Modular Programming

- Modular programming is a strategy applied to the design and development of software systems.
- Defined as organizing large program into small, independent program segments called ***modules*** that are separately named and individually callable ***program units***.
- Modules are carefully integrated to become a software system that satisfies the system requirements.
- Modules are based on ***divide and conquer*** approach, and are identified and designed such that they can be organized into a top-down hierarchical structure.
- Characteristics of modular programming is:

1. Each module should do only one thing.
2. Communication between modules is allowed only by a calling module.
3. A module can be called by one and only one higher module.
4. No communication can take place directly between modules that do not have calling-called relationship.
5. All modules are designed as ***single-entry, single-exit systems using control structures***.

Elements of User-Defined Functions

- ***Functions are classified as one of the derived data types in C.***
- Functions and variables have some similarities as:

1. ***Both function name and variable names are considered identifiers and therefore they must adhere to the rules for identifiers.***
2. ***Like variables functions have types (such as int) associated with them.***
3. ***Like variables, function names and their types must be declared and defined before they are used in a program.***

- Three elements of a function are:

1. ***Function definition:*** independent program module written to implement the requirements of function.
2. ***Function call:*** to use function we need to invoke it at a required location in the program
3. ***Function declaration:*** calling program should declare any function that is to be used later in the program.

- ***Function definition*** or ***function implementation*** includes ***function name***, ***function type***, ***list of parameters***, ***function statements***, and ***a return statement***.

```
function_type function_name(parameter list)
{
local variables declaration;
executable statement1;
}
```

Elements of User-Defined Functions contd.

- ***function_type function_name(parameter list)*** is called function header and comprise of three parts: ***the function type, the function name, and formal parameter***.
- ***function type*** specifies the type of value (***like float or double***) that the function is expected to return to the program. If return type is not explicitly specified then C will assume integer type. If function does not return anything then we need to specify ***void***.
- ***Function name*** is any valid C identifier so it must follow the same rules of formation as variable name in C.
- ***Formal Parameter List: Parameter*** list declares the variables that will receive the data sent by the calling program. They serve as input data to the function to carry out the specified task. Referred to as ***formal parameters or arguments***, and can be utilized to send values to the calling programs.
- Parameter lists contains variable declaration separated by commas and surrounded by parentheses. e.g. ***float quadratic (int a, int b, int c){…} void printline (void) int sum (int a, int b)***
- Function body contains the declarations and statements necessary for performing required task, and contain three parts:

1. ***Local declaration that specify the variables needed by the function.***
2. ***Function statements that perform the task of the function.***
3. ***A return statement that returns the value evaluated by the function***

Elements of User-Defined Functions contd.

- Function examples are:

```
float mul (float x, float y)
{
float result; /* local variable */
result = x * y; /* computes the product */
return (result); /* returns the result */
}
```

```
void sum (int a, int b)
{
printf ("sum = %s", a + b); /* no local variables */
return; /* optional */
}
```

```
void display (void)
{ /* no local variables */
printf ("No type, no parameters");
/* no return statement */
}
```

Note

1. When a function reaches its return statement, the control is transferred back to the calling program. In the absence of a return statement, the closing brace acts as a *void return.*
2. A *local variable* is a variable that is defined inside a function and used without having any role in the communication between functions.

Return Values and Their Types

- A function may or may not return any value to the calling function.
- ***return*** statement is used to return a value of the function invoked. Any number of values can be passed to the called function, however called function at the most can only return ***one value*** per call.
- ***return*** statement can take the following form: ***return;*** or ***return(expression);***
- ***'return'*** statement doesn't return any value, and acts as the closing brace of the function. When a return is encountered the control is immediately passed back to the calling function. e.g. ***if(error) return;***
- ***Note: In C99, if a function is specified as returning a value, the return must have value associated with it.***
- In ***return(expression)*** statement the value of the expression is returned. e.g.

```
int mul (int x, int y)
{
int p; p = x*y; return(p);   // This can also be written as return(x*y)
}
```

- Returns the value of ***p*** which is the product of the values of ***x*** and ***y***.
- Function can have more than one return statements and are encountered in conditional statements. e.g. ***if(x<=0) return (0); else return (1);***
- Type specifier is utilized in the function header to return a particular type of data.
- ***Note: Only one value can be returned from a C function. To return multiple values, we have to use pointers or structures.***

Function Calls

- Function can be called by simply using function name followed by a list of ***actual parameters***. e.g.

```
main( )
{int y;
y = mul(10,5);          /* Function call */
printf("%d\n", y);}
int mul (int x, int y)
{int p; /* local variable*/
p=x*y; /* x = 10, y = 5*/
return(p);}
```

- When compiler encounters a function call, the control is transferred to the function ***mul()***. The function call sends two integer values 10 and 5 to the function, and return 25 to the main function.
- Many different ways to invoke a function are: ***mul(10,5); mul(m,5); mul(10,n); mul(m,n); mul(m+5,10); mul(10,mul(m,n))***; and ***mul(expression1,expression2);***
- ***Note: Sixth call uses its own call as its one of the parameters. When we use expression, they should be evaluated to single values that can be passed as actual parameters***
- Function returning a value can be used in expressions like any other variable like ***printf("%d\n", mul(p,q));y = mul(p,q) / (p+q);if (mul(m,n)>total) printf("large");***

Function Calls contd.

- Function call is a postfix expression. The operator (.,.) is at a very high level of precedence.
- When a function call is used as a part of an expression, it will be evaluated first, unless parentheses are used to change the order of precedence.
- ***In a function call, the function name is the operand and the parentheses set (.,.) containing actual parameters is the operator.***
- Actual parameters must match the function's formal parameters in type, order and number. Multiple actual parameters must be separated by commas.

