**X-UNIT**

**UNDERSTAND BASICS OF POINTERS**

**10.1 Define Pointer:**

A pointer is a derived data type in C. Pointers **contain memory addresses as their values**. Pointers can be used to access and manipulate data stored in the memory.

 A pointer is a variable , that holds the address of another variable in memory. Suppose, we assign the address of **quantity** to a pointer variable **p.** the link between the variables **p** and **quantity** can be visualized as shown fig: let address of **p** is 5048.

Variable value address

Quantity 179 5000

P 5000 5048

Since, a pointer variable ‘points’ to the another variable, so it gets the name ‘pointer’.

**Advantages of using pointers in C:**

* Pointers are more efficient in handling arrays and data tables.
* Pointers can be used to return multiple values from a function.
* The use of pointer arrays to character strings results in saving of data storage space in memory.
* Pointers increases the execution speed and thus reduce the program execution time.
* Pointers support dynamic memory management.
* Pointers permit references to functions and thereby facilitating passing of functions as arguments to other functions.

**10.2 Illustrate declaration and initialization of Pointers:**

 In C every variable must be declared for its type.Since pointer variable contain address that belong to a separate datatype , they must be declared before use them.

 The declaration of a pointer variable takes the following form:

  **Data\_type \*ptr;**

This tell the compiler 3-things:

1. The asterisk (\*) tells the variable **ptr** is a pointer variable.
2. **Ptr** needs a memory location.
3. **Ptr** points to a variable of type **data\_type**

For e.g.: **int \*p;** /\*integer pointe\*/

 Declares the variable **p** as a pointer that points to an integer data type.

Note: Here, the type **int** refers to the data type of the variable being pointed to by **p** not the type value of the pointer.

Pointer declaration styles: int\* p;

 int \*p;

 int \* p;

**Initialization of Pointers:**

 The process of assigning the address of a variable to a pointer is known as **initialization.** Once a pointer variable has been declared we can use the assignment operator to initialize the variable. For e.g.:

 **int quantity;**

 **int \*p; /\*declaration\*/**

 **p=&quantity; /\*initialization\*/**

we can also combine the initialization with the declaration. That is:

 **int\*p=&quantity;**

Here, the variable **quantity** must be declared before the initialization takes place. We are initializing **p** not **\*p.**

**Note:**

1. care should be taken to **avoid wrong pointer assignment**.

For e.g.: float a, b;

 int x,\*p;

 p=&a; /\*wrong, since we are assigning float to inttype pointer\*/

1. 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.

For e.g.: **int x, \*p=&x; /\*three in one\*/**

1. we can also define a pointer variable with an initial value of NULL or 0(zero).

 **int \*p=null;**

 **int \*p=0;**

1. we can **make the same pointer to pointto different data variables** in different statements. Example:

 int x,y,z, \*p;

 ………

 P=&x;

 ………

 P=&y;

 ………….

 P=&z;

 ……….

1. We can also use **different pointers to point to the same data variable**. For e.g.:

 int x;

 int \*p1=&x;

 int \*p2=&x;

 int \*p3=&x;

**10.3 Illustrate accessing the address of a variable using & operator:** The address of a variable can be determine using operator **&** operator. The **&** immediately preceding a variable returns the address of the variable associated with it. For example, the statement:

 **p = &quantity;**

The following program illustrate the above concept.

|  |
| --- |
| main(){ int x=10; float y=2.00; int z; clrscr(); printf("%d is stored at %u",x,&x); printf("%f is stored at %u",y,&y); printf("%d is stored at %u",z,&z);} |

**10.4 Illustrate accessing a value of a variable through pointer:**

The value of the variable using pointer can be accessed using unary operator \*(**asterisk)** , usually known as the **indirection operator(or) dereferencing** operator. Consider the following statements:

 **int quantity, \*p, n;**

 **quantity =179;**

 **p=&quantity;**

 **n=\*p;**

 The first line declares **quantity** and **n** as integer variables and **p** as a pointer variable to an integer. The second line assigns the value 179 to **quantity** and the third line assigns the address of **quantity** to pointer variable **p.** the fourth line contains the indirection operator that returns the value of the variable **quantity** and assigns it to **n.**

**Note:**

 We cannot access the value stored at the address 5368 by writing \*5368. It will not work.

**Example program:**

|  |
| --- |
| main(){ int x,y; int \*ptr; clrscr(); x=10; ptr=&x; y=\*ptr; printf("value of %d is at %u\n",x,&x); printf("values of %d is at %u\n",\*&x,&x); printf("value of %d is at %u\n",\*ptr,ptr); printf("value of %d is at %u\n",ptr,&ptr); printf("value of %d is at %u\n",y,&y);\*ptr=25; printf("x=%d",x); getch();} |

**10.5 Differentiate between address and de-referencing operator:**

|  |  |
| --- | --- |
| **address operator (&)** |  **de-referencing operator (\*)** |
| * This is called address operator or “reference operator”
 | * This is called “in-direction operator” or “de-referencing operator”
 |
| * It is used to access the address of a variable.

