**AALIM MUHAMMED SALEGH COLLEGE OF ENGINEERING**

**DEPARTMENT OF COMPUTER APPLICATION**

**SEMESTER- I**

**MC5102 – PROBLEM SOLVING AND PROGRAMMING**

**UNIT – IV NOTES**

**SYLLABUS:**

**UNIT IV ARRAYS, STRINGS, FUNCTIONS AND POINTERS**

Array – One dimensional Character Arrays- Multidimensional Arrays- Arrays of Strings - Two

dimensional character array – functions - parameter passing mechanism scope – storage classes –

recursion - comparing iteration and recursion- pointers – pointer operators - uses of pointers- arrays and pointers – pointers and strings - pointer indirection- pointers to functions - Dynamic memory allocation.

**ARRAYS**

· C programming language provides a data structure called the array, which can store a

fixed-size sequential collection of elements of the same type.

**What is an Array?**

· **An array is used to store a collection of data, of the same data name and data type.**

**An array is a sequence of data item of homogeneous value (same type).**

**Example:**

If the user wants to store marks of 100 students.

This can be done by creating 100 variables individually but, this process is rather tedious and impracticable.

This type of problem can be handled in C programming using arrays.

**int marks[100];**

This declaration statement can able to store 100 marks of integer values.

That is, 100 blocks of memory of 4 bytes each ranges of index 0 to 99.

**Two major types of Arrays**

· Arrays are of two types:

i. One-dimensional arrays

ii. Multidimensional arrays

***Declaration of one-dimensional array***

· To declare an array in C, a programmer specifies the type of the elements and the number of elements required by an array as follows:

data\_type array\_name[array\_size];

· Example:

int age[5];

Here, **the name of array is age**. The **size of array is 5**,i.e., there are 5 items(elements) of

array age. **All element in an array are of the same type** (int, in this case).

***Array elements***

· Size of array defines the number of elements in an array. Each element of array can be

accessed and used by user according to the need of program.

For example:

**int age[5];**



The first element is numbered is indexed as 0 and so on.

o Here, the size of array age is 5 times the size of int because there are 5 elements.

o Suppose, the starting address of age[0] is 2120d and the size of int be 4 bytes.

Then, the next address (address of a[1]) will be 2124d, address of a[2] will be

2128d and so on.

**Initialization of one-dimensional array:**

· Arrays can be initialized at declaration time in this source code as:

**int age[5]={2,4,34,3,4};**

It is not necessary to define the size of arrays during initialization.

int age[]={2,4,34,3,4};

In this case, the compiler determines the size of array by calculating the number of elements of an array.



**Accessing array elements**

In C programming, arrays can be accessed and treated like variables in C.

**Example of array in C programming**

/\* C program to find the sum marks of n students using arrays \*/

#include <stdio.h>

int main(){

int marks[10],i,n,sum=0;

printf("Enter number of students: ");

scanf("%d",&n);

for(i=0;i<n;++i){

printf("Enter marks of student%d: ",i+1);

scanf("%d",&marks[i]);

sum+=marks[i];

}

printf("Sum= %d",sum);

return 0;

}

**Output**

Enter number of students: 3

Enter marks of student1: 12

Enter marks of student2: 31

Enter marks of student3: 2

sum=45

**MULTIDIMENSIONAL ARRAYS**

· C programming language allows programmer to create **arrays of arrays** known as

multidimensional arrays.

· **Example: float a[2][6];**

o Here, a is an array of two dimension, which is an example of multidimensional

array. This array has 2 rows and 6 columns

o For better understanding of multidimensional arrays, array elements of above

example can be think of as below:



**Initialization of Multidimensional Arrays**

In C, multidimensional arrays can be initialized in different number of ways.

int c[2][3]={{1,3,0}, {-1,5,9}};

OR

int c[][3]={{1,3,0}, {-1,5,9}};

OR

int c[2][3]={1,3,0,-1,5,9};

**Example:**

#include <stdio.h>

int main ()

{

/\* an array with 5 rows and 2 columns\*/

int a[5][2] = { {0,0}, {1,2}, {2,4}, {3,6},{4,8}};

int i, j;

/\* output each array element's value \*/

for ( i = 0; i < 5; i++ )

{

for ( j = 0; j < 2; j++ )

{

printf("a[%d][%d] = %d\n", i,j, a[i][j] );

}

}

return 0;

}

When the above code is compiled and executed, it produces the following result:

a[0][0]: 0

a[0][1]: 0

a[1][0]: 1

a[1][1]: 2

a[2][0]: 2

a[2][1]: 4

a[3][0]: 3

a[3][1]: 6

a[4][0]: 4

a[4][1]: 8

**Initialization Of three-dimensional Array**

double cprogram[3][2][4]={

{{-0.1, 0.22, 0.3, 4.3}, {2.3, 4.7, -0.9, 2}},

{{0.9, 3.6, 4.5, 4}, {1.2, 2.4, 0.22, -1}},

{{8.2, 3.12, 34.2, 0.1}, {2.1, 3.2, 4.3, -2.0}}

};

· Suppose there is a multidimensional array arr[i][j][k][m]. Then this array can hold

i\*j\*k\*m numbers of data.