NOTE:

1. If the actual parameters are more than the formal parameters, the extra actual arguments will be discarded.
2. On the other hand, if the actuals are less than the formals, the unmatched formal arguments will be initialized to some garbage.
3. Any mismatch in data types may also result in some garbage values.

Function Declaration

- Function declaration consists of four parts:
  - Function type (return type)
  - Function name
  - Parameter list
  - Terminating semicolon
- Prototype declaration may be placed above all the functions (including main), and inside a function definition. Syntax for the function declaration is: ***Function-type function-name*** **(parameter list);**

Points to note

1. The parameter list must be separated by commas.
2. The parameter names do not need to be the same in the prototype declaration and the function definition.
3. The types must match the types of parameters in the function definition, in number and order.
4. Use of parameter names in the declaration is optional.
5. If the function has no formal parameters, the list is written as (void).
6. The return type is optional, when the function returns **int** type data.
7. The retype must be **void** if no value is returned.
8. When the declared types do not match with the types in the function definition, compiler will produce an error.

Function Category

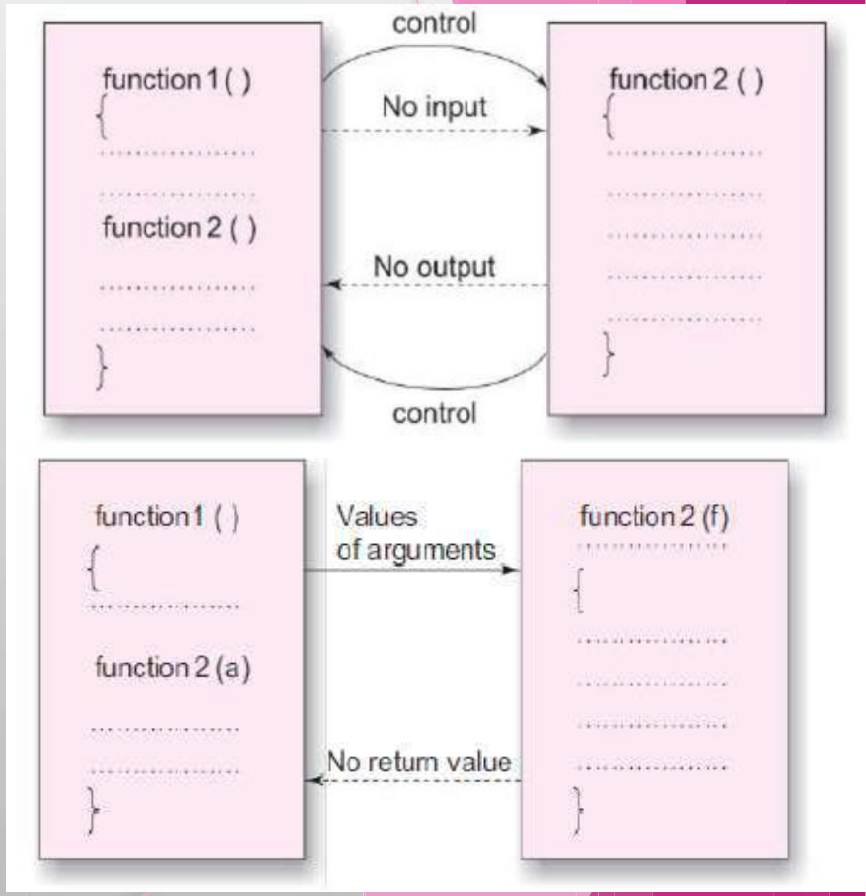
- Function declaration depending on whether arguments are present or not and whether a value is returned or not can be classified as:

- ***Category1: Functions with no arguments and no return values***

- When a function has no arguments, it does not receive any data from the calling function. Similarly, when it does not return a value, the calling function does not receive any data from the called function. In effect there is no data transfer between the calling and called function.

- ***Category2: Functions with arguments and no return values***

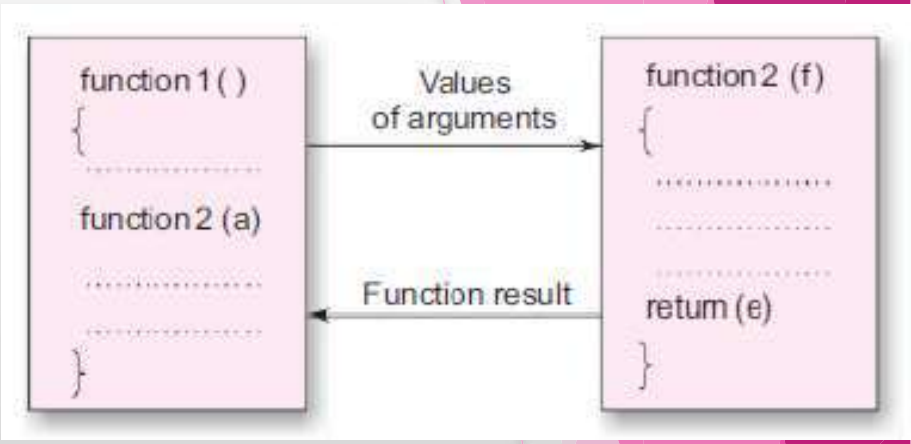
- In category 1 ***main*** function has no control over the way the functions receive input data. The calling function can read the data from the terminal and pass it on to the called function. This approach is more suitable because the calling function can check for the validity of data.



Function Category contd.

- ***Category3: Functions with arguments and one return value***

In category 2, function receives data from the calling function through arguments but does not send back any value. Rather, it displays the calculations at the terminal. However, instead of displaying the function result we want to use the result for further processing. A self-contained and independent function should behave like a black-box that receives a predefined form of input and outputs a desired value.