Syntax: **&variable;** | * it is used to access the value of a variable through its pointer.

 Syntax: **\*pointervariable;** |
| * e.g.: int x=10;

printf(“address of x is%u”,&x);orint x=10,\*ptr;ptr=&x;printf(“address of x is%u”,ptr); | * e.g.: int x=10,\*ptr;

 ptr=&x; printf(“address of x is%d”, \*ptr); |
| * it is an unary operator. Normally used with a normal variable
 | * it is also a unary operator , used with a pointer variable
 |

**10.6 Discuss about pointer arithmetic:**

**Rules to perform arithmetic operations on pointers:**

1. a pointer can be incremented. i.e., to point to the next element in the array.

An expression like : **p++;**

 P=p+2;

 P=p+1;

When a pointer is incremented, its value is increased by the ‘length’ of the data type that it points to. This length is called the **scale factor.**

The length of various data types are:

 Char 1byte

 Int 2bytes

 float 4bytes

 long Int 4bytes

 double 8bytes.

1. A pointer can be decremented i.e., to point to the previous element in the array.
2. A constant k can be added or subtracted from the pointer. i.e., to point to the kth element in the array from the current element in forward or backward direction respectively.
3. Two pointers cannot be added or multiplied or divided.

i.e., p1+p2 , p1/p2 or p1\*p2 or p1/3 are illegal.

1. But two pointers can be subtracted. i.e., when we subtracted two pointers, the difference between them will be given.

Note: the subtraction can be done only when the two pointers point to the same object.

1. In addition to arithmetic operations, pointers can also compared using the relational operators. The expressions such as **p1>p2 , p1==p2** and **p1 != p2** are allowed.

Note: any comparison of pointers can be done only when the two pointers point to the same object.

**Pointer Expression :** Pointer variables can be used in expressions .If p1 & p2 properly declared and initialized pointers the following statements are valid.

Y=\*p1 \* \*p2;

 Sum=sum+ \*p1;

 Z=5\* - \*p2 / \*p1;

**10.7 Illustrate precedence of address and de-referencing operators:**

 Both pointer with de-referencing operator \* and address operator & can be used in any arithmetic expression. These two are unary operators. When these two operators are used:

1. **same precedence is given to both**
2. **The associativity is from right to left.**

For e.g.: consider the expression

 **u=&x+ \*(&x);** /\* assume x=4 and the address of x is 65496\*/

 then u=&x+ \*(65496);

 u=&x+ 4 /\* the \* is evaluated first so the value in 65496 address is u=65496+4 /\* 4x2 +65496=65504\*/

 u=65504

**Example program:**

|  |
| --- |
| #include<stdio.h>void main(){ int x,\*ptr; clrscr(); x=4; ptr=&x; printf("address of x=%u\n",&x); printf("value of x=%d\n", \*ptr); printf("address of x= %u + value of x=%d =%u \n",&x,\*(&x),&x+\*(&x)); getch();} |

**10.8 Discuss on pointer comparison and pointer conversion:**

**Pointer comparison:**  pointers can also be compared **using the relational operators.**

For e.g.: the following statements are valid.

 **p1 > p2**

 **or**

 **p1==p2**

 **or**

 **p1!= p2**

Comparisons can be **used meaningfully in handling arrays and strings**.

**Note:**

* Any comparison of pointers that refer to separate and unrelated variables not allowed.

**Example program:**

main()

{

 int a=10,b=15;

 int \*p1, \*p2;

 p1=&a;

 p2=&b;

 if(p1>p2)

 printf("p1 is greater than p2");

 else

 printf("p1 is less than p2");

}

**Pointer conversion (or ) casting :** we cannot assign a pointer of one type to a pointer of another type, although both have addresses as their value. This is known as **incompatibility of pointers.**

We can use explicit assignment between incompatible pointer types by using **cast** operator. For e.g.: **int x;**

 **char \*p;**

 **p=(char \*)&x;**

**note:**

* We need not perform casting for **generic pointer** i.e., **void \*** pointer. A void pointer can be assigned to any pointer without casting. A void pointer is created as follows :

 **void \*vp;**

**Example program:**

main()

 {

 char ch=’A’;

 int a=20;

 float b=20.3;

 void \*vp; /\* declare a void pointer \*/

 vp=&ch;

 printf("value of ch=%c\n", \*(char \*)vp);

 vp=&a;

 printf("value of a=%d\n", \*(int \*)vp);

 vp=&b;

 printf("value of b=%f\n", \*(float \*)vp);

}

**10.9 Illustrate relationship between arrays and pointers:**When an array is declared, the compiler allocated a base address and sufficient amount of storage to contain all the array of elements. The base address is the location of the first element of the array. Suppose we declares as

 int x [5]={1,2,3,4,5};

 The base address of x is 1000. The five elements will be stored as follows.