· Similarly, the array of any dimension can be initialized in C programming.

**The two major types of Arrays are :**

· Arrays are of two types:

i. One-dimensional arrays

ii. Multidimensional arrays

**(i) ONE-DIMENSIONAL ARRAY**

**An array having a single index is called one-dimensional array in C.**

**For example:**

If the user wants to store marks of 100 students.

**int marks[100];**

This declaration statement can able to store 100 marks of integer values.

That is, 100 blocks of memory of 4 bytes each ranges of index 0 to 99.

***Declaration of one-dimensional array***

· To declare an array in C, a programmer specifies the type of the elements and the number of elements required by an array as follows:

**data\_type array\_name[array\_size];**

· Example: int age[5];

Here, **the name of array is age**. The **size of array is 5**,i.e., there are 5 items(elements) of

array age. **All element in an array are of the same type** (int, in this case).

***Array elements***

· Size of array defines the number of elements in an array. Each element of array can be accessed and used by user according to the need of program.

For example:

**int age[5];**



* The first element is numbered is indexed as 0 and so on.
* Here, the size of array age is 5 times the size of int because there are 5 elements.
* Suppose, the starting address of age[0] is 2120d and the size of int be 4 bytes.
* Then, the next address (address of a[1]) will be 2124d, address of a[2] will be
* 2128d and so on.

**Initialization of one-dimensional array:**

· Arrays can be initialized at declaration time in this source code as:

**int age[5]={2,4,34,3,4};**

It is not necessary to define the size of arrays during initialization.

int age[]={2,4,34,3,4};

In this case, the compiler determines the size of array by calculating the number of elements of an array.



**Accessing array elements**

In C programming, arrays can be accessed and treated like variables in C.

**Example of array in C programming**

/\* C program to find the sum marks of n students using arrays \*/

#include <stdio.h>

int main(){

int marks[10],i,n,sum=0;

printf("Enter number of students: ");

scanf("%d",&n);

for(i=0;i<n;++i){

printf("Enter marks of student%d: ",i+1);

scanf("%d",&marks[i]);

sum+=marks[i];

}

printf("Sum= %d",sum);

return 0;

}

**Output**

Enter number of students: 3

Enter marks of student1: 12

Enter marks of student2: 31

Enter marks of student3: 2

sum=45

**ii) MULTIDIMENSIONAL ARRAY**

· C programming language allows programmer to create **arrays of arrays** known as

multidimensional arrays.

· **Example: float a[2][6];**

o Here, a is an array of two dimension, which is an example of multidimensional

array. This array has 2 rows and 6 columns

o For better understanding of multidimensional arrays, array elements of above

example can be think of as below:



**Initialization of Multidimensional Arrays**

In C, multidimensional arrays can be initialized in different number of ways.

int c[2][3]={{1,3,0}, {-1,5,9}};

OR

int c[][3]={{1,3,0}, {-1,5,9}};

OR

int c[2][3]={1,3,0,-1,5,9};

**Example:**

#include <stdio.h>

int main ()

{

/\* an array with 5 rows and 2 columns\*/

int a[5][2] = { {0,0}, {1,2}, {2,4}, {3,6},{4,8}};

int i, j;

/\* output each array element's value \*/

for ( i = 0; i < 5; i++ )

{

for ( j = 0; j < 2; j++ )

{

printf("a[%d][%d] = %d\n", i,j, a[i][j] );

}

}

return 0;

}

When the above code is compiled and executed, it produces the following result:

a[0][0]: 0

a[0][1]: 0

a[1][0]: 1

a[1][1]: 2

a[2][0]: 2

a[2][1]: 4

a[3][0]: 3

a[3][1]: 6

a[4][0]: 4

a[4][1]: 8

**STRING IN C**

· In C programming, array of character are called strings. A string is terminated by null

character /0.

· **Example: "**C-data type"

Here, "c string tutorial" is a string. When, compiler encounters strings, it

appends null character at the end of string.

