-POINTERS IN C-

Programming Fundamentals

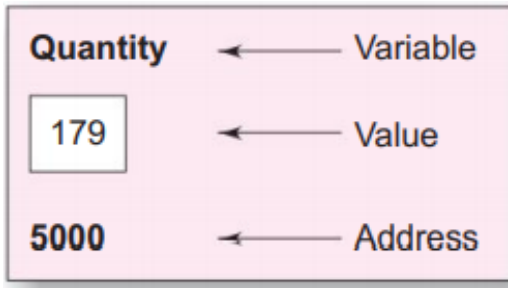
Background

- A pointer is a derived data type in C.
- It is built from one of the fundamental data types available in C.
- Pointers contain memory addresses as their values.
- Pointers can be used to access and manipulate data stored in the memory.
- Pointers allow C to support dynamic memory management.
- Pointer is an efficient tool to manipulate dynamic data-structures like linked list, stack, queue, structures etc.

Whenever we declare a variable, the system allocates a location to hold the value of the variable.

Since, every byte has a unique address, this location will have its own address.

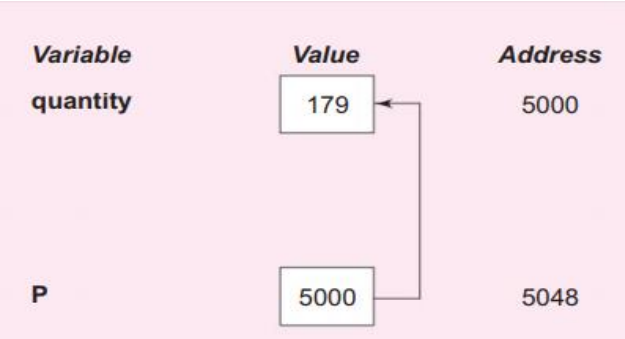
E.g. int quantity = 179;



Address of a variable is address of 1st byte of that variable.

Definition

- Memory addresses are simply numbers, and can be assigned to some variables, that can be stored in memory, like any other variable.
- Such 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.
- E.g.: Here, p is a pointer variable storing address of quantity variable.

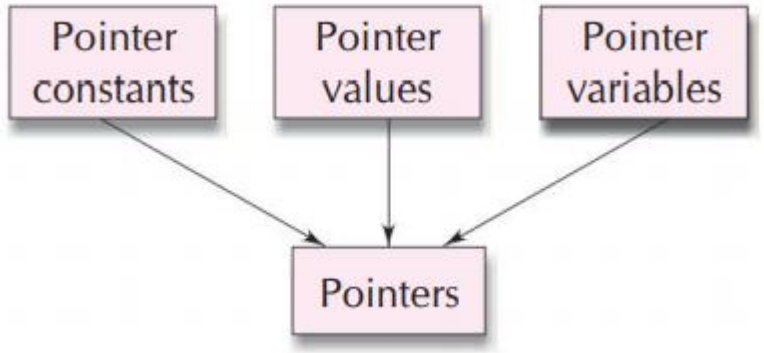


Some terminologies

- There are 3 important terminologies in pointers:

a. **Pointer constants**: memory address within a computer

b. **Pointer values**: We cannot save the value of a memory address directly. We can only obtain the value through the variable stored there using & operator. It may change across different runs.



b. **Pointer variables**: It is a variable that stores pointer value.

ACCESSING THE ADDRESS OF A VARIABLE

- & (ampersand) operator allows accessing the address of a variable.

- E.g. **int var=2; //var means value inside ‘a’ (2)**

var : 1000

printf(“%d”, &var); //&var means 1000

DECLARING POINTER VARIABLES

- In C, every variable must be declared for its type.
- To differentiate b/w normal (data) variable and pointer variable, C enforces to declare pointer variable with a *(unary star operator/asterisk).
- Syntax:

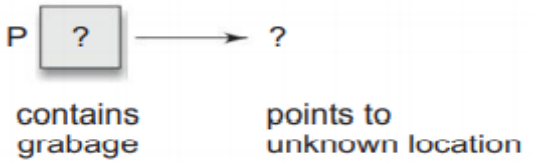
***data_type* *pt_name;**

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.:

1. int *p; //p is a pointer variable that holds address of an integer datatype



2.float *q; // q is a pointer variable that holds address of a float datatype

- Valid declarations:

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

INITIALIZATION OF POINTER VARIABLES

- The process of initializing address of a variable to a pointer variable is called initialization.
- 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 otherwise compiler will give erroneous output because we are trying to assign the address of a different datatype.
- E.g.: int a=5;

  float *b=&a; //wrong as pointer variable b expects address of float datatype but we are assigning address of int datatype.