Element x[0] x[1] x[2] x[3] x[4]

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 |

Value

Address 1000 1002 1004 1006 1008

 Then the name **x** is defined as a constant pointer pointing to the first element, **x[0]** and therefore the value of **x** is 1000.

 x=&x[0]=1000;

 If we declare **p** as an integer pointer, then we can make the pointer **p** to point to the array **x** by the following assignment:

 **int \*p=&x**;

This is equivalent to

 **p=&x[0];**

Now, we can access every value of **x** using **p++** to move from one element to another. The relationship between pointer variable **p** and array variable **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)**

The address of an element is calculated using its index and the scale factor of the data type.

**Syntax : address of arrayname[index]= baseaddress+(index\*scale factor of datatype)**

 Ex: address of x[3]=1000+(3\*2)=1006.

We can use pointers to access array elements. The pointer accessing method is much faster than array indexing.

* To a access the element in one dimensional array the expression is : **\*(p+i) or \*(x+i)**

For e.g.: \*(p+3) gives the value of x[3].

* To a access an element in two-dimensional array the expression is :

 \*(\*(**p+i )+j) or**\*(\*(**x+i )+j)**

For e.g.: \*(p+4+3) gives the value of x[4][3]

**10.10:llustrate accessing array elements using pointers.**

|  |
| --- |
| **Example:** c-program using pointers to compute the sum of all elements stored in one dimensional array.main(){ int \*p,sum=0,i; int x[5]={5,9,6,3,7}; clrscr(); i=0; p=x; /\* intializing pointer variable with base address of x\*/ printf("ELEMENT VALUE ADDRESS\n\n"); for(i=0;i<5;i++) { printf("x[%d] %d %u\n",i,\*p,p); sum=sum+\*p; /\*accessing array element\*/ p++; /\*increment pointer variable to next element\*/ } printf("\n sum= %d\n",sum);}**Example:** c-program using pointers to compute the sum of all elements stored in tw o dimensional array.main(){ int \*p,sum=0,i,j; int x[2][3]={5,9,6,3,7,8};   p=x; /\* intializing pointer variable with base address of x\*/ printf("ELEMENT VALUE ADDRESS\n\n"); for(i=0;i<2;i++) { for(j=0;j<3;j++) { printf("x[%d][%d] %d %u\n",i,j,\*p,p); sum=sum+\*p; /\*accessing array element\*/ p++; /\*increment pointer variable to next element\*/ } printf("\n sum= %d\n",sum);} |

|  |
| --- |
|  |

**10.11: Illustrate use of pointers as function arguments:** We can pass the address of a variable as an argument to a function in the normal fashion.When we pass addresses to a function , the parameters receiving the addresses should be pointers. The process of calling a function using pointers to pass the addresses of a variable is known as ‘**call by reference’.** Call by reference provides a mechanism by which the function can change the stored values in the calling function.

Consider the following program which shows how two locations can be exchanged using their address locations. The function **exchange()**  receives the addresses of the variable **x** and **y**  and exchanges their contents.

|  |
| --- |
| #include<stdio.h>void exchange(int \*a,int \*b); /\*prototype\*/main(){ int x,y; x=100; y=200;  printf("before exchange : x=%d y=%d\n\n",x,y); exchange(&x,&y); printf("after exchang : x=%d y=%d\n\n",x,y);}exchage(int \*a,int \*b){ int t; t=\*a; \*a=\*b; \*b=t;} |

Note points:

1. The function parameters are declared as pointers.
2. The de-referenced pointers are used in the function body.
3. When the function is called, the addresses are passed as actual arguments.

**Functions returning pointers:** A function may also return a pointer to the calling function. Consider the following code:

|  |
| --- |
| int \*larger(int \*, int \*);main(){ int a=10; int b=20; int \*p; p=largest(&a,&b); printf(“%d”,p);}int \*largest(int \*x,int \*y){ if(x>y) return(x); /\*address of a\*/ else  return (y); /\*address of b\*/} |

Note:

The address returned must be the address of a variable in the calling function.