**C – d a t a t y p e \0**

**Example**

char greeting[] = "Hello";

· Following is the memory presentation of above defined string in C/C++:



· Following is the memory presentation of above defined string in C/C++:

**Program**

#include <stdio.h>

int main ()

{

char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};

printf("Greeting message: %s\n", greeting );

return 0;

}

**Output**

Greeting message: Hello

**ii) String handling functions in C**

C supports a wide range of functions that manipulate null-terminated strings:

|  |  |  |
| --- | --- | --- |
| **S.No.** | **Function** | **Purpose** |
| 1 | **strcpy(s1, s2);** | Copies string s2 into string s1. |
| 2 | **strcat(s1, s2);** | Concatenates string s2 onto the end of string s1. |
| 3 | **strlen(s1);** | Returns the length of string s1. |
| 4 | **strcmp(s1, s2);** | Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2. |
| 5 | **strchr(s1, ch);** | Returns a pointer to the first occurrence of character ch in string s1. |
| 6 | **strstr(s1, s2);** | Returns a pointer to the first occurrence of string s2 in string s1. |

Following example makes use of few of the above-mentioned functions:

#include <stdio.h>

#include <string.h>

int main ()

{

char str1[12] = "Hello";

char str2[12] = "World";

char str3[12];

int len ;

/\* copy str1 into str3 \*/

strcpy(str3, str1);

printf("strcpy( str3, str1) : %s\n", str3 );

/\* concatenates str1 and str2 \*/

strcat( str1, str2);

printf("strcat( str1, str2): %s\n", str1 );

/\* total lenghth of str1 after concatenation \*/

len = strlen(str1);

printf("strlen(str1) : %d\n", len );

return 0;

}

**Output**

strcpy( str3, str1) : Hello

strcat( str1, str2): HelloWorld

strlen(str1) : 10

**ARRAYS OF STRINGS**

· An array of strings is a special form of a two-dimensional array.

· The size of the left index determines the number of strings.

· The size of the right index specifies the maximum length of each string.

For example, the following declares an array of 30 strings, each having a maximum

length of 80 characters (with one extra character for the null terminator):

char string\_array[30][81];

· For accessing an individual string, one simply specifies only the left index:

firstString = string\_array[0];

sixthString = string\_array[5];

· The following example calls the gets() function with the third string in the array:

gets(string\_array[2]);

**This program accepts lines of text entered at the keyboard and redisplays them after a blank line is entered.**

// includes go here

int main()

{

int t, i;

char text[100][80];

for(t=0; t<100; t++) {

cout << t << “: “;

gets(text[t]);

if(!text[t][0]) break; // quit on blank line

}

for(i=0; i<t; i++) // redisplay the strings

cout << text[i] << ‘\n’;

return(0);}

**An Example Using String Arrays**

Arrays of strings are commonly used for handling tables ofvinformation.

One such application would be an employee database that stores

• the name

• telephone number

• hours worked per pay period, and

• hourly wage.

**These data we could store in arrays**:

char name[20][80]; // employee names

int phone[20]; // phone numbers

float hours[20]; // hours worked

float wage[20]; // wage

**ONE DIMENSIONAL CHARACTER ARRAY INITIALIZATION**

Character arrays that will hold strings allow a shorthand initialization that takes this form:

Char array-name[size] = “string”;

**For example, the following code fragment initializes** Str to the phrase “hello”:

char str[6] = “hello”;

This is the same as writing

char str[6] = {‘h’, ‘e’, ‘l’, ‘l’, ‘o’, ‘\0’};

Remember that one has to make sure to make the array long enough to include the

null terminator.

#include <stdio.h>

int main(void)

{

char text[10][80];

int i;

for(i = 0; i < 10; i++) {

printf("some string for index %d: ", i + 1);

gets(text[ i ]);

}

do {

printf("Enter number of string (1 - 10) : ");

scanf("%d", &i);

i--; /\* adjust value to match array index \*/

if(i >= 0 && i < 10)

printf("%s\n", text[i]);

} while(i>=0);

return 0;

}

**TWO-DIMENSIONAL ARRAY OF CHARACTERS**

· It has 2 sub scripts.

· The order of the subscripts in the array declaration is important. The first subscript gives

the number of names in the array, while the second subscript gives the length of each

item in the array.

· Example

char masterlist[6][10] = {

"akshay",

"parag",

"raman",

"srinivas",

"gopal",

"rajesh"};

· Instead of initializing names, names can supplied from the keyboard

for ( i = 0 ; i <= 5 ; i++ )

scanf ( "%s", &masterlist[i][0] ) ;

· The names would be stored in the memory as shown in Figure. Note that each string ends

with a ‘\0’. The arrangement is similar to that of a two-dimensional numeric array.



Here, 65454, 65464, 65474, etc. are the base addresses of successive names.

· Even though 10 bytes are reserved for storing the name “akshay”, it occupies only 7

bytes. Thus, 3 bytes go waste.

· ‘Array of pointers’ used to avoid wastage.

**FUNCTION**

· In programming, a function **is a segment that groups code to perform a specific task.**

· A function is **a self-contained block of statements that performs a specified task.** The

specified task is repeated each time that the program calls the function.

· Functions **break large computing tasks into smaller ones.** They work together to

accomplish the goal of the whole program.

· **Every program must contain one function named main()** where the program always

begin execution.

· The function name is an identifier and should be unique.

**Types of C functions**

Basically, there are two types of functions in C on basis of whether it is defined by user or not.