ACCESSING A VARIABLE THROUGH ITS POINTER

- Once a pointer has been assigned the address of a variable, we can access the pointer variable using *(asterisk)
- *: indirection operator/dereference operator
- E.g.:

```
int quantity, *p, n;
quantity = 179;
p = &quantity;
n = *p;
```

See line: n=*p;

Here *p indicates value at (address stored at p: which is address of variable quantity).

*p and quantity both will produce value 179.

Other example:

int a; *b;

a=5;

b=&a;

int n= *b;

OR

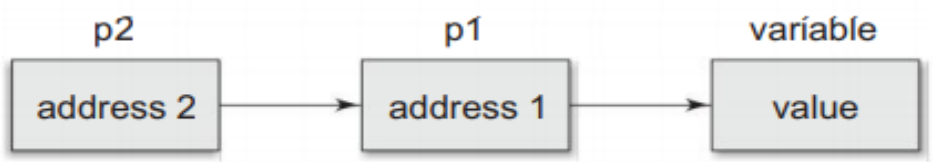
n=*&a;

OR

n=a; //all three are same

CHAIN OF POINTERS

- It is possible to make a pointer to point to another pointer, thus creating a chain of pointers.
- E.g.:



Here, variable is a data-variable,

p1 is a pointer variable storing address of variable, and

p2 is a pointer variable storing address of pointer variable p1.

Declaration looks like this:

int variable; *p1; **p2,*p3;**

POINTER EXPRESSIONS

- Like data variables, pointer variables can be used in expressions.
- E.g.: int a=1, *b=&a, **c=&b;

```
int res = a + *b + **c; //3
**c = **c+2; //3 i.e. value of 'a'
a = a + *b; //3+3=6
```

Note: int division_res = **c**/*b (wrong: as in C, /* indicates start of multi-line comment)**

int division_res = a**/ *b (Correct: introduce space between / and *)**

- C allows us to add integers to or subtract integers from pointers, as well as to subtract one pointer from another.

E.g.

int *p1,*p2;

int *p3=p1 + 4;

p2= p2–2;

If p1 and p2 are both pointers to the same array, then p2 – p1 gives the number of elements between p1 and p2.

E.g.: int arr[5]={1,2,3,4,5};

int *p1= &arr[0] OR arr, *p2=&arr[4] OR arr+4;

int num_ele = **p2-p1; //number of ele b/w p2 and p1**

// (1020-1000)=20/4=5 (consider base address as 1000)

Other examples:

- p1++ and -- p2; (increment/decrement)
- sum += *p2; (arithmetic)
- p1 > p2, p1 = = p2, and p1 != p2 (relational operators)

- We may not use pointers in division or multiplication, or addition.
- E.g.: p1 / p2 or p1 * p2 or p1 / 3 (not allowed)

POINTER INCREMENTS AND SCALE FACTOR

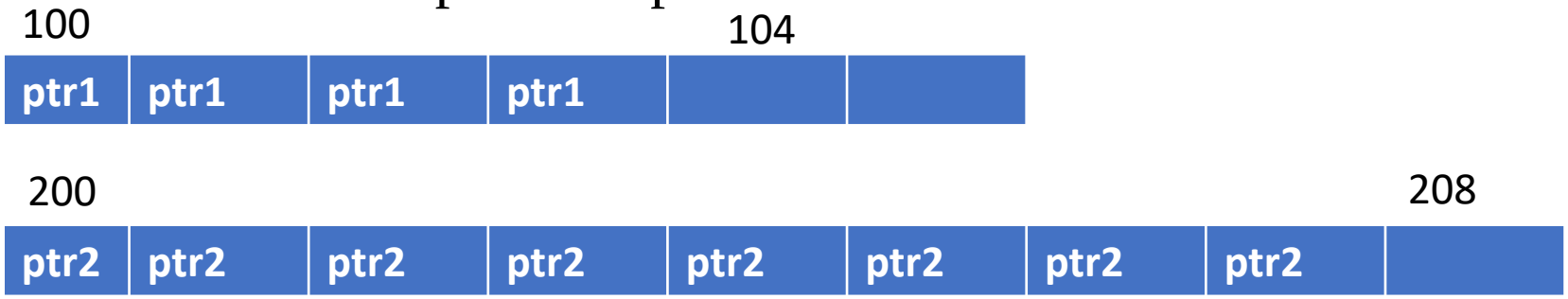
- When a pointer variable is incremented/decremented, its value is inc/dec by the length of datatype that is called scale factor.
- E.g. int *ptr1;

  double *ptr2;

  ptr1=ptr1+1;

  ptr2++;

Effect of increment on ptr1 and ptr2:



RULES FOR POINTER OPERATIONS

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 work with ++ or - - operator.
5. An integer value may be added or subtracted from a pointer variable (scale factor).

