Infosys®

POWERED BY INTELLECT
DRIVEN BY VALUES

C Programming

Pointers in C

Education and Research

Unit Plan

- Introduction to pointers.
- Need for pointers.
- To illustrate pass by value and pass by reference.
- Pointer Arithmetic.
- Pointer to pointer.

Unit Plan (Contd…)

- Relation between pointers and arrays.
- Array of pointers.
- Dynamic memory allocation.
- Allocation of 2 dimensional arrays dynamically.
- Function pointers.

Introduction to Pointers

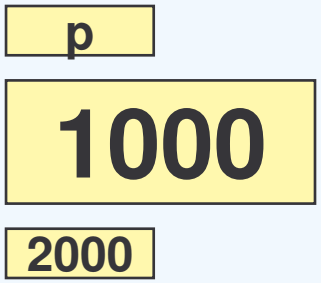
- C compiler assigns a set of memory locations to each variable.
- A variable is a *name* given to a set of memory locations allocated to it.
- For every variable there is an address, the starting address of its set of memory locations.

Introduction to Pointers

- A pointers is a variable that holds the memory address of other variable.
- If a variable called p holds the address of another variable q then p is said to point to q and p is called as a pointer variable.

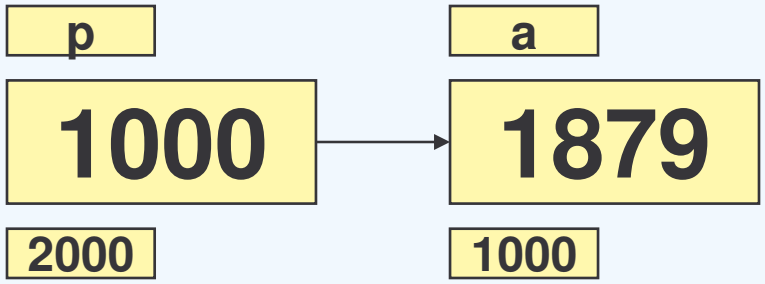
Introduction to Pointers (Contd…)

- To Declare a pointer variable, use the following syntax
  - type * var-name;
  - Here type is the base type of the pointer and the * operator preceding the var-name specifies that the variable is a pointer variable.
  - int * p; This statement declares a variable of type int pointer.



Introduction to Pointers (Contd…)

- To store the address of a integer into a integer pointer, use the following syntax
  - p=&a;
  - Here a is a integer variable and its address is extracted by using the **&** operator and assigned to p, which is an integer pointer. **&** operator is called as the “address of” operator



Introduction to Pointers (Contd…)

- To access the ***value at the address*** stored in the pointer, use the following syntax
  - printf(“%d”,*p);
  - Here the * operator preceding p will fetch the value at the address stored in p.

Need For Pointers

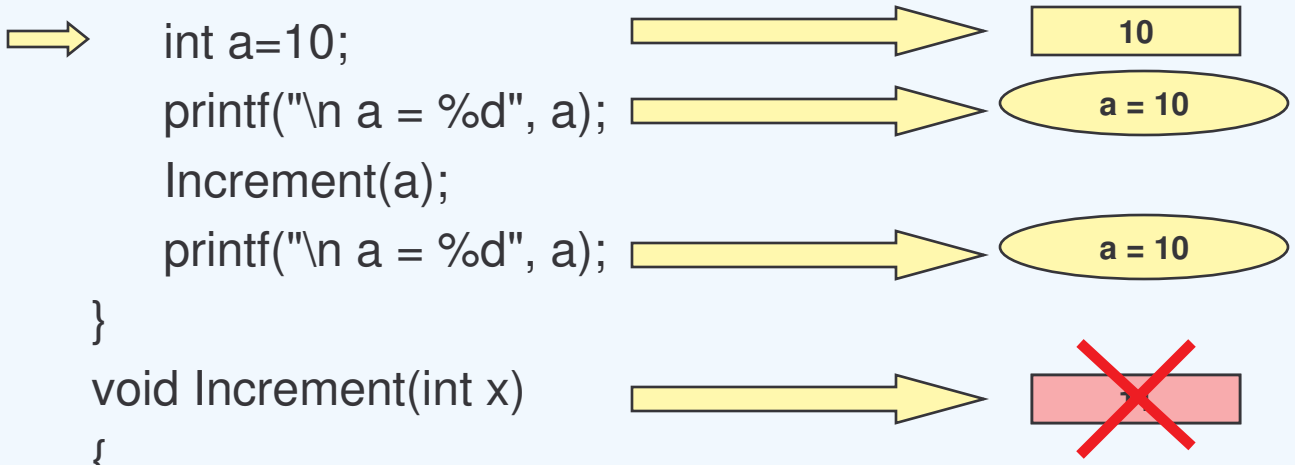
- Pass information to functions and return information from functions.
  - A variable declared within a function cannot be accessed in some other function.
  - To achieve this the caller function needs to pass the address of the variable to the called function.
  - A function can return only one value at a time. Incase we need to return multiple values to the caller function, the caller function should pass the address of variables declared in it so that the called function can write to these variables serving the purpose of multiple return values to the calling environment.

