value	8008
Garbage value	8009
Garbage value	800A
Garbage value	800B
Garbage value	800C
Garbage value	800D
Garbage value	800E



## Null Terminators in Strings

- A null character marks the end of the string

Character	ASCII Value	In Code
Null character	0 (Hex: 0)	'\0'
0 (Zero)	48 (Hex:30)	'0'



## Static initialization of Strings

- `char acltemCategory[15]="Greeting Cards";`

In the above declaration, the size of the array is specified according to the number of characters in the string. One extra space for '\0'

- `char acltemCategory[15] ="Ornaments" ;`

Here the size is more than the number of characters which is valid.

- `char acltemCategory[] ="Groceries";`

Here the size of the array is computed automatically which is 10. Total number of characters in the string is 9 and 1 for '\0';

- `char acltemCategory[3]="Books";`

Here the size specified is 3. But the number of characters in the string is 5. This is invalid.

- `char acltemCategory[] ={ 's','t','a','t','i','o','n','a','r','y','\0'};`

Here the character constants are supplied to initialize the string.



The arrays can be initialized to string constants only during **declaration**. It is invalid to write the assignment `acltemCategory="Fruits"`; To assign the value to the string, string function `strcpy` should be used. This will be dealt in next session.

## Printing Strings to console

- **Using Character pointer:**

```
char *pcProg = "Programming Fundamentals";  
printf(pcProg);  
printf("This is %s course",pcProg);
```

- **Using Character Array:**

```
char acProg[ ] = "Programming Fundamentals";  
printf(acProg);
```

**OR**

```
printf("%s", acProg );  
printf("This is %s course",acProg);
```

- **Printing a string as part of a formatted string**

```
int iCourseId = 27;  
char *pcProg = "Programming Fundamentals";  
/* print the courseId (Int) and Course name (String – char*) */  
printf("The Id of this course is %d and this course is %s \n",  
       iCourseId, pcProg);
```



## strlen() Function (1 of 2)

- **strlen()** function is used to count the number of characters in the string
- Counts all the characters excluding the null character '\0'

- Syntax

**size\_t strlen (const char\* s);**

Here **const char \*s** can be a string constant or a character pointer or a character array and **size\_t** can be interpreted as unsigned int

- Example:

```
strlen("Programming Fundamentals"); returns 24
```

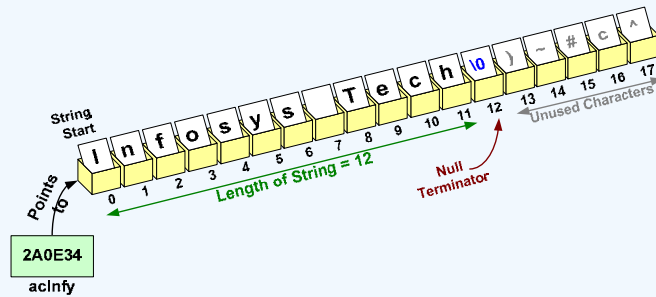
```
strlen(acItemCategory) ;
```

returns the number of characters in the character array 'acItemCategory'



## strlen() Function (2 of 2)

- An array can be of a bigger size than its length.
- The end of string is marked by the **null** character.
- The memory locations beyond the **null** character are unused and may contain unknown (garbage) values.





## A brief recap on Strings

- Strings are a sequence of characters in a group
- In C, strings are stored in character arrays
- The end of string is marked by the null terminator '\0'(ASCII code 0)
- Size of string can be smaller than the array size
- Strings can be printed to console using printf function
- Length of string refers to the size of string without the null terminator considered



## stdin, stdout and stderr (1 of 2)

- Most operating systems use the concept of streams to channel data into or from files
- A stream is a sequence of bytes of data
- A sequence of bytes flowing into a program is an input stream
- A sequence of bytes flowing out of a program is an output stream
- The major advantage of streams is that input/output programming is device independent
- A stream can be viewed as a communication channel between
  - Two computer programs
  - Between a computer program and the operating system
  - Between a computer program and the file system
- The three streams are called:
  - **stdin**: Standard Input
  - **stdout**: Standard Output
  - **stderr**: Standard Error

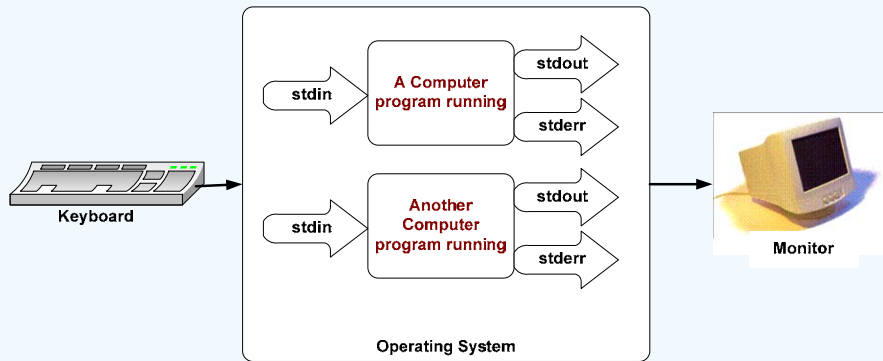


stdin, stdout and stderr are standard streams (stream pointers) for input, output, and error output.