· Library function

· User defined function

**i. Library function**

Library functions are the **in-built function** in C programming system. For example:

**main()**

- The execution of every C program starts from this main() function.

**printf()**

- prinf() is used for displaying output in C.

**scanf()**

- scanf() is used for taking input in C.

**ii. User defined function**

· C **allows programmer to define their own function** according to their requirement.

These types of functions are known as user-defined functions.

· **Example:**

Suppose, a programmer wants to **find factorial of a number and check whether it is prime or not in same program.**

Then, he/she can create two separate user-defined functions in that program: one for finding factorial and other for checkingwhether it is prime or not.

**Working of user-defined function in C Programming**

· Every C program begins from main() and program starts executing the codes inside

main() function.

· When the control of program reaches to function\_name() inside main() function. The

control of program jumps to void function\_name() and executes the codes inside it.

· When all the codes inside that user-defined function are executed, control of the program jumps to the statement just after function\_name() from where it is called.



**Example of user-defined function**

**Write a C program to add two integers. Make a function** add **to add integers and**

**display sum in** main() **function.**

/\*Program to demonstrate the working of user defined function\*/

#include <stdio.h>

int add(int a, int b); //function prototype(declaration)

int main(){

int num1,num2,sum;

printf("Enters two number to add\n");

scanf("%d %d",&num1,&num2);

sum=add(num1,num2); //function call

printf("sum=%d",sum);

return 0;

}

int add(int a,int b) //function declarator

{

/\* Start of function definition. \*/

int add;

add=a+b;

return add; //return statement of function

/\* End of function definition. \*/

}

**Function prototype(declaration):**

· Every function in C programming should be declared before they are used. These type of declaration are also called function prototype.

· Function prototype gives compiler information about function name, type of arguments

to be passed and return type.

**Syntax of function prototype**

**return\_type function\_name(type(1) argument(1),....,type(n) argument(n));**

· In the above example,int add(int a, int b); is a function prototype which provides

following information to the compiler:

o name of the function is add()

o return type of the function is int.

o two arguments of type int are passed to function.

· **Function prototype are not needed if user-definition function is written before**

**main() function.**

**Function call**

· Control of the program cannot be transferred to user-defined function unless it is

called invoked.

· **Syntax of function call**

**function\_name(argument(1),....argument(n));**

· In the above example, function call is made using statement add(num1,num2);

from main().

· This makes the control of program jump from that statement to function definition

and executes the codes inside that function.

**Function definition**

· Function definition contains programming codes to perform specific task.

**Syntax of function definition**

return\_type function\_name(type(1) argument(1),..,type(n) argument(n))

{

//body of function

}

**Function definition has two major components:**

***1. Function declarator***

· Function declarator is the first line of **function definition**. When a function is

called, control of the program is transferred to function declarator.

**Syntax of function declaratory**

type function\_name(type(1) argument(1),....,type(n) argument(n))

· Syntax of function declaration and declarator are almost same except, there is no

semicolon at the end of declarator and function declarator is followed by function

body.

***2. Function body***

· Function declarator is followed by body of function inside braces.

**The advantages of functions:**

1. The length of a source program can be reduced using functions at appropriate places.

2. It is easy to locate and isolate a faulty function.

3. A function may be used by many other programs.

**TYPES OF FUNCTIONS**

For better understanding of arguments and return type in functions, user-defined functions can be categorized as:

1. Function with no arguments and no return values

2. Function with arguments but no return values

3. Function with arguments and return values

**Passing arguments to functions**

· In programming, argument(parameter) refers to data this is passed to

function(function definition) while calling function.

· In above example two variable, num1 and num2 are passed to function during

function call and these arguments are accepted by arguments a and b in function

definition.



Arguments that are passed in function call and arguments that are accepted in

function definition should have same data type. For example:

If argument num1 was of int type and num2 was of float type then, argument

variable a should be of type int and b should be of type float,i.e., type of

argument during function call and function definition should be same.

· A function can be called with or without an argument.

· **Calling a Function :**

o Call a C function just by **writing function name with opening and**

**closing round brackets followed with semicolon.**

o If we have to supply parameters then we can write parameters inside pair

of round brackets.

o Parameters are optional.

· *Call Function without Passing Parameter :*

display();

· *Passing 1 Parameter to function :*

display(num);

· *Passing 2 Parameters to function :*

display(num1,num2);

· **Called function:** It is a function and a function is a group of statements that

together perform a task.

**Return Statement**

Return statement is used for returning a value from function definition to calling

function.