Need For Pointers (Contd…)

- Storing addresses of functions.
  - In certain situations we need to pass functions as arguments to another function.
  - This can be achieved by passing the address of the function.
- Dynamic Memory allocation.
  - If memory is allocated at runtime its starting address need to stored in the program so that we can access the entire memory based on its starting address.

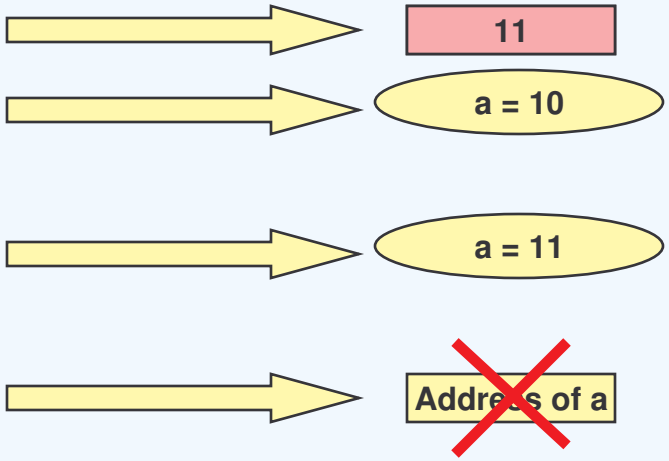
Pass By Value

```
void main()
{
    int a=10;
    printf("\n a = %d", a);
    Increment(a);
    printf("\n a = %d", a);
}
void Increment(int x)
{
    x++;
}
```



Pass By reference

```
void main()
{
    int a=10;
    printf("\n a = %d", a);
    Increment(&a);
    printf("\n a = %d", a);
}
void Increment(int *x)
{
    (*x)++;
}
```

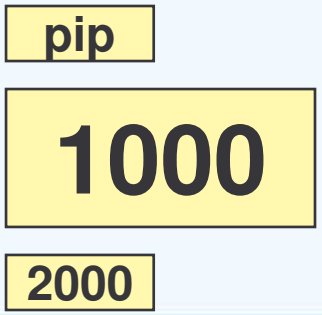


Pointer Arithmetic

- Size of a pointer variable is always 4 bytes (irrespective to its type) i.e. a integer pointer and a character pointer will always be of the same size.
- Adding a integer i to a pointer will increment its value by x * i, where x is the size of the base type of the pointer.
- Subtracting a integer i from a pointer will decrement its value by x * i, where x is the size of the base type of the pointer.