By default, standard input is read from the keyboard, while standard output and standard error are printed to the screen.

You can think of a stream as a “smart file” that acts as a source and destination for bytes.

## stdin, stdout and stderr (2of 2)



## Reading Keyboard Input – scanf function (1 of 2)

- 'scanf', which uses a similar conversion specifier like 'printf' to read different types of data from the keyboard
- The parameters passed to the function 'scanf' after the format are pointers (address of the variables) to the data types being read from keyboard

- **Syntax:**

```
int scanf(char* format, ...);
```

- **Example:**

```
/* To read an integer */  
printf("Enter a number: ");  
scanf("%d", &iCount);
```

```
/* To read a float */  
printf("Enter Amount: ");  
scanf("%f", &fAmount);
```

Common programming error:

Forgetting '&' before the variable name



The scanf statement is used for accepting user input.

In scanf the variables are pointers that indicate the address of data items in memory. More about pointers in Day3.

Consider `scanf("%d", &iNum);`

above statement implies that one integer should be read from the keyboard and should be stored at the address of variable iNum.

A standard library function used often in conjunction with scanf is the `fflush(stdin);` function. This function is responsible for clearing the input buffer before / after any input operation.

If there is no address operator (&) specified the compiler does not generate any compile time error. That means the program can compile successfully. It may just generate only warnings depending on the compiler. But when the user gives the data during run time the data does not get stored because there is no address specified.

Common Programming Errors:

- (1) Forgetting & along with the variable name.
- (2) The number of conversion specifiers may not match the number of variables

Good programming practice: Use a printf statement before the scanf statement that displays a message what the user is to enter. This is called interactive programming.

## Reading Keyboard Input – scanf function (2 of 2)

*/\* Reading multiple input \*/*

```
short iDay,iMonth,iYear;  
printf("Enter day, month and year in format (dd mm yyyy)");  
scanf("%d %d %d",&iDay,&iMonth,&iYear);
```

*/\* Reading character input \*/*

```
char cChoice;  
printf("Want to continue ");  
scanf("%c",&cChoice);
```

*/\* Reading the value through the pointer \*/*

```
int iNumber, *piPointer;  
piPointer = &iNumber;  
scanf("%d",piPointer);
```



## Input of strings – scanf and gets functions

```
scanf("%s", acItemCategory );  
scanf("%s", &acItemCategory[0] );
```

Both the input functions are valid. The first one passes the base address (Address of the first element) implicitly.

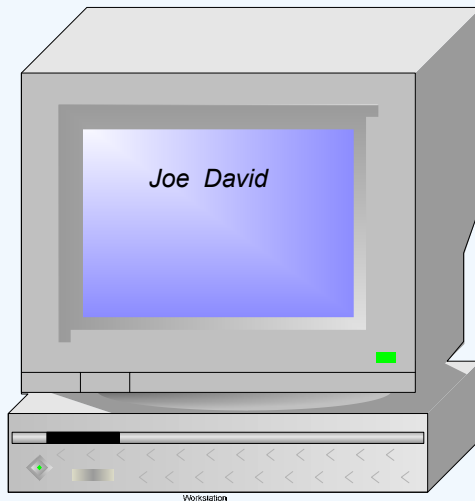
The second function passes the address of the first element explicitly

```
gets(acItemCategory) ;  
gets(&acItemCategory[0]) ;
```

This is an unformatted function to read strings



## Difference between scanf() and gets()



```
scanf("%s", acStudentName);  
char acStudentName[10];
```

acStudentName	
J	6750
o	6751
e	6752
\0	6753
D	6754
a	6755
v	6756
i	6757
d	6758
\0	6758

```
gets(acStudentName);
```



## Reading and Printing Strings - Example

```
/*Program to accept Item Category and display*/  
int main(int argc, char **argv){  
    char acItemCategory[15];  
    printf("Enter the category code ");  
    scanf("%s",acItemCategory);  
    printf("The given Item Category is %s",  
        acItemCategory);  
    return 0;  
}
```





## String Handling functions

- The following are the string functions that are supported by C

<code>strlen()</code>	<code>strcpy()</code>	<code>strcat()</code>	<code>strcmp()</code>
<code>strncat()</code>	<code>strncpy()</code>	<code>strncat()</code>	<code>strncmp()</code>

