**Pointer Operator (& and \* operator)**:-

int a =7;

The declaration tells the compiler to perform the following tasks

1. Reserve memory space to hold the integer value.
2. Associate the name i with the memory location.
3. Store the value 3 at this memory location.

i location name

7

Value at location

6485 location address

We can say that the computer selected the memory location 6485 to store the value 7. This location is not fixed because next time the computer may select any other memory location to store the value. This location is selected randomly depending upon the available memory space.

Void main()

{

int i = 7;

printf(“\n Address of i=%u”,&i);

printf(“\n Value of i=%d”,i);

printf(“\nValue of i=%d ”,\*(&i));

}

Output:-

Address of i = 6485

Value of i = 7

Value of i =7

& (address of operator):- **(&i)** returns the address of the variable i

\*(Value at address operator): -**\*(&i)** returns the value stored at the location 6485 i.e. the address of i.

**\*(&i) \*(address of i) \*(6485) value at the location 6485 7**

**Pointer:** - Pointer is a variable that can hold the address of another variable of same type.

**Declaring a pointer:-**

**int \*ptr;**

* ptr is the name of the pointer variable.
* The '\*' informs the compiler that ptr is a pointer variable.
* The int says that the pointer variable stores the address of an integer i.e. a pointer is said to "point to" an integer.

**i(pointee) ptr(pointer)**

6485

7

**6485 3276**

Here **ptr** is holding the address of **i** thus **ptr** must be a pointer of integer type.

void main()

{

int i=7;

int \*ptr;

ptr = &i;

clrscr();

printf("\nvalue of i=%d",i);

printf("\naddress of i=%u",ptr);

printf("\naddress of ptr=%u",&ptr);

printf("\nvalue of i=%d",\*ptr);

getch();

}

Output : -

value of i= 7

address of i= 6485

address of ptr = 3276

value of i= 7

**Pointer Dereference: -**

In the last printf statement value of \*ptr becomes 7. This is called pointer dereference. The "dereference" operation follows a pointer's reference to get the value of its pointee. The value of the dereference of \*ptr above is 7.

**Types of pointer**: -

char \*p; 🡪 character pointer

float \*q; 🡪 float pointer

int \*pt; 🡪 integer pointer

double \*k; 🡪 double pointer

All are the pointer capable of holding the address of a variable. Memory addresses are always whole numbers, so pointers always contain whole numbers.

So a **float \*q** does not mean that q is going to hold a floating point number rather q is capable to hold the address of a floating point number.

Eg. float a =3.6;

float \*q;

q=&a;

char \*p 🡪 p is going to hold then address of a character value.

char t =’c’;

char \*p;

q=&t;

**Pointer to a pointer or double pointer**: - when a pointer variable **k** contains address of another pointer variable then **k** is called pointer to a pointer or double pointer. Declared as follows

int \*\*k;

i**(pointee) j(pointer) k(pointer to a pointer)**

3276

6485

7

**6485 3276 7234**

void main()

{

int i=7;

int \*j;

int \*\*k;

j = &i;

k=&j;

clrscr();

printf("\nvalue of i=%d",i);

printf("\naddress of i=%u",j);

printf("\naddress of j=%u",&j);

printf("\nvalue of i=%d",\*j);

printf("\naddress of k=%u",&k);

printf("\naddress of j=%u",k);

printf("\naddress of i=%u",\*k);

printf("\nvalue of i=%d",\*\*k);

getch();

}

Output : -

value of i= 7

address of i= 6485

address of j= 3276

value of i= 7

address of k=7234

address of j=3276

address of i=6485

value of i=7

**Passing Address to a Function**:- in this method addresses of actual arguments are copied into the formal arguments.

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

void main()

{

int x, y;

  printf("Enter the value of x and y\n");

scanf("%d%d",&x,&y);

  printf("Before Swapping\nx = %d\ny = %d\n", x, y);

swap(&x, &y);

  printf("After Swapping\nx = %d\ny = %d\n", x, y);

  }

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

{

int temp;

temp = \*b;

\*b = \*a;

\*a = temp;

}

**Function Returning a Pointer**: - a function can return any value such as int, float, and char; likewise a function can return a pointer. However to make a function return a pointer it has to explicitly mentioned in the called function as well as in the function definition.

int \* fun();

void main()

{

int \*p;

p=fun();

printf(“%u”,p);}

int \* fun()

{

int i =20;

return(&i);

}

Function prototype

int\* fun ( );

Function return type no argument

(int pointer)

Function name

The function prototype tells the compiler that fun() is a function which receives no argument and returns an integer pointer.