**Syntax of return statement**

return (expression);

For example:

return a;

return (a+b);

***i. Function With No Arguments and No Return Values***

· *The function does not receive any data from the calling function.*

· *The function has no return values, which mean calling function does not receive*

*any data from the called function.*

· *There is no data transfer between the calling function and the called function.*

/\*C program to check whether a number entered by user is prime or not using function with no

arguments and no return value\*/

#include <stdio.h>

void prime();

int main(){

prime(); //No argument is passed to prime().

return 0;

}

void prime(){

/\* There is no return value to calling function main(). Hence, return type of prime() is void \*/

int num,i,flag=0;

printf("Enter positive integer enter to check:\n");

scanf("%d",&num);

for(i=2;i<=num/2;++i){

if(num%i==0){

flag=1;

}

}

if (flag==1)

printf("%d is not prime",num);

else

printf("%d is prime",num);

}

**Expected output**

Enter positive integer enter to check 5

5 is prime

***ii. Function With Arguments But No Return Values***

· *The function receives data from the calling function.*

· *The main() function will not have any control over the way the functions receive*

*input data.*

· *User can read data from the input terminal and pass it to the called function.*

#include <stdio.h>

void prime(int);

int main(){

int n=5;

**// printf("Enter positive integer enter to check:\n");**

**// scanf("%d",&n);**

prime(n); //No argument is passed to prime().

return 0;

}

void prime(int n){

/\* There is no return value to calling function main(). Hence, return type of prime() is void \*/

int num=n,i,flag=0;

for(i=2;i<=num/2;++i){

if(num%i==0){

flag=1;

}

}

if (flag==1)

printf("%d is not prime",num);

else

printf("%d is prime",num);

}

**Expected output**

Enter positive integer enter to check 5

5 is prime

***iii. Function With Arguments and Return Values***

· *The function receives data from the calling function and does some process and*

*then returns the result to the called function.*

· *In this way, the main() function will have control over the function. This*

*approach seems better because the calling function can check the validity of data*

*before it passed to the calling function and to check the validity of the result*

*before it is sent to the standard output device (i.e. screen).*

· *When a function is called, a copy of the values of actual arguments is passed to*

*the called function.*

Example : find Factorial of a number.

**FUNCTION PROTOTYPES**

· Any C function returns n integer value by default.

· If a function should return a value other than an int, then it is necessary to mention the

calling function in the called function, which is called as the function prototype.

· Function prototype are usually written at the beginning of the program explicitly

before all user defined functions including the main() function.

The syntax is :

***return\_type function\_name(dt1 arg1, dt2 arg2, . . . dtn argn) ;***

o Where return\_type represents the data type of the value that is returned by

the function and dt1, dt2, . . . dtn represents the data type of the arguments

arg1, arg2, . . . argn.

The data types of the actual arguments must confirm to the data types of

the arguments with the prototype. For example :

**long fact(long num) ;**

o Here fact is the name of the function, long before the function name fact

indicates that the function returns a value of type long. num inside the

parenthesis is the parameter passed to the called function. long before the

num indicates that it is of type long.

**RECURSION**

· In C it is possible for the functions to call itself.

· Recursion is a process by which a function calls itself repeatedly until some specified

condition has been satisfied.

· A function is called recursive if a statement within the body of a function calls the

same function, sometimes called circular definition.

· Recursion is the process of defining something in terms of itself.

· When a recursive program is executed the recursive function calls are not executed

immediately, they are placed on a Stack (Last In First Out) until the condition that

terminates the recursive function.

· The function calls are then executed in reverse order as they are popped off the stack.

· Recursive functions can be effectively used in applications in which the solution to a

problem can be expressed in terms of successively applying the same solution to the

subsets of the problem.

· Simple example

void count\_to\_ten ( int count )

{

/\* we only keep counting if we have a value less than ten

if ( count < 10 )

{

count\_to\_ten( count + 1 );

}

}

int main()

{

count\_to\_ten ( 0 );

}

· Example: Factorial of a number

/\* Source code to find factorial of a number. \*/

#include<stdio.h>

int factorial(int n);

int main()

{

int n;

printf("Enter an positive integer: ");

scanf("%d",&n);

printf("Factorial of %d = %ld", n, factorial(n));

return 0;

}

int factorial(int n)

{

if(n!=1)

return n\*factorial(n-1);

}

**Difference between Recursion and Iteration**

|  |  |
| --- | --- |
| RECURSION | ITERATION |
| Recursive function – is a function that is  partially defined by itself | Iterative Instructions –are loop based  repetitions of a process |
| Recursion Uses selection structure | Iteration uses repetition structure |
| Infinite recursion occurs if the recursion step  does not reduce the problem in a manner that  converges on some condition.(base case) | An infinite loop occurs with iteration if  the loop-condition test never becomes  false |
| Recursion terminates when a base case is  Recognized | Iteration terminates when the loopcondition  Fails |
| Recursion is usually slower then iteration due  to overhead of maintaining stack | Iteration does not use stack so it's faster  than recursion |
| Recursion uses more memory than iteration | Iteration consume less memory |
| Infinite recursion can crash the system | infinite looping uses CPU  cycles repeatedly |
| Recursion makes code smaller | Iteration makes code |

Here, p is the name of a variable that stores the value 10 and is stored in memory in the

address, say 1234. This could be represented as follows :

**/\* Example to demonstrate use of reference operator in C programming. \*/**

#include <stdio.h>

int main(){

int var=5;

printf("Value: %d\n",var);

printf("Address: %d",&var); //Notice, the ampersand(&) before var.

return 0;

}

**Output**

Value: 5

Address: 2686778

**STORAGE CLASS**

· Every variable in C programming has two properties**: type and storage class.**

o **Type** refers to the data type of variable whether it is character or integer or

floating-point value etc.

o **Storage class** determines how long it stays in existence.