- These functions are defined in *string.h* header file
- All these functions take either a character pointer or a character array as an argument



## strcpy() Function

- **strcpy()** function is used to copy one string to another
- **Syntax:**  
strcpy (Dest\_String , Source\_String);
- Here Dest\_string should always be variable
- Source\_String can be a variable or a string constant
- The previous contents of Dest\_String, if any, will be over written

- **Example:**  
char acCourseName[40];  
**strcpy(acCourseName , "C Programming");**  
Now *acCourseName* will get the value "C Programming"



## strcat() Function

- **strcat()** function is used to concatenate (Combine) two strings
- Syntax  
**strcat( Dest\_String\_Variable , Source\_String ) ;**
- In this, the Destination should be a variable and Source\_String can either be a string constant or a variable.
- The contents of Dest\_String is concatenated with Source\_String contents and the resultant string is stored into Dest\_String variable.

- **Example:**

```
char acTraineeFpCourse [50] = "The course is ";
```

```
strcat(acTraineeFpCourse,"Oracle 8i");
```

The resultant string in acTraineeFPCourse will be

"The course is Oracle 8i"



## strcmp() Function

- **strcmp()** function is used to compare two strings
- This is case sensitive
- **Syntax:**  
`int strcmp( String1 , String2 )`
- Here both String1 and String2 can either be a variable or a string constant
- **strcmp()** function returns an integer value.
- If strings are equal it returns **zero**
- If the first string is alphabetically greater than the second string then it returns a **positive value**
- If the first string is alphabetically less than the second string then it returns a **negative value**
- **Example:**  
strcmp("My Work", "My Job") returns a **positive value**



## strcpy() and strcat() - Example

```
/*Program to show the usage of strcpy() and strcat()
functions*/
#define DEPARTMENTNAMESIZE 10
int main(int argc, char **argv) {
    /*Array Declaration*/
    char acDepartmentName[DEPARTMENTNAMESIZE];
    /*Copy the string into the array*/
    strcpy(acDepartmentName, "Infy-");
    /*Concatenate the string with the department name
    supplied*/
    strcat(acDepartmentName, "HRD");
    printf("The department name is
           %s\n\n", acDepartmentName);
    return 0;
}
```



## strcmpi() Function

- **strcmpi()** function is same as strcmp() function but it is case insensitive
- strcmpi("My Work", "My work") returns **zero**



## Other String Functions (1 of 2)

- **strncpy()** is used to copy only the left most n characters from source to destination
- Syntax:  
**strncpy( Destination\_String , Source\_String, int no\_of\_characters )**  
Here only n characters from Source\_String is copied to Destination\_String
- **strncat()** is used to concatenate only left most n-characters from source with destination
- Syntax:  
**strncat( Destination\_String, Source\_String, int no\_of\_characters )**



## Other String Functions (2 of 2)

- **strncmp()** is used to compare only left most 'n' characters from the strings

Syntax:

**strncmp( char string1, char string2, int no\_of\_characters );**





## sscanf() Function

- sscanf() is used to read formatted data from a string

### Syntax:

```
sscanf (CharArray, "Conversion Specifier", addr_variables);
```

- This will extract the data from the character array according to the conversion specifier and store into the respective variables

### Example:

```
sscanf (acBuffer, "%s %d", acName, &iAge);
```

Now the strings are extracted from `acBuffer` and stored into `acName` and `iAge`.



sscanf() is used to extract the strings from the given string. In the above example if the array `acBuffer` contains "Age 25" then after execution of the function, `acName` will be "Age" and `iAge` will be 25. sscanf() will read subsequent characters until a whitespace is found (whitespace characters are blank, newline and tab).

## sscanf() Function - Example

```
/*Program to show the usage of sscanf() function*/
#define NAMESIZE 20
int main(int argc, char **argv) {
    char acPersonRecord[]="Williams 18-JAN-1982 35000";
    char acName[NAMESIZE];
    char acDateOfBirth[11];
    double dSalary;
    sscanf(acPersonRecord,"%s%s%lf",
           acName,acDateOfBirth,&dSalary);
    printf("Name %s\n",acName);
    printf("Date Of Birth %s\n",acDateOfBirth);
    printf("Salary %lf\n\n",dSalary);

    return 0;
}
```



## sprintf() Function

- This function is similar in behavior to 'printf', but instead of writing to screen, it writes the formatted text to a character array (buffer).

### Syntax:

```
sprintf (CharArray, "Conversion Specifier", variables);
```

- This function is very useful for formatting strings without displaying them.