**void pointer**: -A void pointer is a pointer that has no associated data type with it. A void pointer can hold address of any type and therefore cannot be dereferenced.

**Declaration of Void Pointer**:

void \* pointer\_name;

void main()

{

void \*ptr; // ptr is declared as Void pointer

char cnum;

int inum;

float fnum;

ptr = &cnum; // ptr has address of character data

ptr = &inum; // ptr has address of integer data

ptr = &fnum; // ptr has address of float data

}

In this example ptr is a void pointer. We have declared 3 variables of integer, character and float type. When we assign address of integer to the void pointer, pointer will become Integer Pointer. When we assign address of Character Data type to void pointer it will become Character Pointer. Similarly we can assign address of any data type to the void pointer. It is capable of storing address of any data type

**Dereferencing a void pointer:-**

 Void pointers cannot be dereferenced. We need to typecast the void pointer variable to dereference it. This is because a void pointer has no data type associated with it. There is no way the compiler can know (or guess?) what type of data is pointed to by the void pointer. Therefore the typecasting is necessary before dereferencing.

void main()

{

int a=10;

void \*ptr; // Declaring a void pointer

ptr=&a; // Assigning address of integer to void pointer.

printf("The value of integer variable is= %d",\*( (int\*) ptr) ); // (int\*)ptr - is used for type casting. Where as \*((int\*)ptr) dereferences the type casted void pointer variable.

}

Output: - The value of integer variable is= 10

**Null Pointer**:- The constant NULL is a special pointer value which encodes the idea of "points to nothing." It represents the idea that a pointer does not have a pointee.

It is always a good practice to assign a NULL value to a pointer variable in case we do not have exact address to be assigned. This is done at the time of variable declaration. A pointer that is assigned NULL is called a null pointer.

The NULL pointer is a constant with a value of zero defined in several standard libraries. Consider the following program:

#include <stdio.h>

int main ()

{ int \*ptr = NULL;

printf("The value of ptr is : %d\n", ptr );

return 0;

}

Ouput:- The value of ptr is : 0

**Pointer and Array**: -

When an array is declared, the compiler allocates a base address and a sufficient amount of storage to hold all the elements of the array in contiguous memory locations. The base address is the location of the first element of the array.

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

Suppose the base address of x is 1000 and assuming that each integer requires 2 bytes (**scale factor of int**)

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

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

Value  
 Address 1000 1002 1004 1006 1008

The name of the array **x** means constant pointer pointing to the first element, x[0]. Value of x is 1000.

x=&x[0]=1000;

If we declare an integer pointer p, then

int \*p;

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

Now p is also points to the same array

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

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

1000 1002 1004 1006 1008

p x

When we increment p it points to the next element of the array.

p= &x[0]=1000

p+1=&x[1]=1002

p+2=&x[2]=1004

p+3=&x[3]=1006

p+4=&x[4]=1008

Addresses are calculated using the following formula

Address of x[i] = base address +(i\*scale factor of data\_type)

Eg.

Address of x[3] = 1000 +(3\*scale factor of int)

Address of x[i] = 1000+3\*2

=1006

int main ()

{

/\* an array with 5 elements \*/

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

int \*p;

int i;

p = x;

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

printf( “Values\tAddress\n");

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

{

printf("%d\t%u", \*p, p) ;

p++;

}

getch();

}

Output:-

Value Address

1 1000

2 1002

3 1004

4 1006

5 1008

**Array of Pointers**: - Like array of int and array of float we can declare the array of pointer. A pointer variable always contains an address thus array of pointer is nothing but a collection of addresses. All te rules that we can apply on ordinary array can be applied onto array of pointers.

Eg.

char \* name[]={“India”,

“New zealand”,

“Australia”

};

**name** is an array of 3 pointers to character each pointing to particular name

name[0] India

name[1] New Zealand

name[2] Australia

this declaration allocates 28 bytes only

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| I | n | d | i | a | \0 |
| N | e | w |  | z | e | a | l | a | n | d | \0 |
| A | u | s | t | r | a | l | i | a | \0 |

void main()

{

char \* name[]={“India”,

“New zealand”,

“Australia”

};

int i;

for(int i=0;i<=2;i++)

{

printf(“%s\n”,name[i]);

}

}