· There are 4 types of storage class:

i. automatic

ii. external

iii. static

iv. register

**Automatic storage class**

**Keyword for automatic variable is** auto

· Variables declared inside the function body are automatic by default. These variable

are also known as local variables as they are local to the function and doesn't have

meaning outside that function

· Variable inside a function is automatic by default, keyword auto are rarely used.

· **Example**

void main()

{

auto mum = 20 ;

{

auto num = 60 ;

printf("nNum : %d",num);

}

printf("nNum : %d",num);

}

**Output :**

Num : 60

Num : 20

Check();

Check();

}

void Check(){

static int c=0;

printf("%d\t",c);

c+=5;

}

**Output**

0 5 10

· During first function call, it will display 0. Then, during second function call, variable

*c* will not be initialized to 0 again, as it is static variable. So, 5 is displayed in second

function call and 10 in third call.

· If variable c had been **automatic variable**, the output would have been:

0 0 0

**External storage class**

· External variable can be accessed by any function. They are also known as global

variables. Variables declared outside every function are external variables.

· In case of large program, containing more than one file, if the global variable is

declared in file 1 and that variable is used in file 2 then, compiler will show error.

· To solve this problem, keyword extern is used in file 2 to indicate that, the variable

specified is global variable and declared in another file.

**Example to demonstrate working of external variable**

int num = 75 ;

void display();

void main()

{

extern int num ;

printf("nNum : %d",num);

display();

}

void display()

{

extern int num ;

printf("nNum : %d",num);

}

**Output :**

Num : 75

Num : 75

**Register Storage Class**

**Keywords to declare register variable** register

**Example of register variable**

register int a;

· Register variables are similar to automatic variable and exists inside that particular

function only.

· If the compiler encounters register variable, it tries to store variable in

microprocessor's register rather than memory.

· Values stored in register are much faster than that of memory. In case of larger

program, variables that are used in loops and function parameters are declared register

variables.

Since, there are limited number of register in processor and if it couldn't store the

variable in register, it will automatically store it in memory.

**Register storage classes example**

#include<stdio.h>

int main()

{

int num1,num2;

register int sum;

printf("\nEnter the Number 1 : ");

scanf("%d",&num1);

printf("\nEnter the Number 2 : ");

scanf("%d",&num2);

sum = num1 + num2;

printf("\nSum of Numbers : %d",sum);

return(0);

}



**Static Storage Class**

· The value of static variable persists until the end of the program. A variable can be

declared static using keyword: static. For example:

static int i;

Here, *i* is a static variable.

**Example to demonstrate the static variable**

#include <stdio.h>

void Check();

int main(){

Check();

**POINTERS**

· Pointer is a variable that holds a memory address, usually the location of another variable in memory.

The pointers are one of C’s most useful and strongest features.

· C Pointer is a variable that stores/points the address of another variable. C Pointer is used to allocate memory dynamically i.e. at run time. The pointer variable might be belonging to any of the data type such as int, float, char, double, short etc.

**Syntax : data\_type \*var\_name;**

**Example : int \*p; char \*p;**

· Where, \* is used to denote that “p” is pointer variable and not a normal variable.

1234 🡨Address of the variable p

10 🡨 Value of the variable p

**Uses of POINTERS**

There are number of advantages of using pointers. They are :

1. Pointers increase the speed of execution of a program.
2. Pointers reduce the length and complexity of a program.
3. Pointers are more efficient in handling the data tables.
4. A pointer enables us to access a variable that is defined outside the function.
5. The use of a pointer array to character strings results in saving of data storage space in memory.

**INITIALIZING POINTERS**

· It is always good practice to initialize pointers as soon as it is declared. Since a pointer is just an address, if it is not initialized, it may randomly point to some location in memory.

· The ampersand (&) symbol, also called address operator, is applied to a variable to refer

the address of that variable.

· Initializing pointers can be made to point to a variable using an assignment statement.

The syntax is :

**ptr\_variable = &variable ;**

Here, the address of the variable is assigned to ptr\_variable as its value.

· example :

**ptr = &price ;**

· Will cause ptr to point to price i.e., ptr now contain the address of price.

· A pointer variable can be initialized in its declaration itself.

For example :

int price, \*ptr = &price ;

Is also valid.