- **Example:**

```
int iEmpNo = 2048;
```

```
char *acName = "Thomas Anderson";
```

```
char acBuffer[100];
```

```
/* acBuffer will contain "Name: Thomas Anderson, EmpNo: 2048" after this  
sprintf call */
```

```
sprintf (acBuffer, "Name: %s, EmpNo: %d", acName, iEmpNo);
```



## A brief recap on String library functions

- `strlen()` : Finding the length of the string
- `strcpy()` : Copying a string
- `strncpy()` : Copying first 'n' bytes of the string
- `strcmp()` : Comparing two strings (case sensitive search)
- `strcmapi()` : Case insensitive search
- `strncmp()` : Comparing only first 'n' characters of the string (case sensitive search)
- `strncmpi()` : Comparing only first 'n' characters of the string (case insensitive search)
- `strcat()` : Concatenating strings
- `sprintf()` : Writing formatted text to a character array



Infosys®

POWERED BY INTELLECT  
DRIVEN BY VALUES



## Control Structures

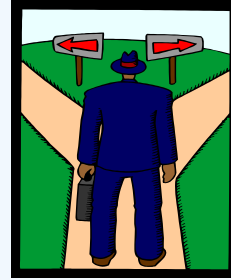
## Control Structures

- A control structure refers to the way in which the Programmer specifies the order of executing the statements
- The following approaches can be chosen depending on the problem statement:
  - Sequential
    - In a sequential approach, all the statements are executed in the same order as it is written
  - Selectional
    - In a selectional approach, based on some conditions, different set of statements are executed
  - Iterational (Repetition)
    - In an iterational approach certain statements are executed repeatedly



## Selectional Control Structures

- There are two selectional control structures
  - If statement
  - Switch statement



Control statements specify the order of the execution of the statements in the program.

Control statements are of 2 types: Selectional and Iterational (Iterational constructs are discussed later)

Selectional statements are of 2 types: unconditional control statements and conditional control statement

(1) unconditional control statement:

goto statement: This statement is used to transfer the control to another part of the program without checking for any conditions.

This is an unstructured way of programming and is recommended not to use.

Program written using goto is difficult to indent. Indentation shows the logical structure of the program and using goto in programs makes the indentation impossible.

Also, goto statements defeats compiler optimizations. Some optimizations depend on a program's flow of control. An unconditional goto makes the flow harder to analyze and reduces the ability of compiler optimizations.

(2) conditional control statements:

if..else statement: This statement is used to test a condition and then take one of the two possible actions.

Syntax:

```
if (condition) {  
    statement/s to be executed when condition evaluates to true;  
}  
else {  
    statement/s to be executed when condition evaluates to false;  
}
```

If a condition is true, then the statement/s below the if are executed and when the condition evaluates to false, the statement/s below the else are executed.

The if..else statements can also be nested.

## Simple if Statement

- In a simple 'if' statement, a condition is tested
- If the condition is true, a set of statements are executed
- If the condition is false, the statements are not executed and the program control goes to the next statement that immediately follows if block

Syntax:

```
if (condition) {  
    Statement(s);  
}  
Next Statement;
```

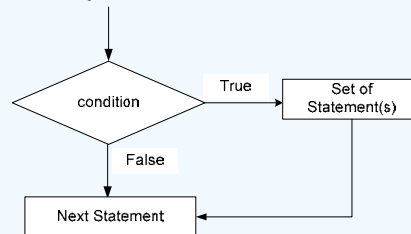
```
if (iDuration >= 3) {
```

```
    /* Interest for deposits equal to or more than 3 years
```

```
    is 6.0% */
```

```
    fRateOfInterest = 6.0;
```

```
}
```



Ex: what does the following code snippet do?

```
if (iNum1 == iNum2 )  
    printf("The two numbers are equal");  
else  
    if (iNum1 > iNum2)  
        printf("iNum1 is greater than iNum2");  
    else  
        printf("iNum2 is greater than iNum1");
```

The if statement controls conditional branching. The body of an if statement is executed when the value of the expression is nonzero.

Stmt1 is executed if the condition evaluates to true, else Stmt2 is executed. Note that semicolon is placed at the end of statements forming the if and else clauses. If the number of statements to be executed is more than one then the statements need to be enclosed in a block.

```
if (expr) {  
    Stmt1; Stmt2; Stmt3;  
}
```

For each else there needs to be a matching if.

The compiler resolves this by associating each else with the closed if that lacks else. If constructs can also be nested.