6. When two pointers point to the same array, one pointer variable can be subtracted from another (provides #elements b/w these two addresses)

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.

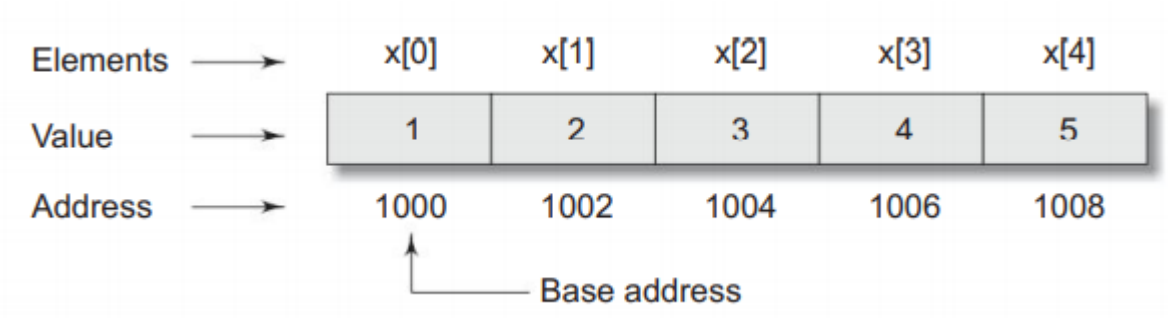
9. Two pointer variables can not be added.

10. A value cannot be assigned to an arbitrary address (i.e., &x = 100; is illegal).

POINTERS AND ARRAYS

- When an array is declared, compiler allocates base-address and sufficient contiguous memory location to hold entire array.
- Base address: address of 1st element/0th index.

E.g.: int x[5]= {1,2,3,4,5}; //scale factor: 2



- Array name (say ‘x’ as per last example) represents address of 0^{th} index element. So, x and &x[0] are same.
- E.g. int *p;

  p = &x[0];

To access elements of the array using pointer variable, we can use ++.

E.g.:

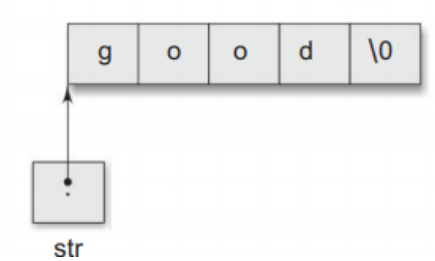
```
p = &x[0] (= 1000)
p+1 = &x[1] (= 1002)
p+2 = &x[2] (= 1004)
p+3 = &x[3] (= 1006)
p+4 = &x[4] (= 1008)
```

Also, address of x[i] = (base address) + (i x scale factor of datatype)

= (1000) + (3 x 2) = 1006

POINTERS AND CHARACTER STRINGS

- Strings are internally treated as character arrays.
- Declaration string using array notation: *char string [5] = "good";*
- Declaring string using pointers: *char *str = "good";*



*char *str = "good";* //not a string copy, because the variable str is a pointer, not a string

We can print the content of the string str using either printf or puts: *printf("%s", str);*

puts (str);

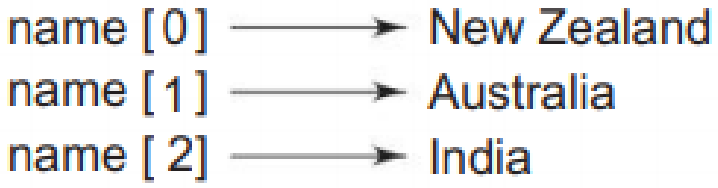
ARRAY OF POINTERS

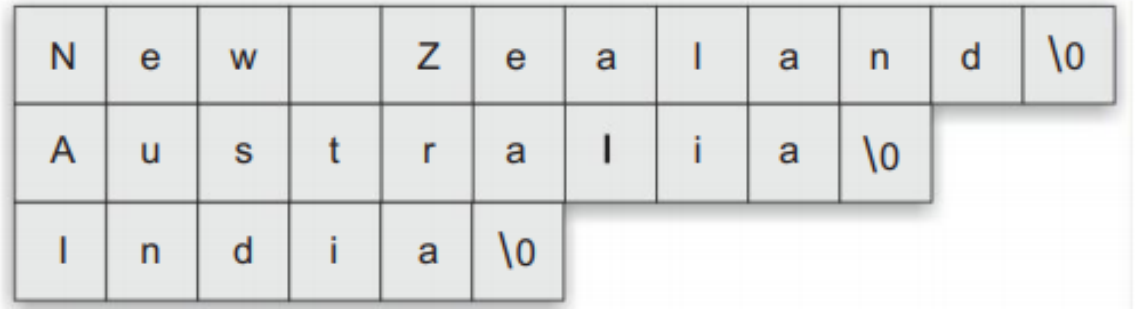
- Consider declaration of an array of strings.