Note: In C function returns integer type values as the default case when nothing is specified explicitly. But while calculating mean and standard deviation we require floating or double data type so we need to specify the floating return function value.

- **Category4: Functions with no arguments but return a value**

There are scenarios where we may need to design function that may not take any arguments but return a value to the calling function. e.g. the ***getchar*** function declared in header file ***<stdio.h>***. The ***getchar*** function has no parameters but it return an integer data type that represents a character. We can also create a user defined function with return value and no arguments. e.g.

```
int get_number(void);main()
{int m = get_number( );printf("%d",m);}int get_number(void)
{int number;scanf("%d", &number);return(number);}
```

Function Category

- ***Category5: Functions that return multiple values***
- Uptill now we have analyzed functions with return value, that can return only one value. To return multiple values the arguments are used not only to receive information but also to send back information to the called function. The mechanism of information sending is achieved through ***address operator &*** and ***indirection operator ****.

```
void mathoperation (int x, int y, int *s, int *d);
main( )
{int x = 20, y = 10, s, d;mathoperation(x,y, &s, &d);
printf("s=%d\n d=%d\n", s,d);}
void mathoperation (int a, int b, int *sum, int *diff)
{*sum = a+b;*diff = a-b;}
```

- ***x and y are actual arguments, s and d are output arguments***. In function call we pass the actual values of ***x*** and ***y,*** and we also pass the addresses of locations where the values of ***s*** and ***d*** are stored in memory.
- During function call the following assignments occur: ***value of x to a, value of y to b, address of s to sum, address of d to diff***
- **** indirection operator*** in ***sum*** and ***diff*** indicates these variables store address not the actual variables value.
- **** sum = a+b;*** adds the values and stores the result at memory location pointed by ***sum*** which is same as ***s***. Therefore the values stored at location pointed by ***sum*** is the value of ***s***.
- ***This type of function call is called the call by reference or pass by pointers***.

Rules for Pass by Pointers

1. The types of the actual and formal arguments must be same.
2. The actual arguments (in the function call) must be the addresses of variables that are local to the calling function.
3. The formal arguments in the function header must be prefixed by the indirection operatior *.
4. In the prototype, the arguments must be prefixed by the symbol *.
5. To access the value of an actual argument in the called function, we must use the corresponding formal argument prefixed with the indirection operator *.

Nested Functions

- C permits nesting of functions freely. ***main()*** can call ***function1()***, which can call ***function2()***, which can call ***function3()***, and so on.

```
float ratio (int x, int y, int z);
int difference (int x, int y);
main( )
{int a, b, c; scanf("%d %d %d", &a, &b, &c);printf("%f \n", ratio(a,b,c));}
float ratio(int x, int y, int z)
{if(difference(y, z))
return(x/(y-z));
else
return(0.0);}
int difference(int p, int q)
{if(p != q)
return (1);
else
return(0);}
```

Recursion

- The process of chaining occurs when a function in turn calls another function.
- Recursion is a special case of chaining process where a function calls itself. e.g.

```
main( )
{printf("This is an example of recursion\n");
main( );}
```

- When executed the program will produce output as:

```
This is an example of recursion
This is an example of recursion
This is an example of recursion
This is an ex
```

- Execution is terminated abruptly; otherwise the execution will continue indefinitely.

Recursion contd.

- Another example of recursion is the evaluation of factorials of a given number.
- The factorial of a number n is expressed as a series of repetitive multiplications given as ***n!= n(n-1)(n-2)….1.***
  e.g. 4!=4x3x2x1=24

```
factorial(int n)
{int fact;
if (n==1)
return(1);
else
fact = n*factorial(n-1);return(fact);}
```

- So, lets see how recursion works. Assume ***n=4***, since ***n*** is not ***1***, the statement ***fact=n*factorial(n-1)*** will be executed ***3*** times, i.e., ***fact=3*factorial(2);*** will be evaluated. The expression has a function call ***factorial*** with ***n=2***, so the execution becomes ***2*factorial(1);*** will be evaluated. The expression has a function call ***factorial*** with ***n=1***, so the execution becomes ***1;***
- Hence the resultant fact is ***fact=3x2x1=6***.
- Recursive functions can be effectively used to solve problems where solution is expressed in terms of successively applying the same solution to subsets of the problem.
- In recursive functions, ***if*** statement must be present somewhere to force the function to return without the recursive call being executed.

Passing Arrays to Functions

- Like variables, array values can be passed into a function.
- To pass one-dimensional an array to a called function, it is sufficient to list the name of the array, without subscript, and array size as arguments. e.g. the function call ***largest(a,n)*** will pass whole array ***a*** to the called function. The declaration of ***largest*** function will look like:

float largest(float array[],int size)

- Function ***largest*** is defined to ***take two arguments***, the ***array name*** and the ***array size*** to specify the number of elements in the array. Formal argument array declaration is made as: ***float array[];***
- ***Note: It is not necessary to specify the size of the array here.***

```
main( )
{float largest(float a[ ], int n);
float value[4] = {2.5,-4.75,1.2,3.67};printf("%f\n", largest(value,4));}
float largest(float a[], int n)
{int i;float max;max = a[0];
for(i = 1; i < n; i++){if(max < a[i])}
max = a[i];return(max);}
```

- In C array name represents the address of its first element. By passing array name, we are passing the address of the array to the called function. The array in the called function now refers to the same array stored in the memory.
- ***Passing address of parameters to the function is referred to as pass by address***

Rules to Pass Arrays to Functions

- When dealing with array arguments, if a function changes the values of the elements of an array, then these changes will be made to the original array.
- When an entire array is passed as an argument, the contents of the array are not copied into the formal parameter array; instead, information about the addresses of array elements are passed on to the function.
- Any change introduced to the array elements are truly reflected in the original array in the calling function. However, this does not apply when an individual element is passed on as argument.