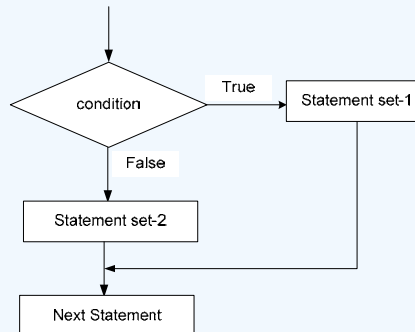


## else Statement (1 of 2)

- In simple 'if' statement, when the condition is true, a set of statements are executed. But when it is false, there is no alternate set of statements
- The statement 'else' provides the same

- **Syntax:**

```
if (condition) {  
    Statement set-1;  
}  
else {  
    Statement set-2;  
}  
Next Statement;
```



Some examples :-

```
int iN1,iN2,iN3 ,iN =0;
```

```
iN1=1,iN2=2,iN3=3;
```

1) if(iN > iN2)

```
    printf("iN1 is larger");
```

else

```
    printf("iN2 is larger");
```

2) if(iN1 > iN2)

```
    printf("iN1 is larger");
```

else if(iN1 < iN2)

```
        printf("iN2 is larger");
```

else

```
        printf("Both iN1 and iN2 are equal");
```

## else Statement (2 of 2)

- **Example:**

```
if (iDuration >= 3) {  
    /* If duration is equal to or more than 3 years,  
    Interest rate is 6.0 */  
    fRateOfInterest = 6.0;  
}  
else {  
    /* else, Interest rate is 5.5 */  
    fRateOfInterest = 5.5;  
}
```



```
3) if(iN==1)                                     // prints iN  
is not one  
    printf("iN is one");  
else  
    printf("iN is not one");
```

```
4) if(iN=1)                                     // prints  
iN1 is one  
    printf("iN is one");  
else  
    printf("iN1 is not-one");
```

In the above example the expr that's evaluated is an assignment expr and not a conditional expr. (becoz of = and not ==). The expr will assign a non-zero value to iN and then iN is evaluated. Since it contains a non-zero value (indicating true evaluation) the if part is executed

```
5) if(iN1=0)                                     // prints  
iN1 is non-zero  
    printf("iN1 is zero");  
else  
    printf("iN1 is non-zero");
```

In the above example the expr that's evaluated is an assignment expr and not a conditional expr. (becoz of = and not ==). The expr will assign a zero value to iN1 and then iN1 is evaluated. Since it contains a zero value (indicating false evaluation) the else part is executed

```
6) if(5)                                         //prints  
true.  
    printf("True");                             //becoz expr evaluates to a  
non-zero value.  
else  
    printf("false");
```

## else if Statement (1 of 2)

- The 'else if' statement is to check for a sequence of conditions
- When one condition is false, it checks for the next condition and so on
- When all the conditions are false the 'else' block is executed
- The statements in that conditional block are executed and the other 'if' statements are skipped

• **Syntax:**



```
if (condition-1) {  
    Statement set-1;  
}  
else if (condition-2) {  
    Statement set-2;  
}  
.....  
.....  
else if (condition-n) {  
    {  
        Statement set-n;  
    }  
}  
else {  
    Statement set-x;  
}  
Next Statement;
```



```
7) if(0)                                //prints false  
    printf("True");    // becoz the expr evaluates to zero value  
else  
    printf("False");  
8) if(iN==3);                          //gives a compilation as if terminates after expr.  
    printf("true");                    // this means there is no matching if for the  
else below                             // hence a error at compilation time is  
reported  
    printf("false");  
9) if(iN1==1 && iN2 <3)                  //prints true as the expr evaluates to true – T && T  
    printf("True");  
else  
    printf("false");  
10) if(!4)                             // not of a non zero value is zero hence  
false is printed  
    printf("true");  
else  
    printf("false");
```

## else if Statement (2 of 2)

- Always use 'else if' when a sequence of conditions has to be tested.
- Using only 'if' will make the compiler to check all the conditions. This increases the execution time

 Amateur code (Using only 'if')	 Professional code (Using 'else if')
<pre>if (iDuration &gt;= 6) {     fRateOfInterest = 6.5; } if ((iDuration &gt;= 3) &amp;&amp;     (iDuration &lt; 6)) {     fRateOfInterest = 6.0; } if (iDuration == 2) {     fRateOfInterest = 5.5; } if (iDuration == 1) {     fRateOfInterest = 5.0; }</pre>	<pre>if (iDuration &gt;= 6) {     fRateOfInterest = 6.5; } else if ((iDuration &gt;= 3) &amp;&amp;         (iDuration &lt; 6)) {     fRateOfInterest = 6.0; } else if (iDuration == 2) {     fRateOfInterest = 5.5; } else {     fRateOfInterest = 5.0; }</pre>



Guidelines while writing if statements:

- (1) Write the nominal path first then write the exceptions- put the nominal case in the if part rather than in the else part.
- (2) Make sure that you branch correctly on equality- using > instead of >= makes an off-by-one error.
- (3) Follow the if clause with a meaningful statement - Do not use if statement as shown below:

```
Ex: if (somecondition)      ;  
    else  
    /* do something*/
```

Instead use:

```
if ( ! Somecondition)  
    /* do something */  
/*****
```

FileName: enrpass\_fail.c

Author: E&R Department, Infosys Technologies Limited

Description: This file is a demo C program to find  
the average marks scored by a student in 3 subjects and  
display whether has passed or failed.  
average passing marks is 65.