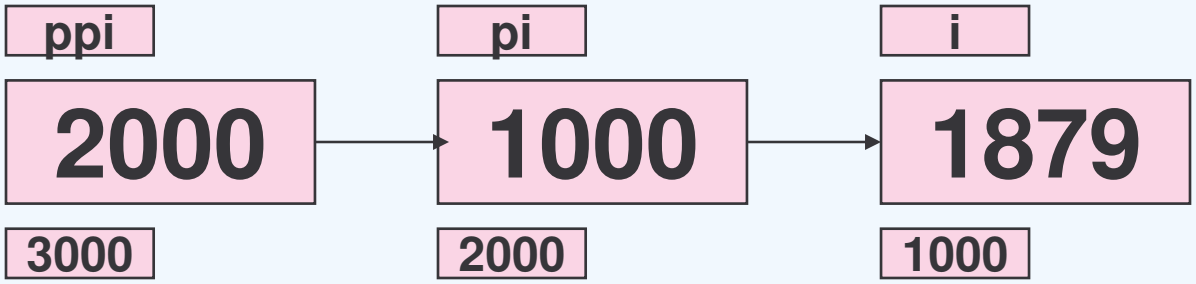
Pointer to a Pointer

- Pointer variables are also allocated memory hence they also have some address.
- The variable which stores the address of a pointer variable is termed as a pointer to pointer.
- To Declare a pointer to a integer pointer use the following syntax
  - int ** pip;



Pointer to a Pointer

- To extract the address of a ***integer pointer*** in a ***pointer to an integer pointer,*** use the following syntax.
  - int *pi;
  - int **ppi=π
- To Access the value of i using ppi use the following syntax
  - printf("\n The value of i is %d", **ppi);



Pointers & Arrays

- Pointers & Arrays are closely related to each other.
- The name of the array is the address of the zeroth element of the array. Hence it implies that the name of the array acts like a pointer to the first element in the array
- Arrays elements are allocated memory in a contiguous fashion.

Pointers & Arrays (Contd…)

- Arrays make use of pointers internally to navigate through the array.
- The statement a[1] refers to the first element of the array.
- Internally the compiler adds a integer to the name of the array (address of the zeroth element) to reach the first element of the array.
- a[i] is converted to *(a+ (i * size of the integer))
- The expression in the parenthesis will evaluate the address of ith element in the array
- The * operator will extract the value of the ith element.

int A[5];
Address of the Zeroth element is 1000

10	20	30	40	50

Array of pointers

- Array of pointers is a collection of pointers which are of similar type (ex. collection of integer pointers or character pointers)

- To Declare a array of integer pointers use the following syntax.
  - int* a[5];// declares an array of 5 integer pointers.

  - Each element in the array is a integer pointer and can point to an integer variable.

  - Initially the elements of the array are having garbage values which can be initialized with the address of the integer variables.

Dynamic Memory Allocation

- Arrays in C have the following draw back
  - We need to specify the size of the array while declaring the array and its not always possible to predict the size requirements of the array before its usage.
  - Once the array is declared we cannot increase/decrease its size at runtime.
- In certain circumstances, allocation of memory should happen at runtime based on inputs from the user.

Dynamic Memory Allocation (Contd...)

- malloc() is the standard library functions used to allocate memory at runtime.
- malloc() accepts the size of the memory block to be allocated at runtime.
- malloc() allocates memory on the fly from the heap and returns the starting address of the memory block allocated.
- The address returned by malloc() is in the form of a void pointer which needs to be typecasted to the appropriate pointer type.
- If case of insufficient memory on the heap, malloc() returns a NULL.

Dynamic Memory allocation (Contd…)

- calloc () function is similar to malloc() but the only difference is that it accepts 2 integers, first one specifies the no of elements and second one specifies the size of each elements.
- calloc(10,4) will allocate a array of 10 integers where each integer occupies 4 bytes.
- calloc() returns a void pointer which needs to be typecasted to the appropriate pointer type.

Dynamic Memory allocation (Contd…)