Three Rules to Pass an Array to a Function

1. The function must be called by passing only the name of the array.
2. In the function definition, the formal parameter must be an array type; the size of the array does not need to be specified.
3. The function prototype must show that the argument is an array.

Passing Multi-dimensional Arrays to Functions

- Like simple one-dimensional array we can also pass multi-dimensional arrays to functions.
- The approach is similar to the one-dimensional arrays with certain rules.

1. The function must be called by passing only the array name.
2. In the function definition, we must indicate that the array has two-dimensions by including two sets of brackets
3. The size of the second dimension must be specified.
4. The prototype declaration should be similar to the function header.

```
double average(int x[][N], int M, int N)
{int i, j;double sum = 0.0;
for (i=0; i<M; i++)
{for(j=1; j<N; j++)
{sum += x[i][j];}}return(sum/(M*N));}
main( )
{int M=3, N=2;double average(int [ ] [N], int, int);double mean;
int matrix [M][N]=
{{1,2},{3,4},{5,6}};
mean = average(matrix, M, N);}
```

Pass by Value vs Pass by Pointers

- Technique used to pass data from one function to another is known as parameter passing, and can be done by:

1. ***Pass by value (also known as call by value):*** Values of actual parameters are copied to the variables in the parameter list of the called function. The called function works on the copy and not on the original values of the actual parameters, ensuring the original data in the calling function cannot be changed accidently.
2. ***Pass by reference (also known as call by pointers):*** Memory addresses of the variables rather than the copies of values are sent to the called function. In this case, the called function directly works on the data in the calling function and the changed values are available in the calling function. This is utilized for array and string manipulation, along with when multiple values are to be returned by the called function.

```
// Call by Value
void swap(int a, int b)
{int temp = a; a = b;b = temp;}
// Driver code
int main()
{int x = 10, y = 20;
printf("Values of x and y before swap are: %d, %d\n",
x,y);
swap(x, y);
printf("Values of x and y after swap are: %d, %d",
x,y);
return 0;}
```

```
// Call by Reference
void swap(int* a, int* b)
{int temp = *a;*a = *b;*b = temp;}
// Driver code
int main()
{int x = 10, y = 20;
printf("Values of x and y before swap are: %d,%d\n",
x,y);
swap(&x, &y);
printf("Values of x and y after swap are: %d, %d",
x,y);
return 0;}
```

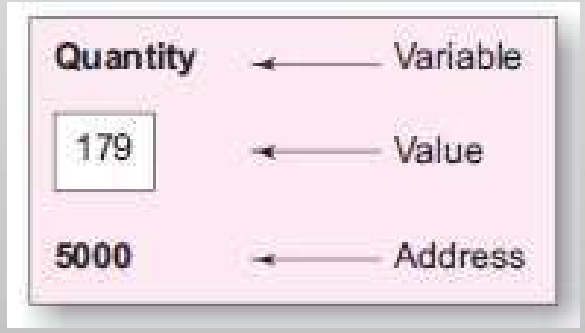
Pointers

- ***Pointer is a derived data type in C, and is made from the fundamental data types available in C.***
- Pointers contain ***memory addresses*** as their values.
- Since these memory addresses are the locations in the computer memory where program instructions and data are stored, pointers can be used to access and manipulate data stored in the memory.
- Pointers are undoubtedly one of the most distinct and exciting features of C language.
- It has added power and flexibility to C language. Benefits of using pointers are:

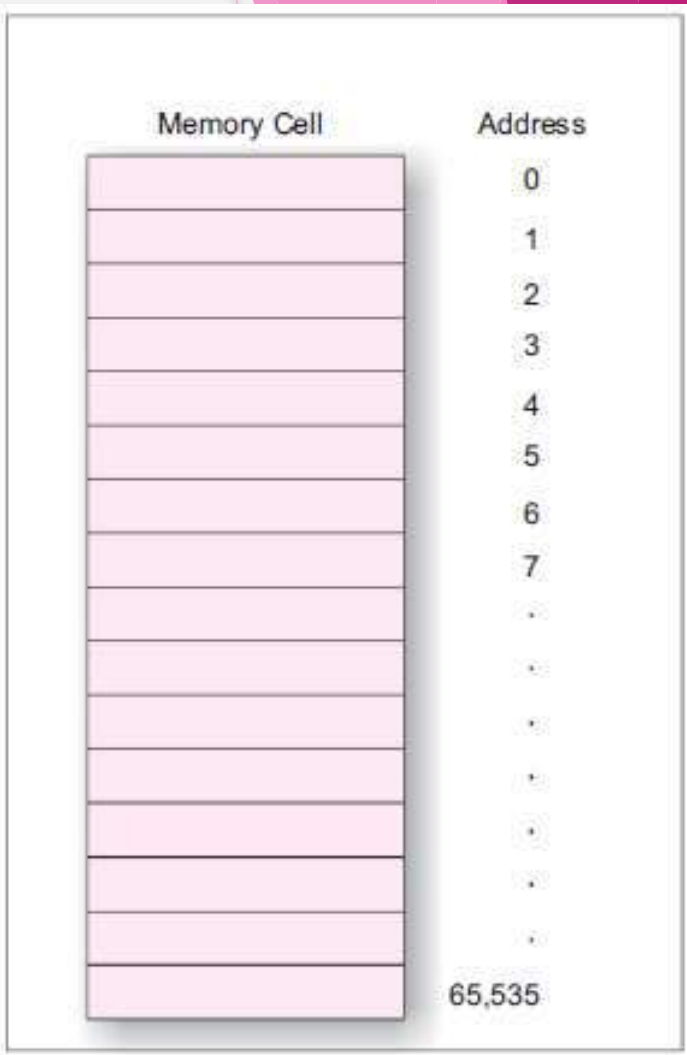
1. Pointers are more efficient in handling arrays and data tables.
2. Pointers can be used to return multiple values from a function via function arguments.
3. Pointers permit references to functions and thereby facilitating passing of functions as arguments to other functions.
4. The use of pointer arrays to character strings results in saving of data storage space in memory.
5. Pointers allow C to support dynamic memory management.
6. Pointer provide an efficient tool for manipulating dynamic data structures such as structure, linked lists, queues, stacks and trees.
7. Pointers reduce length and complexity of programs.
8. They increase the execution speed and thus reduce the program execution time.