**ACCESSING A VARIABLE THROUGH ITS POINTER**

· Accessing the value of the variable using pointer is done by using unary operator \*

(asterisk), usually known as indirection operator. Consider the following statements :

int price, \*ptr, n ;

price = 100 ;

ptr = &price ;

n = \*ptr ;

* + The **first line** declares the price and n as integer variables and ptr as a pointer variable pointing to an integer.
  + The **second line** assigns the value 100 to price and
  + Third line assigns the address of price to the pointer variable ptr.
  + The fourth line contains the indirection operator \*.

· When the operator \* is placed before a pointer variable in an expression (on the right

hand side of the equal sign), the pointer returns the value of the variable of which the

pointer value is the address.

\*ptr returns the value of the variable price, because ptr is the address of price. The \* can

be remembered as value at address. Thus the value of n would be 100.

**C Program to compute sum of the array elements using pointers**

#include<stdio.h>

#include<conio.h>

void main() {

int numArray[10];

int i, sum = 0;

int \*ptr;

printf("\nEnter 10 elements : ");

for (i = 0; i < 10; i++)

scanf("%d", &numArray[i]);

ptr = numArray; /\* a=&a[0] \*/

for (i = 0; i < 10; i++) {

sum = sum + \*ptr;

ptr++;

}

printf("The sum of array elements : %d", sum);

}

**Output:**

Enter 10 elements : 11 12 13 14 15 16 17 18 19 20

The sum of array elements is 155

**ARRAYS AND POINTERS**

· Arrays are closely related to pointers in C programming but the important difference is, a pointer variable can take different addresses as value whereas, in case of array it is fixed.

This can be demonstrated by an example:

#include <stdio.h>

int main(){

char c[4];

int i;

for(i=0;i<4;++i){

printf("Address of c[%d]=%x\n",i,&c[i]);

}

return 0;

}

Address of c[0]=28ff44

Address of c[1]=28ff45

Address of c[2]=28ff46

Address of c[3]=28ff47

· Notice, that there is equal difference (difference of 1 byte) between any two consecutive

elements of array.

**Relation between Arrays and Pointers**

Consider an array:

int arr[4];



In arrays of C programming, name of the array always points to the first element of an

array. Here, address of first element of an array is &arr[0]. Also, arr represents the

address of the pointer where it is pointing. Hence, &arr[0] is equivalent to arr.

· Also, value inside the address &arr[0] and address arr are equal. Value in address &arr[0]

is arr[0] and value in address arr is \*arr. Hence, arr[0] is equivalent to \*arr.

· Similarly,

&a[1] is equivalent to (a+1) AND, a[1] is equivalent to \*(a+1).

&a[2] is equivalent to (a+2) AND, a[2] is equivalent to \*(a+2).

&a[3] is equivalent to (a+1) AND, a[3] is equivalent to \*(a+3).

.

.

&a[i] is equivalent to (a+i) AND, a[i] is equivalent to \*(a+i).

· In C, one can declare an array and can use pointer to alter the data of an array.

//Program to find the sum of six numbers with arrays and pointers.

#include <stdio.h>

int main(){

int i,class[6],sum=0;

printf("Enter 6 numbers:\n");

for(i=0;i<6;++i){

scanf("%d",(class+i)); // (class+i) is equivalent to &class[i]

sum += \*(class+i); // \*(class+i) is equivalent to class[i]

}

printf("Sum=%d",sum);

return 0;

}

**Output**

Enter 6 numbers:

2

3

4

5

3

4

Sum=21

**POINTER AND FUNCTIONS**

· When, argument is passed using pointer, address of the memory location is passed instead of value.

**Program to swap two number using call by reference.**

/\* C Program to swap two numbers using pointers and function. \*/

#include <stdio.h>

void swap(int \*a,int \*b);

int main(){

int num1=5,num2=10;

swap(&num1,&num2); /\* address of num1 and num2 is passed to swap

function \*/

printf("Number1 = %d\n",num1);

printf("Number2 = %d",num2);

return 0;

}

void swap(int \*a,int \*b){ /\* pointer a and b points to address of num1 and

num2 respectively \*/

int temp;

temp=\*a;

\*a=\*b;

\*b=temp;

}

**Output**

Number1 = 10

Number2 = 5

· **Explanation**

o The address of memory location *num1* and *num2* are passed to function and the

pointers \*a and \*b accept those values. So, the pointer *a* and *b* points to address of

*num1* and *num2* respectively.

o When, the value of pointer are changed, the value in memory location also

changed correspondingly. Hence, change made to \*a and \*b was reflected in

*num1* and *num2* in main function.

**DYNAMIC MEMORY ALLOCATION**

· The size of array declared initially can be sometimes insufficient and sometimes more

than required.

· **Dynamic memory allocation** allows a program to obtain more memory space, while

running or to release space when no space is required.

· Although, C language inherently does not have any technique to allocated memory

dynamically, there are 4 library functions under **"stdlib.h"** for dynamic memory

allocation.