Way-1: *char name [3][25];*

Way-2: *char *name[3] = { "New Zealand", Australia", "India" };*

//Here, name is an array having three elements where every element is a pointer to character.





- Difference between array of pointers and pointer to an array :

Note: Precedence of * is lower than []

int *p[3]; //p is an array of 3 elements where every element is a pointer to an int.

int (*p)[3]; //p is a pointer to an array of 3 elements

POINTERS AND STRUCTURES

- *ptr: ptr is a pointer to struct inventory
- Let's assign address of 0^{th} struct element to ptr

```
ptr = product;
```

```
struct inventory
{
        char    name[30];
        int     number;
        float   price;
}       product[2], *ptr;
```

- We can access individual struct elements using → (arrow operator (also known as member selection operator)

`ptr -> name`	(*ptr).name
`ptr -> number`	(*ptr).number
`ptr -> price`	(*ptr).price

Iterating through an array of structure:

```
for(ptr = product; ptr < product+2; ptr++)
        printf ("%s %d %f\n", ptr->name, ptr->number, ptr->price);
```

We can also use `(*ptr).number` against ptr→number (same for other data-members)

POINTERS AS FUNCTION ARGUMENTS

- When we pass addresses to a function, the parameters receiving the addresses should be pointers.
- **Call by value**: passing value of variable inside a function call.
- **Call by reference**: passing address of a variable inside a function call.
- When call by reference is made, changes are made to the actual variables (arguments from the function call).
- Pointer variable is required in function definition to hold the address of variable.

- Example:

```
main()
{
        int x;
        x = 20;
        change(&x);   /* call by reference or address */
        printf("%d\n",x);
}
change(int *p)
{
        *p = *p + 10;
}
```

Actual arguments- arguments passed from calling function. (x in main)
Formal arguments- arguments in function definition to store copy of value passed from calling function

Here, calling function: main(); called function: change().

At the time of function call to change(), address of variable ‘x’ is passed (&x)

At the time of function definition, address of variable ‘x’ is received in pointer variable ‘*p.

*p= *p+10 (value at address of p i.e. x is getting modified to 20+10=320)

FUNCTIONS RETURNING POINTERS

- A function can return a single value by its name or return multiple values through pointer parameters (as pointers directly operate on actual arguments).
- Since pointers are a data type in C, we can also force a function to return a pointer to a calling function.
- So, return type should be **<datatype>***
- E.g. ***int * sum(int a, int b);***

Here, sum is a function having two integer arguments ‘a’ and ‘b’ and returning an int*.

POINTERS TO FUNCTIONS

- A function, like variable, has a type and an address location in the memory.
- It is possible to declare a pointer to a function, which can then be used as an argument in another function.
- Syntax: `type (*fptr) ();`

Here fptr is pointer to a function, where return type of function is ‘type’

E.g. **int (*sum) (int a, int b);**

Here sum is a pointer to a function that receives ‘a’ and ‘b’ as arguments and returns an int.

Pointer to a function vs function with pointer return type

- Remember **type (*fptr) ();** and **type *fptr();** are different.
- The later will declare a function named fptr that has return type as type*.
- Say, int (*sum)(int a, int b);

- sum is a function with two int arguments a and b that returns an integer pointer

Assigning function to a pointer

- It is possible to make a function pointer point to a specific function.

E.g. **int sum(int, int);** //a function named sum having 2 int arguments and single return value-int

int (*p1)(); //p1 is a pointer to a function having return type as int

p1 = sum; //p1 points to function sum

We can call function sum as follows:

1. sum(2,3); //using function name
2. (*p1)(2,3); //using pointer to a function

Compatibility and Casting

- A variable declared as a pointer is not just a pointer type variable.
- A pointer always has a type(int, char etc) associated with it.
- We cannot assign a pointer of one type to a pointer of another type. This is called incompatibility of pointers.
- All the pointer variables store memory addresses, which are compatible, but what is not compatible is the underlying data type to which they point to. (way of accessing, manipulation, operation)
- We cannot use the assignment operator with the pointers of different types. We can however make explicit assignment between incompatible pointer types by using cast operator.

Void/Generic Pointer

- E.g. int x; char *p; p = (char *) & x;
- Exception: generic pointer/void pointer
- Generic/void pointer: can represent any pointer type
- All pointer types can be assigned to a void pointer and a void pointer can be assigned to any pointer without casting.
- Syntax: void *<name-of-void-pointer>;
- E.g. void *vp;
- As void pointer is not having any object type, it can not be dereferenced.