Pointers contd.

- Memory is a sequential collection of storage cells as shown in Fig., where each cell is commonly known as byte, has address associated with it.
- ***Typically, the addresses are numbered consecutively, starting from zero. Last address depends on the memory size. e.g. 16-bit computer system having 64 KB memory will have 65535 as its final address location.***
- Whenever we declare a variable, the system allocates, somewhere in the memory, an appropriate location to hold the value of the variable.
- Since, every byte has a unique address number, this location will have its own address number. e.g. ***int quantity =179;***

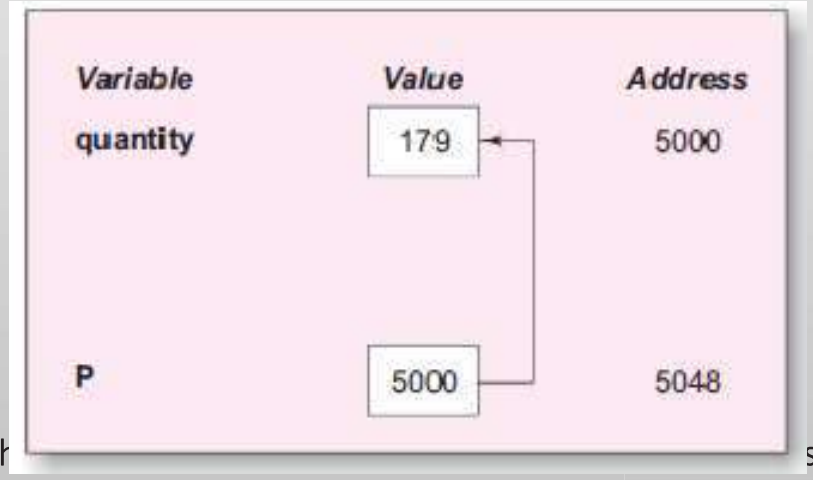


- During program execution, the system always associates the name quantity with the address 5000. We may have access to the value 179 by using either the name ***quantity*** or the address 5000.



Pointers contd.

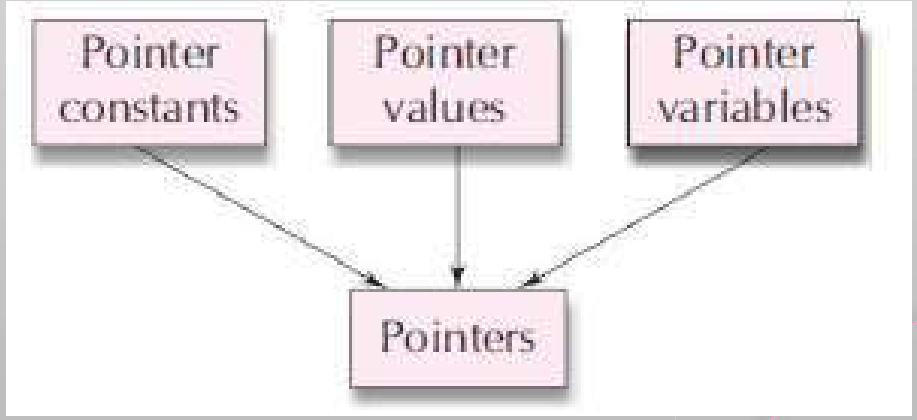
- Since memory addresses are simply numbers, they can be assigned to some variables, that can be stored in memory, like any other variable. ***Variables that hold memory addresses are called pointer variables.***
- A pointer variable is nothing but a variable that contains an address, which is a location of another variable in memory.
- Since a pointer is a variable, its value is also stored in the memory in another location. Suppose, we assign the address of ***quantity*** to a variable ***p***. The link between the variables ***p*** and ***quantity*** can be visualized as shown with address of ***p*** being ***5048.***



- Since the value of the variable ***p*** is th s the value of ***quantity*** by using the value of ***p*** and therefore, we say that the variable ***p*** points to the variable ***quantity***. Thus, ***p*** gets the name ***'pointer'.***

Pointers contd.

- Pointers are built on the three underlying concepts. Memory addresses within a computer are referred to as pointer constants.
- We cannot change them; we can only use them to store data values. They are like house numbers.
- We cannot save the value of a memory address directly. We can only obtain the value through the variable stored there using the address operator (&).
- The value thus obtained is known as pointer value. The pointer value (i.e. the address of a variable) may change from one run of the program to another.
- Once we have a pointer value, it can be stored into another variable. The variable that contains a pointer value is called a pointer variable.