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

## Assignment(=) vs. Equality Operator (==) (1 of 3)

- The operator '=' is used for assignment purposes whereas the operator '==' is used to check for equality
- It is a common mistake to use '=' instead of '==' to check for equality
- The compiler does not generate any error message

- **Example:**

```
if (fRateOfInterest = 6.5) {
    printf("Minimum Duration of deposit: 6 years");
}
else if (fRateOfInterest = 6.0) {
    printf("Minimum Duration of deposit: 3 years");
}
else {
    printf("No such interest rate is offered");
}
```

The output of the above program will be

**"Minimum Duration of deposit: 6 years"**

irrespective of any input for 'fRateOfInterest'



Copyright © 2004,  
Infosys Technologies Ltd

61

ER/CORP/CRS/LA06/003  
Version no: 2.0

Infosys®

```
#include <stdio.h>
```

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

```
Function main
```

```
PARAMETERS:
```

```
int argc - Number of command line parameters
```

```
char ** argv - Command line parameters in a pointer array
```

```
Entry point to the program.
```

```
Finds the average marks scored by a student and find whether  
he has passed or failed
```

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

```
int main (int argc, char** argv) {
```

```
/*declaration of variables */
```

```
float fMark1, fMark2, fMark3, fSum, fAvg;
```

```
/* Accepting the marks scored by the student in 3 subjects */
```

```
/* Display a message before accepting the marks*/
```

```
printf("Enter the marks scored by the student in 3 subjects\n");
```

```
scanf("%f%f%f",&fMark1,&fMark2,&fMark3);
```

```
/* calculating the average marks */
```

```
fSum=fMark1+fMark2+fMark3;
```

```
fAvg=fSum/3;
```

```
/* compare the average with 65 and decide whether student has passed or failed */
```

```
if ( fAvg >= 65.0)
```

```
    printf("PASS");
```

```
else
```

```
    printf("FAIL");
```

```
return 0;
```

```
}
```

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

```
End of enrpas_fail.c
```

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

## Assignment(=) vs. Equality Operator (==) (2 of 3)

- To overcome the problem, when constants are compared with variables for equality, write the constant on the left hand side of the equality symbol
- **Example:**

```
if (6.5 = fRateOfInterest) {  
    printf("Minimum Duration of deposit: 6 years");  
}  
else if (6.0 = fRateOfInterest) {  
    printf("Minimum Duration of deposit: 3 years");  
}  
else {  
    printf("No such interest rate is offered");  
}
```
- When the above code is compiled it generates **compilation errors** because the variable's value is being assigned to a constant
- This helps in trapping the error at compile time itself, even before it goes to unit testing



## Assignment(=) vs. Equality Operator (==) (3 of 3)

- **Corrected Code:**

```
if (6.5 == fRateOfInterest) {  
    printf("Minimum Duration of deposit: 6 years");  
}  
else if (6.0 == fRateOfInterest) {  
    printf("Minimum Duration of deposit: 3 years");  
}  
else {  
    printf("No such interest rate is offered");  
}
```



## Nested if Statement

- An 'if' statement embedded within another 'if' statement is called as **nested 'if'**
- **Example:**

```
if (iDuration > 6 ) {  
    if (dPrincipalAmount > 25000) {  
        printf("Your percentage of incentive is 4%");  
    }  
    else {  
        printf("Your percentage of incentive is 2%");  
    }  
else {  
    printf("No incentive");  
}
```





## What is the output of the following code snippet?

```
iResult = iNum % 2;  
if ( iResult = 0) {  
    printf("The number is even");  
}  
else {  
    printf("The number is odd");  
}
```

CASE 1: When iNum is 11

CASE 2: When iNum is 8

The output is  
"The number is odd"

The output is  
"The number is odd"

WHY???



Guidelines while writing if statements:

- (1) Write the nominal path first then write the exceptions- put the nominal case in the if part rather than in the else part.
- (2) Make sure that you branch correctly on equality- using > instead of >= makes an off-by-one error.
- (3) Follow the if clause with a meaningful statement - Do not use if statement as shown below:

Ex: if (somecondition)

;

else

/\* do something\*/

Instead use:

if ( ! Somecondition)

/\* do something \*/

## Switch case Statement

- The 'switch' statement is a selectional control structure that selects a choice from the set of available choices
- It is very similar to 'if' statement
- But 'switch' statement cannot replace 'if' statement in all situations

- **Syntax:**

```
Switch(integer variable or integer expression or character variable) {  
    case integer or character constant-1 : Statement(s);  
                                         break;  
    case integer or character constant-2 : Statement(s);  
                                         break;  
    .....  
    .....  
    case integer or character constant-n : Statement(s);  
                                         break;  
    default:                             Statement(s);  
                                         break;  
}
```



The switch statement is a multi directional conditional control statement.