**Pointers to functions:**

 It is possible to declare a pointer to a function, which can then be used as an argument in another function. A pointer to a function is declared as follows:

 **Data\_type (\*fptr) ();**

This tells the compiler that **fptr** is a pointer to a function, which returns type value. The parentheses around **\*fptr** are necessary.

**10.12: Discuss pointer arrays with examples.**

 Array of pointers are very helpful in handling character array with variable length strings.for e.g.: consider the following statement,

 Char name[3][25];

 This says that the name is a table containing 3-names, each with a maximum length of 25 characters. The total storage requirements for the **name** table are 75 bytes.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

If we declare **name** to be an array of 3 pointers to characters, each pointer pointing to particular name

 **char \*name[3]={**

 **“new zealand”, “australia”, “india”**

 **};**

name[0]🡪 **new zealand**

name[0]🡪 **australia**

name[0]🡪 **india**

This declaration allocated only 28 bytes , sufficient to hold all the characters as shown

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| N | E | W  |   | Z | E | A | L | A | N | D | \0 |
| A | U | S | T | R | A | L | I | A | \0 |
| I | N | D | I | A | \0 |

The following statement would print out all the 3-names:

 for (i=0;i<3;i++)

 Printf(“%s”,name[i]);

To access the jth characterin the ith name, we may write as

 **\*(name[i]+j);**

The difference between the notations \*p[3] and (\*p)[3].Since \* has alower precedence than [], \*p[3] declares p as an array of 3 pointers while (\*p)[3] declaares p as apointer to an array of three elements.

Example:

 #include<stdio.h>

main()

 {

 char \*name[3]= {"new zealand","australia", "india"};

 for (int i=0;i<3;i++)

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

}

**10.13 Explain dynamic memory management functions and illustrate with example to use these functions:**

 The process of allocating memory at runtime is known as **Dynamic memory allocation.**

The dynamic memory allocation is done by four functions:

1. malloc.

2. calloc.

3. realloc.

4. free.

**malloc :** A block of memory may be allocated using the function **malloc.** The **malloc** function

allocates request size of bytes and returns a pointer to the first byte of the allocated space. It takes

following form:

 **ptr= ( cast\_type \* )malloc(byte\_size);**

where ptr is a pointer of type cast\_type. The **malloc** returns a pointer to an area of memory

with size byte\_size.

Ex: x= ( int \*)malloc (100\*sizeof (int));

The statement allocates 10 bytes of space for the pointer cptr of type char.

cptr= ( char \*)malloc(10);

We may also use malloc to allocate space for complex data types.

The malloc allocates a block of contiguous bytes.

**Example program:**

 main ()

{

 int i, n;

 int \*ptr;

 printf("Number of elements to be entered:");

 scanf("%d",&n);

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

 printf("Enter %d numbers:"");

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

 {

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

 }

 printf("The numbers entered are: ");

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

 {

 printf("%d ",\*(a+i));

 }

 free( a );

 return(0);

}

**Calloc():**

calloc is another memory allocation function that is normally used for requesting memory

space at run time for storing derived data types. calloc allocates mu;tiple blocks of storage, each of

the same size, and then sets all bytes to zero. The general form of calloc is:

**ptr= ( cast\_type \*) calloc (n, ele\_size);**

The above statement allocates contiguous space for *n* blocks, each of size *ele\_size* bytes.

Ex: p=(int \*) calloc(10,5);

**Example program:**

 main ()

 {

 int i, n;

 int \*ptr;

 printf("Number of elements to be entered:");

 scanf("%d",&n);

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

 printf("Enter %d numbers:"");

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

 {

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

 }

 printf("The numbers entered are: ");

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

 {

 printf("%d ",\*(a+i));

 }

 free( a );

 return(0);

}

**free:**

The **free** function frees previously allocate space. We may release the block of memory by

using **free** function:

**free(ptr);**

Ex: free(p);

ptr is pointer to a memory block, which has already been created by **malloc** or **calloc.** We

should remember two things:

1. It is not the pointer that is being released but rather what it points to.

2. To release an array of memory that was allocated by **calloc** we need only to release the

pointer once. It is an error to attempt to release elements individually.

**realloc:**

realloc is another memory allocation function. The **realloc** function modifies the size of

previously allocated space.

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

Ex: p=realloc(p,20);

This function allocates a new memory space of size *newsize* to the pointer variable **ptr.** If

the function is unsuccessful in locating additional space, it returns **null** pointer and the original is freed.

**Example program:**

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

int main ()

{

 char \*str;

 str = (char \*) malloc(10);

 strcpy(str, "CLANGUAGE");

 printf("String = %s, Address = %u\n", str, str);

 str = (char \*) realloc(str, 15);

 strcat(str, ".COM");

 printf("String = %s, Address = %u\n", str, str);

 free(str);

 return(0);

}