Accessing the address of a variable

- The actual location of a variable in the memory is system dependent and therefore, the address of a variables is not known to us immediately.
- This can be done with the help of the operator ***&*** available in C. We have already seen the use of this address operator in the ***scanf*** function.
- The operator ***&*** immediately preceding a variable returns the address of the variable associated with it. e.g. ***p=&quantity;*** would assign the address ***5000*** (the location of quantity) to the variable ***p***.
- The ***&*** operator can be remembered as ***'address of'.***
- ***The & operator can be used only with a simple variable or an array element. The following are illegal use of address operator:***

1. ***&125 (pointing at constants).***
2. ***int x[10];***

   &x (pointing at array names).
3. ***&(x+y) (pointing at expressions).***

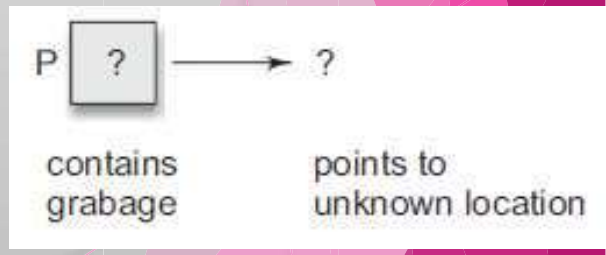
- If ***x*** is an array, then expressions such as ***&x[0] and &x[i+3]*** are ***valid*** and represent the addresses of ***0th*** and ***(i+3)th*** elements of ***x***.

Declaring Pointer variable

- In C, every variable must be declared for its type. Since pointer variables contain addresses that belong to a separate data type, they must be declared as pointers before we use them.
- The pointer variable is declared as: ***data_type *pt_name;***
- This tells the compiler three things about the variable ***pt_name.***

1. ***The asterisk (*) tells that the variable pt_name is a pointer variable.***
2. ***pt_name needs a memory location.***
3. ***pt_name points to a variable of type data_type.***

- e.g. ***int *p; /* integer pointer */*** declares the ***variable p*** as a ***pointer variable*** that ***points to an integer data type.***
- The type ***int*** refers to the ***data type of the variable being pointed to by p*** and ***not the type of the value of the pointer.***
- The statement ***float *x; / * float pointer */*** declares x as a pointer to a floating -point variable.



- The declaration cause the compiler to allocate memory locations for the pointer variables ***p and x.***
- Since the memory locations have not been assigned any values, these locations may contain some unknown values in them and they point to unknown locations.

Pointer Declaration Style

- Pointer variables are declared similarly as normal variables except for the addition of the unary * operator.
- * can appear anywhere between the type name and the printer variable name.
- ***Various pointer declaration styles are:***

```
int*  p;  /*Style 1*/
int *p; /*Style 2*/
int * p; /*Style 3*/
```

- Style 2 the most convinient method and used for accessing the target values. e.g.

```
int x, *p, y;
x = 10;
p = & x;
y = *p; /*accessing x through p*/
*p = 20; /*accessing 20 through x*/
```

Pointer Initialization

- Process of assigning the address of a variable to a pointer variable is known as ***initialization***.
- All uninitialized pointers will have some unknown values that will be interpreted as memory addresses. They may not be valid addresses or they may point to some values that are wrong. Since the compilers do not detect these errors, the programs with uninitialized pointers will produce erroneous results.
- It is therefore important to initialize pointer variables carefully before they are used in the program.
- Once a pointer variable has been declared we can use the assignment operator to initialize the variable. e.g.

```
int quantity;
int *p;  //declaration
P=&quantity; //initialization
```

- We must ensure that the pointer variables always point to the corresponding type of data. E.g.

```
float a, b;int x, *p;p = &a; /* wrong */
b = *p;
```

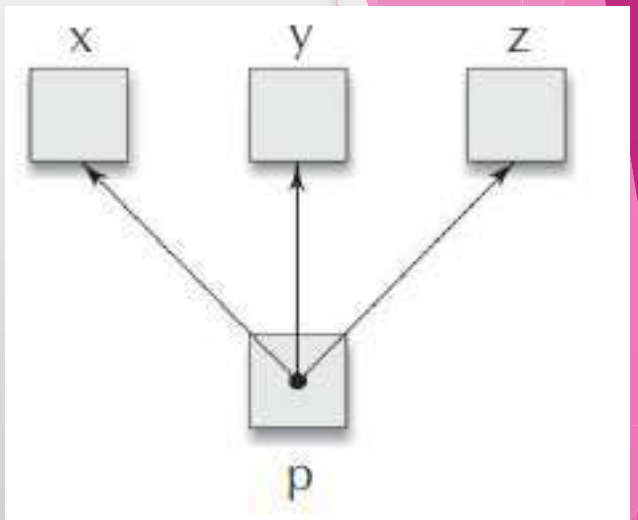
will result in erroneous output because we are trying to assign the address of a ***float*** variable to an ***integer pointer***.

- When we declare a pointer to be of ***int*** type, the system assumes that any address that the pointer will hold will point to an integer variable. Since the compiler will not detect such errors, care should be taken to avoid wrong pointer assignments.
- It is also possible to combine the declaration of data variable, the declaration of pointer variable and the initialization of the pointer variable in one step. e.g. ***int x, *p = &x; /* three in one */*** is perfectly valid.
- It declares ***x*** as an integer variable and ***p*** as a pointer variable and then initializes ***p*** to the address of ***x***. And also remember that the target variable ***x*** is declared first.
- ***The pointer variable can be declared with initial value of NULL or 0 (zero)***

Pointer Flexibility

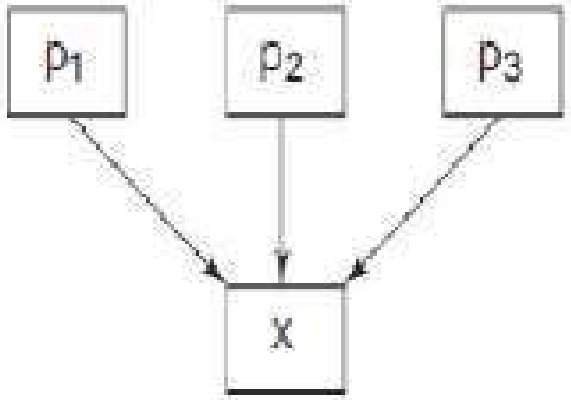
- Pointers are flexible. We can make the same pointer to point to different data variables in different statements.

```
int x, y, z, *p;
. . . . .
p = &x;
. . . . .
p = &y;
. . . . .
p = &z;
. . . . .
```