Whenever a choice is to be made among a number of alternatives, we use switch statement.

The switch block is used as an alternative to if construct when the value in a variable needs to be matched against a set to values.

The expr needs to be of integral type only. All case values need to be unique.

The switch-case help control complex conditional and branching operations. The expr value is matched against all case values. If a match is found the case block executes and the control comes out of the switch (becoz of a break). In absence of a break statement the execution continues with the next case block till it encounters a break / the switch block terminates (which ever is earlier).

If expr does not match with any of the listed case values then the default block executes. The default is an optional statement and it need not come at the end; it can appear anywhere in the body of the switch statement.

This is one of the code tuning techniques in which we place cases according to the frequency which they occur in . i.e if we know that the user is going to give the options that is invalid most of the times we can place the default as the first case so that the first case statement is satisfied and executed , instead of checking all the cases and then executing the default.

According to ANSI at least 257 case labels are allowed in a switch statement.

The break statement has two uses:

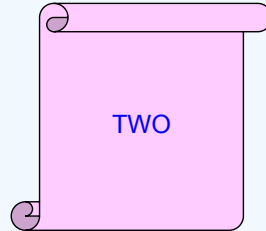
To terminate the case in the switch statement (refer to switch slides)

To force the termination of a loop. When a break statement is encountered in a loop, the loop terminates immediately and the execution resumes the next statement following the loop.

## What is the output of the following code snippet?

```
int iNum = 2;

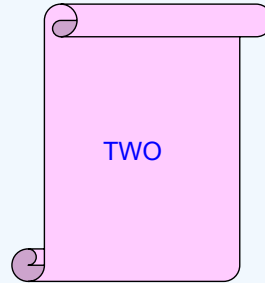
switch(iNum){
    case 1:
        printf("ONE");
        break;
    case 2:
        printf("TWO");
        break;
    case 3:
        printf("THREE");
        break;
    default:
        printf("INVALID");
        break;
}
```



## What is the output of the following code snippet?

```
int iNum = 2;

switch(iNum) {
    default:
        printf("INVALID");
    case 1:
        printf("ONE");
    case 2:
        printf("TWO");
        break;
    case 3:
        printf("THREE");
}
```



## What is the output of the following code snippet?

```
switch (iDepartmentCode){  
    case 110 :  
        printf("HRD");  
    case 115 :  
        printf("IVS");  
  
    case 125 :  
        printf("E&R");  
  
    case 135 :  
        printf("CCD");  
  
}
```

- Assume iDepartmentCode is 115 and find the output



## What is the output of the following code snippet?

```
int iNum = 2;
switch(iNum) {
    case 1.5:
        printf("ONE AND HALF");
        break;
    case 2:
        printf("TWO");
    case 'A' :
        printf("A character");
}
```

Case 1.5: this is invalid  
because the values in  
case statements must be  
integers



## What is the output of the following code snippet?

```
unsigned int iCountOfItems = 5;
switch (iCountOfItems) {
    case iCountOfItems >= 10 :
        printf("Enough Stock" );
        break;

    default :
        printf("Not enough stock");
        break;
}
```

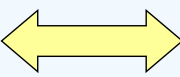
**Error:** Relational Expressions cannot be used in switch statement



## An Example – switch case

```
char ch='a';
switch(ch) {
    case 'a' : printf("Vowel");
                break;
    case 'e' : printf("Vowel");
                break;
    case 'i' : printf("Vowel");
                break;
    case 'o' : printf("Vowel");
                break;
    case 'u' : printf ("Vowel");
                break;
    default : printf("Not a vowel");
}

char ch='a';
switch(ch) {
    case 'a' :
    case 'e' :
    case 'i' :
    case 'o' :
    case 'u' :
        printf("Vowel");
        break;
    default :
        printf("Not a
                vowel");
}
```



Guidelines for writing switch case statements:

- (1) Order the cases alphabetically or numerically – improves readability
- (2) Put the normal cases first – put the exceptional cases later
- (3) Order cases by frequency- put the most frequently executed cases first and the least frequently used cases later
- (4) Use default case to detect errors and unexpected cases



## Iterational Control Structures

- Iterational (repetitive) control structures are used to repeat certain statements for a specified number of times
- The statements are executed as long as the condition is true
- These kind of control structures are also called as **loop control structures**
- are three kinds of loop control structures:
  - while
  - do while
  - for