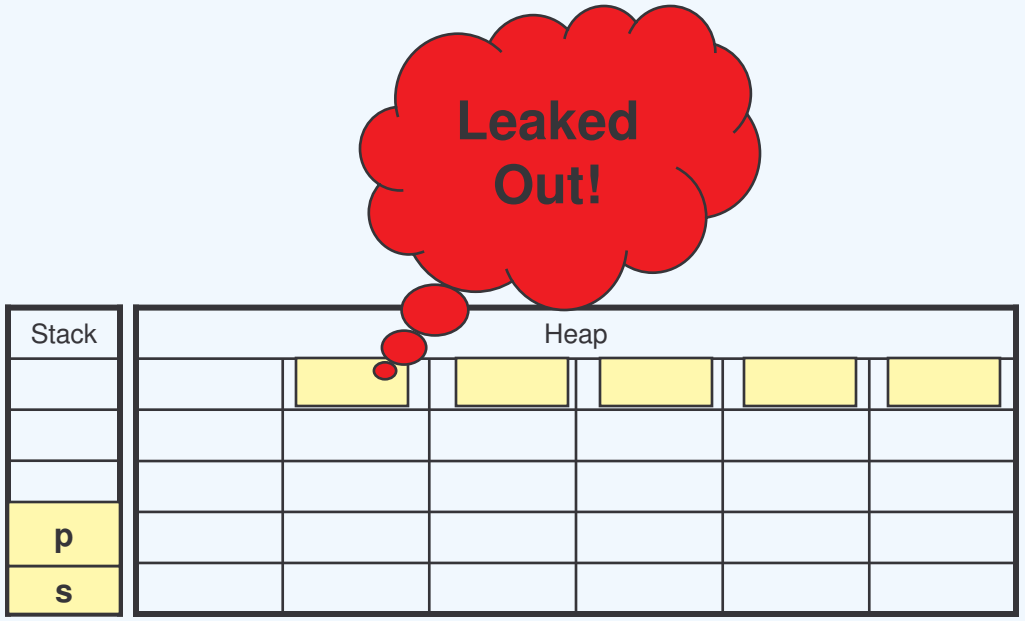
- realloc() function is used to expand/ shrink the memory allocated by malloc() earlier.
- realloc() functions accepts 2 arguments, first one specifies pointer to previously allocated memory block and second one specifies the new size in bytes.
- realloc() returns the base address of the newly allocated block of memory. It also copies the values from the old block to the new block.

Dynamic Memory allocation (Contd…)

- Memory allocated by malloc() needs to be given back to the system if we no longer need it, so that the system can allocate this memory to some other program.
- free() is the standard library functions to release the memory.
- free() accepts a void pointer, which holds the starting address of the memory block allocated by malloc().
- Attempting to free an invalid pointer (a pointer to a memory block that was not allocated by **calloc or malloc**) is illegal and the results are unpredictable.

Memory Leak

```
int main(int argc, char* argv[])
{
        Allocate(5);
        /* some code*/
        return 0;
}
void Allocate(int s)
{
        int *p;
        p=(int*)malloc(s*sizeof(int));
        /* Use the array */
}
```



2D Arrays

- A 2D array of size M *N can be considered as a collection of M 1D arrays, each of size N. (M is the number of rows and N is the number of columns in the 2D array)

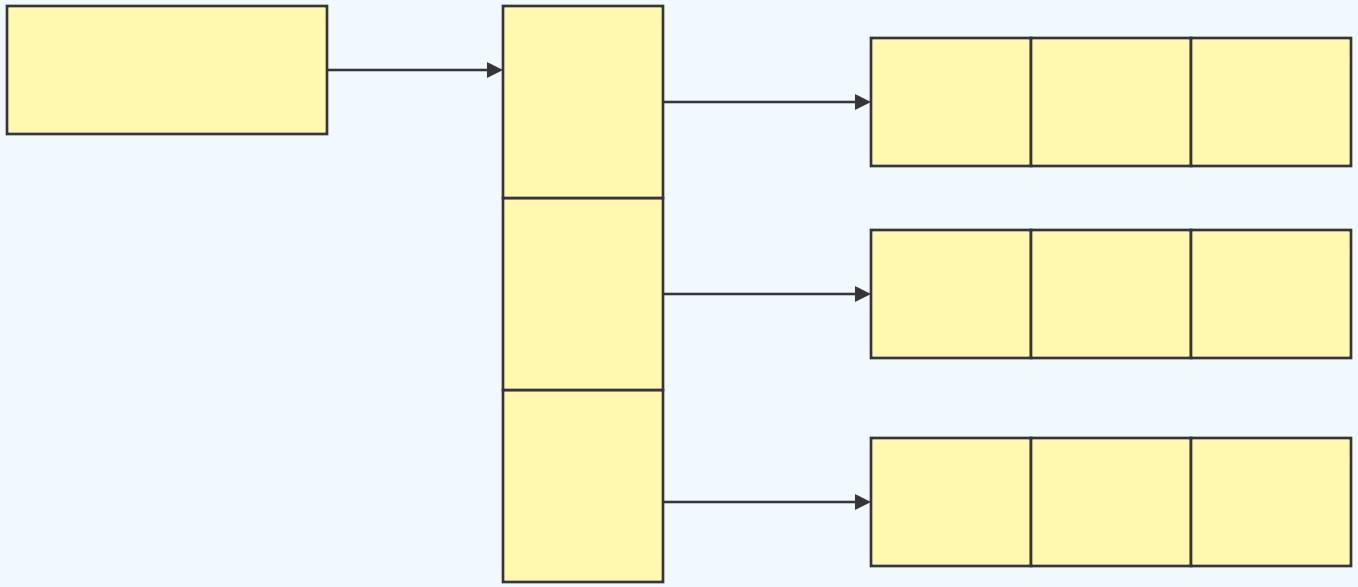
- Allocating 2 D arrays, of size M*N at runtime needs to be done in 2 steps.