- We can also use different pointers to point to the same data variable.

```
int x;
int *p1 = &x;
int *p2 = &x;
int *p3 = &x;
. . . . .
. . . . .
```



Chain of Pointers

- Possible to make a pointer to point to another pointer, thus creating a chain of pointers.
- The pointer variable p2 contains the address of the pointer variable p1, which points to the location that contains the desired value. This is known as ***multiple indirections***.
- A variable that is a pointer to a pointer must be declared using additional indirection operator symbols in front of the name. ***int **p2;***
- ***This declaration tell the compiler that p2 is a pointer to a pointer of int type.***
- ***Note: The pointer p2 is not a pointer to an integer, but rather a pointer to an integer pointer.***

- To access target value indirectly pointed to by pointer to a pointer by applying the indirection operator twice.

```
main ( )
{
int x, *p1, **p2;
x = 100;
p1 = &x; /* address of x */
p2 = &p1 /* address of p1 */
printf ("%d", **p2);
}
```

- The code will display the value 100. where, p1 is declared as a pointer to an integer and p2 as a pointer to a pointer to an integer

Pointer Expressions

- Pointer variables can be used in expressions. e.g. if ***p1*** and ***p2*** are properly declared and initialized pointers, then the following statements are valid.

  y = *p1 * *p2; same as (*p1) * (*p2)

  sum = sum + *p1;

  z = 5* - *p2/ *p1; same as (5 * (- (*p2)))/(*p1)

  ****p2 = *p2 + 10;***

- C allows us to add integers to or subtract integers from pointers, as well as to subtract one pointer from another. ***p1 + 4***, ***p2-2*** and ***p1 - p2*** are all allowed.
- If ***p1*** and ***p2*** are both pointers to the same array, then ***p2 - p1*** gives the number of elements between ***p1*** and ***p2***.
- We may also use short-hand operators with the pointers. e.g. ***p1++;-p2; sum+=*p2;***
- In addition to arithmetic operations discussed above, pointers can also be compared using the relational operator. ***p1 > p2, p1 = = p2, and p1 != p2 are allowed***.
- However, any comparison of pointers that refer to separate and unrelated variables makes no sense. Comparisons can be used meaningfully in handling arrays and strings.
- We may not use pointers in division or multiplication. For example, expressions such as ***p1 / p2 or p1 * p2 or p1 / 3***

Pointer Increment and Scale Factor

- Pointers can be incremented like: ***p1 = p2 + 2; p1 = p1 + 1;***
- Expression like ***p1++;*** will cause the pointer ***p1*** to point to the next value of its type.
- ***p1*** is an integer pointer with an initial value, say ***2800***, then after the operation ***p1 = p1 + 1***, the value of ***p1*** will be ***2802***, and not ***2801***
- ***When we increment a pointer, its value is increased by the length of the data type that it points to.***
- ***This length called the scale factor.***
- Number of bytes used to store variable data types depends on the system and can be found by making use of the ***sizeof*** operator. e.g. if x is a variable, then ***sizeof(x)*** returns the number of bytes needed for the variable.

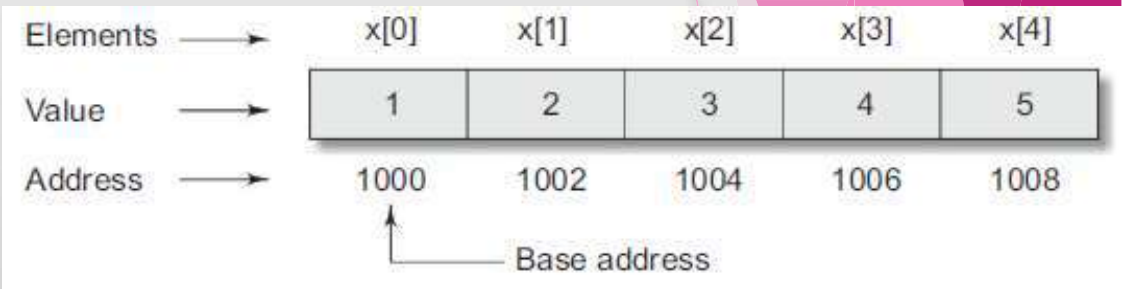
Rules of Pointer Operations

The following rules apply when performing operations on pointer variables.

1. A pointer variable can be assigned the address of another variable.
2. A pointer variable can be assigned the values of another pointer variable.
3. A pointer variable can be initialized with NULL or zero value.
4. A pointer variable can be pre-fixed or post-fixed with increment or decrement operators.
5. An integer value may be added or subtracted from a pointer variable.
6. When two pointers point to the same array, one pointer variable can be subtracted from another.
7. When two pointers point to the objects of the same data types, they can be compared using relational operators.
8. A pointer variable cannot be multiplied by a constant.
9. Two pointer variables cannot be added.
10. A value cannot be assigned to an arbitrary address (i.e., &x = 10; is illegal).