## while Loop Control Structure

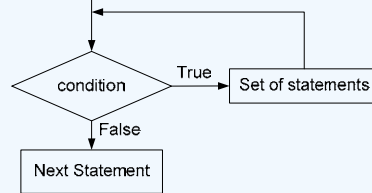
- A 'while' loop is used to repeat certain statements as long as the condition is true
- When the condition becomes false, the 'while' loop is quitted
- This loop control structure is called as an **entry-controlled** loop because, only when the condition is true, are the statements executed

- **Syntax:**

```
while (condition) {  
    Set of statements;  
}  
Next Statement;
```

- **Example:**

```
unsigned int iCount = 1;  
while (iCount <= 3) {  
    printf("%d\n",iCount);  
}
```



The while statement lets you repeat a statement until a specified expression becomes false.

The *expression* must have arithmetic or pointer type. Execution proceeds as follows:

1. The *expression* is evaluated.
2. If *expression* is initially false, the body of the while statement is never executed, and control passes from the while statement to the next statement in the program.
3. If *expression* is true (nonzero), the body of the statement is executed and the process is repeated beginning at step 1.

The while statement can also terminate when a break, goto, or return within the statement body is executed. Use the continue statement to terminate an iteration without exiting the while loop. The continue statement passes control to the next iteration of the while statement. The following is an example of the while statement:

```
while ( i >= 0 ) {  
    sum = sum + i;  
    i = i -1;  
}
```

## What is the output of the following code snippet?

```
unsigned int iCount=3;
while (iCount) {
    printf("%u\n",iCount);
    icount++;
}
```



Continued from previous slide...

This example adds the value of i to sum if i is greater than or equal to 0 and then decrements i. When i reaches or falls below 0, execution of the while statement terminates. The above example 1.4 can also be written using the while loop.

```
while(2) {    - creates a infinite loop
}
while (true)  { - creates a infinite loop
}
while(expr);  - create a infinite loop
{
}
```

## Example – while loop

```
while( (cChoice == 'Y') || (cChoice == 'y') ) {

    printf("Enter the marks scored by the student in 3
    subjects\n");
    scanf("%f%f%f",&fMark1,&fMark2,&fMark3);

    fSum=fMark1+fMark2+fMark3;
    fAvg=fSum/3;

    if ( fAvg >= 65.0)
        printf("Student PASSES\n");
    else
        printf("Student FAILS\n");

    printf("Do you wish to continue? Enter Y/N or y/n  :");
    fflush(stdin);
    scanf("%c",&cChoice);
}
```



Copyright © 2004,  
Infosys Technologies Ltd

76

ER/CORP/CRS/LA06/003  
Version no: 2.0

Infosys®

/\*\*\*\*\*

FileName: enrwhile\_demo.c

Author: E&R Department, Infosys Technologies

Limited

Description: This file is a demo C program to find  
the average marks scored for unknown number of students in 3 subjects and  
display whether has has passed or failed.  
average passing marks is 65.

\*\*\*\*\*/

#include <stdio.h>

/\*\*\*\*\*

\* Function main  
\* PARAMETERS:  
\* int argc - Number of command line parameters  
\* char \*\* argv - Command line parameters in a pointer array  
\* Entry point to the program.  
\* Finds the average marks scored by a student and find whether he has passed or failed

\*\*\*\*\*/

int main (int argc, char\*\* argv) {

/\*declaration of variables \*/

float fMark1, fMark2, fMark3,fSum,fAvg;

char cChoice = 'y';

## Example – while loop

- Refer to Notes Page



```
while( (cChoice == 'Y') || (cChoice == 'y') ) {
    /* Accepting the marks scored by the student in 3 subjects */
    /* Display a message before accepting the marks*/
    printf("Enter the marks scored by the student in 3 subjects\n");

    scanf("%f%f%f",&fMark1,&fMark2,&fMark3);
        /* calculating the average marks */
    fSum=fMark1+fMark2+fMark3;
    fAvg=fSum/3;
        /* compare the average with 65 and decide whether student has
passed or failed */
    if ( fAvg >= 65.0)
        printf("Student PASSES\n");
    else
        printf("Student FAILS\n");
        /* accept the choice from user if he wants to find the result of
another student */
    printf("Do you wish to continue? Enter Y/N or y/n :");
        /* clear the input buffer */
    fflush(stdin);
    scanf("%c",&cChoice);
}
return 0;
}
/*****
End of enrwhile_demo.c
*****/
```

## What is the output of the following code snippet?

```
unsigned int iCount = 1;  
while (iCount<10);  
{  
    printf("%u",iCount);  
}
```

Because of THIS → ;

NO OUTPUT!!!

Results in an infinite loop.. WHY???



Infosys®

POWERED BY INTELLECT  
DRIVEN BY VALUES



**End of Day-2**