  1. Allocate a array of size **M** using malloc(), which will be used in step 2. Store the base address of this array in a pointer variable. This will act as the **array name**.

  2. Allocate **M** one dimensional arrays each of size **N** using malloc() and store their base addresses in the array allocated in step 1.

2D Arrays (Contd...)

Illustration of dynamic creation of a 2D array



2D Arrays (Contd…)

- Once the 2D array is allocated its elements can be accessed using the usual notation
  - To access the 2nd element in the 2nd row of the array use the following syntax a[2][2]=10, where a is the array name.

2D Arrays (Contd…)

- Releasing 2 D arrays, of size M*N at runtime needs to be done in 2 steps.
  - Release the **M** one dimensional arrays each of size **N** using free()
  - Release the array of size **M** using free(), which was used to store the base addresses of the **M** arrays released in step 1

Pointers to functions

- A function pointer is a variable that contains the address of the entry point to a function.
- When the compiler compiles a program it creates the entry point for each function in the program
- The entry point is the address to which execution transfers when a function is called.

Pointers to functions (Contd...)

- Please refer to Notes page for more explanation on previous slide

Pointers to functions (Contd…)

- Declaring a function pointer
  - int Add (int, int); //function Add accepts 2 integers and return a integer.
  - int (*pAdd)(int, int); //pAdd is a pointer to a function which can only point to those functions which accept 2 integers and return a integer
  - pAdd=Add; //Assign the address of Add to pAdd.
  - pAdd(10,20); //call to Add() function using pointer to function.

Pointers to functions (Contd…)

- Passing function pointers as arguments to a another function
  - It is very common to pass function pointers as arguments to other functions.
  - The function to which the pointer is passed will call the function using the pointer variable and not the function name.
- Prototype of a function which accepts a function pointer. The function pointer is such that it can point to any function which accepts a integer and returns a void
  - void F (*void (*pf) (int)*);

Pointers to functions (Contd…)

- Declaring a array of function pointers
  - int Add (int, int); //function Add accepts 2 integers and return a integer.
  - int Sub (int, int); //function Sub accepts 2 integers and return a integer.
  - int (*pf[4])(int, int); //pf is a array of pointers to a functions.

Pointers to functions (Contd…)

- Initializing the array elements with functions addresses.
  - pf[0]=Add; //Assign the address of Add() to the zeroth element of the array.
  - pf[1]=Sub; //Assign the address of Sub() to the first element of the array.

Pointers to functions (Contd...)

- Calling the functions using function pointers stored as array elements.
  - int result = pf[0](10,20); //This statement will call function Add()
  - int result = pf[1](10,20); //This statement will call function Sub()

Summary

- Topics covered in this unit
  - Need for pointers, intro to pointers, * and & operator, pass by value/ref, pointer arithmetic, pointer to pointer, array of pointers, relation between arrays and pointers, dynamic memory allocation & pointers to functions.

Thank You!