Pointers and Arrays

- When an array is declared, the compiler allocates a base address and sufficient amount of storage to contain all the elements of the array in contiguous memory locations.
- The base address is the location of first element of the array. The compiler defines the array name as a constant pointer to the first element.. e.g. ***int x[5]={1,2,3,4,5};***
- ***Let the base address of x is 1000 and assuming that each integer requires two bytes, the five element.***



- Name ***x*** is a constant pointer to the first element ***x[0]*** and therefore the value of ***x*** is ***1000***, the location where ***x[0]*** is stored, i.e., ***x=&x[0]=1000***
- We declare ***p*** as an integer pointer, then we can make the pointer ***p*** to point to the array ***x*** by the ***p=x; p=&x[0]***
- We can access every value of ***x*** using ***p++*** to move from one element to another. Relationship between ***p*** and ***x*** is:
- ***p = &x[0] (= 1000) p+1 = &x[1] (= 1002) p+2 = &x[2] (= 1004) p+3 = &x[3] (= 1006) p+4 = &x[4] (= 1008)***
- ***Note that address of an element is calculated using its index and the scale factor of the data type. The address of x[3] = base address + (3 x scale factor of int) = 1000 + (3 x 2) = 1006***
- When handling arrays, instead of using array indexing, we can use pointers to access array elements.

Pointers and Character Strings

- Strings are treated like character arrays and therefore, they are declared and initialized as follows: ***char str [5] = "good";***
- Alternative method to create strings using pointer variables of type ***char***. e.g. ***char *str = "good";***
- This creates a string for the literal and then stores its address in the ***pointer variable str***. The pointer ***str*** now points to the first character of the string *"good"*.
- We can also use the run-time assignment for giving values to a string pointer as ***char * string1; string1= "good";***
- Note that the assignment ***string1 = "good";*** is not a string copy, because the variable string1 is a pointer, not a stringand can printed as ***printf("%s",string1);***

Troubles with Pointers

- Some pointer errors that are more commonly committed by the programmers which lead to unexpected results are

1. *Assigning values to uninitialized pointers*

```
int * p, m = 100 ;
*p = m ; /* Error */
```

2. *Assigning value to a pointer variable*

```
int *p, m = 100 ;
p = m; /* Error */
```

3. *Not dereferencing a pointer when required*

```
int *p, x = 100;
p = &x;
printf("%d",p); /* Error */
```

4. *Assigning the address of an uninitialized variable*

```
int m, *p
p = &m; /* Error */
```

5. *Comparing pointers that point to different objects*

```
char name1 [ 20 ], name2 [ 30 ];
char *p1 = name1;
char *p2 = name2;
if(p1 > p2)..... /* Error */
```

- We must be careful in declaring and assigning values to pointers correctly before using them.

Module5

File Management in C, Memory Allocation

File Management

- Until now we have been using the functions such as ***scanf*** and ***printf*** to read and write data.
- These are console oriented I/O functions, which always use the terminal (keyboard and screen) as the target place.
- Works fine for small data but real-life problems involving large volumes of data and in such situations, the console oriented I/O operations pose two major problems.

1. It becomes cumbersome and time consuming to handle large volumes of data through terminals.
2. The entire data is lost when either the program is terminated or the computer is turned off.

- Therefore necessary to have a more flexible approach where data can be stored on disks and read whenever necessary, without destroying the data. This method employs the concept of files to store data.
- ***A file is a place on the disk where a group of related data is stored.***
- ***Basic file operations include:***

1. ***File naming***
2. ***File opening***
3. ***Reading data from file***
4. ***Writing data to a file***
5. ***File closing***

File Management contd.

- File operations require:

1. ***Filename:*** string of characters that make up a valid filename for the operating system, and contains two parts a primary name and an optional period. e.g. ***Input.data, store, PROG.C, Student.c, Text.out***
2. ***Data Structures:*** is a file defined as ***FILE*** in the library of standard I/O function.
3. ***Purpose***

Function name	*Operation*
fopen()	* Creates a new file for use. * Opens an existing file for use.
fclose()	* Closes a file which has been opened for use.
getc()	* Reads a character from a file.
putc()	* Writes a character to a file.
fprintf()	* Writes a set of data values to a file.
fscanf()	* Reads a set of data values from a file.
getw()	* Reads an integer from a file.
putw()	* Writes an integer to a file.
fseek()	* Sets the position to a desired point in the file.
ftell()	* Gives the current position in the file (in terms of bytes from the start).
rewind()	* Sets the position to the beginning of the file.

File Management contd.

File operation	Declaration & Description
fopen() – To open a file	Declaration: FILE *fopen (const char *filename, const char *mode) fopen() function is used to open a file to perform operations such as reading, writing etc. In a C program, we declare a file pointer and use fopen() as below. fopen() function creates a new file if the mentioned file name does not exist. FILE *fp; fp=fopen ("filename", "'mode"); Where, fp – file pointer to the data type "FILE". filename – the actual file name with full path of the file. mode – refers to the operation that will be performed on the file. Example: r, w, a, r+, w+ and a+. Please refer below the description for these mode of operations.
fclose() – To close a file	Declaration: int fclose(FILE *fp); fclose() function closes the file that is being pointed by file pointer fp. In a C program, we close a file as below. fclose (fp);
fgets() – To read a file	Declaration: char *fgets(char *string, int n, FILE *fp) fgets function is used to read a file line by line. In a C program, we use fgets function as below. fgets (buffer, size, fp); where, buffer – buffer to put the data in. size – size of the buffer fp – file pointer
fprintf() – To write into a file	Declaration: int fprintf(FILE *fp, const char *format, ...);fprintf() function writes string into a file pointed by fp. In a C program, we write string into a file as below. fprintf (fp, "some data"); or fprintf (fp, "text %d", variable_name);

Random Access to Files

Statement	*Meaning*
fseek(fp,0L,0);	Go to the beginning. (Similar to rewind)
fseek(fp,0L,1);	Stay at the current position. (Rarely used)
fseek(fp,0L,2);	Go to the end of the file, past the last character of the file.
fseek(fp,m,0);	Move to (m+1)th byte in the file.
fseek(fp,m,1);	Go forward by m bytes.
fseek(fp,-m,1);	Go backward by m bytes from the current position.
fseek(fp,-m,2);	Go backward by m bytes from the end. (Positions the file to the mth character from the end.)