**Function Use of Function**

malloc() - Allocates requested size of bytes and returns a pointer first byte of allocated space

calloc() - Allocates space for an array elements, initializes to zero and then returns a pointer

to memory

free() - dellocate the previously allocated space

realloc() - Change the size of previously allocated space.

**malloc()**

· The name malloc stands for "memory allocation". The function malloc() reserves a block

of memory of specified size and return a pointer of type void which can be casted into

pointer of any form.

**Syntax of malloc()**

ptr=(cast-type\*)malloc(byte-size)

o Here, ptr is pointer of cast-type. The malloc() function returns a pointer to an area

of memory with size of byte size. If the space is insufficient, allocation fails and

returns NULL pointer.

ptr=(int\*)malloc(100\*sizeof(int));

o This statement will allocate either 200 or 400 according to size of int 2 or 4 bytes

respectively and the pointer points to the address of first byte of memory.

**calloc()**

The name calloc stands for "contiguous allocation". The only difference between malloc()

and calloc() is that, malloc() allocates single block of memory whereas calloc() allocates

multiple blocks of memory each of same size and sets all bytes to zero.

**Syntax of calloc()**

**ptr=(cast-type\*)calloc(n,element-size);**

o This statement will allocate contiguous space in memory for an array of n

elements. For example:

**ptr=(float\*)calloc(25,sizeof(float));**

o This statement allocates contiguous space in memory for an array of 25 elements

each of size of float, i.e, 4 bytes.

**free()**

· Dynamically allocated memory with either calloc() or malloc() does not get return on its

own. The programmer must use free() explicitly to release space.

**syntax of free()**

free(ptr);

o This statement cause the space in memory pointer by ptr to be deallocated.

Examples of calloc() and malloc()

**A C program to find sum of n elements entered by user. To perform this program,**

**allocate memory dynamically using malloc() function.**

#include <stdio.h>

#include <stdlib.h>

int main(){

int n,i,\*ptr,sum=0;

printf("Enter number of elements: ");

scanf("%d",&n);

ptr=(int\*)malloc(n\*sizeof(int)); //memory allocated using malloc

if(ptr==NULL)

{

printf("Error! memory not allocated.");

exit(0);

}

printf("Enter elements of array: ");

for(i=0;i<n;++i)

{

scanf("%d",ptr+i);

sum+=\*(ptr+i);

}

printf("Sum=%d",sum);

free(ptr);

return 0;

}

**A C program to find sum of n elements entered by user. To perform this program,**

**allocate memory dynamically using calloc() function.**

#include <stdio.h>

#include <stdlib.h>

int main(){

int n,i,\*ptr,sum=0;

printf("Enter number of elements: ");

scanf("%d",&n);

ptr=(int\*)calloc(n,sizeof(int));

if(ptr==NULL)

{

printf("Error! memory not allocated.");

exit(0);

}

printf("Enter elements of array: ");

for(i=0;i<n;++i)

{

scanf("%d",ptr+i);

sum+=\*(ptr+i);

}

printf("Sum=%d",sum);

free(ptr);

return 0;

}

**realloc()**

· If the previously allocated memory is insufficient or more than sufficient. Then, you can

change memory size previously allocated using realloc().

**Syntax of realloc()**

ptr=realloc(ptr,newsize);

Here, *ptr* is reallocated with size of newsize.

#include <stdio.h>

#include <stdlib.h>

int main(){

int \*ptr,i,n1,n2;

printf("Enter size of array: ");

scanf("%d",&n1);

ptr=(int\*)malloc(n1\*sizeof(int));

printf("Address of previously allocated memory: ");

for(i=0;i<n1;++i)

printf("%u\t",ptr+i);

printf("\nEnter new size of array: ");

scanf("%d",&n2);

ptr=realloc(ptr,n2);

for(i=0;i<n2;++i)

printf("%u\t",ptr+i);

return 0;}

**Source Code to Find Largest Element Using Dynamic Memory Allocation**

#include <stdio.h>

#include <stdlib.h>

int main(){

int i,n;

float \*data;

printf("Enter total number of elements(1 to 100): ");

scanf("%d",&n);

data=(float\*)calloc(n,sizeof(float)); /\* Allocates the memory for 'n' elements \*/

if(data==NULL)

{

printf("Error!!! memory not allocated.");

exit(0);

}

printf("\n");

for(i=0;i<n;++i) /\* Stores number entered by user. \*/

{

printf("Enter Number %d: ",i+1);

scanf("%f",data+i);

}

for(i=1;i<n;++i) /\* Loop to store largest number at address data \*/

{

if(\*data<\*(data+i)) /\* Change < to > if you want to find smallest number \*/

\*data=\*(data+i);

}

printf("Largest element = %.2f",\*data);

return 0;

}

**Output**



\*\*\*\*\*\*\*\*\*\*\*\*\*\*