Practical-10

To study about pointers and dynamic memory management

Variable

- If you have a variable var in your program, &var will give you its address in the memory.

```
int main()
    {           int a= 5;
                printf(“a: %d\n", a); // Notice the use of & before var
                printf("address of a: %p", &a);
                return 0;
          }
```

Pointer

- Pointers (pointer variables) are special variables that are used to store addresses rather than values.
- A pointer variable (or pointer in short) is basically the same as the other variables, which can store a piece of data.
- Unlike normal variable which stores a value (such as an int , a double , a char), a pointer stores a memory address.

Pointer Syntax

- **Declaration**

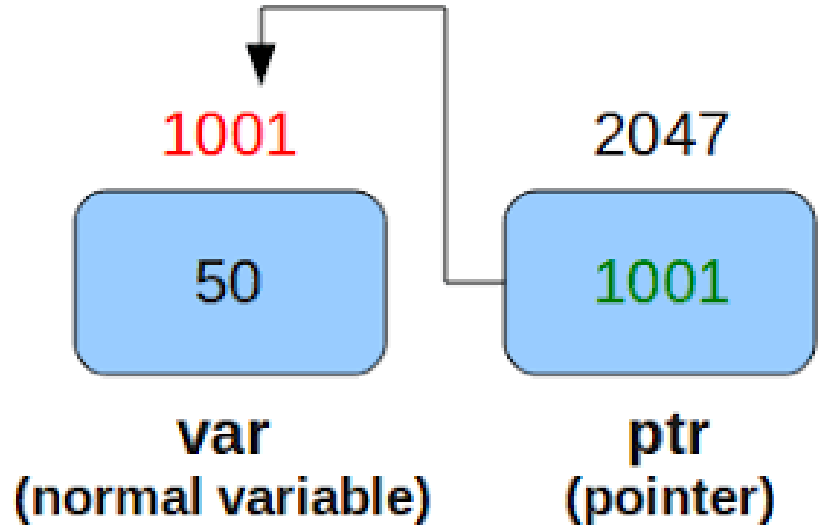
  datatype * variablename

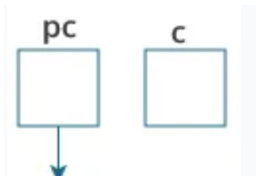
- **Example**

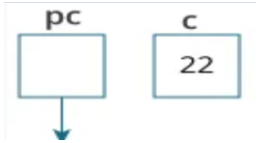
  int *p; int * p; int* p;

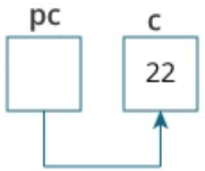
- **Assigning addresses to Pointers**

```
int* pc, c;

c = 22;

pc = &c;
```









Get Value Pointed by Pointers

➢ * Used to get the value stored in that address

```
int* pc, c;

c = 5;

pc = &c;

printf("%d", *pc);
```

- pc is a pointer, not *pc.
- We cannot write like *pc = &c
- * is called the dereference operator (when working with pointers).
- It operates on a pointer and gives the value stored in that pointer.

Common Mistakes

- int c, *pc;
- pc = c; // Error // pc is address but c is not
- *pc = &c; // Error // &c is address but *pc is not
- int *p = &c; //correct //means int *p and p=&c
- pc = &c; // both &c and pc are addresses
- *pc = c; // both c and *pc values

Practical-10.1

- Write a program to print an address of a variable.

- Int *p,x;
- P=&x;
- printf("%**d** is stored at address %u. \n", x, &x)
- printf("%**d** is stored at address %u. \n", *p, p)

Practical-10.2

- Write a program to change the value of a variable using pointer.

Changing Value Pointed by Pointers

- int* pc, c;
- c = 5;
- pc = &c;
- c = 10;
- printf("%d", c);
- printf("%d", *pc);

- int* pc, c;
- c = 5;
- pc = &c;
- *pc = 1;
- printf("%d", *pc);
- printf("%d", c);

Practical-10.3

- Write a program using pointer to read in the array of integers and print its elements in reverse order.
    1. Start
    2. Declare array and pointer variable
    3. Assign address of array to pointer variable //p=&a[0] equal to p=a
    4. For loop – read value from location pointed by P //(p+i)
    5. Print values –for loop-*(p+i)
    6. End

Practical-10.4

- Write a program using pointer to find the length of the character string.

Practical-10.4

- Algorithm
  1. Start
  2. Declare char array ,char pointer variable and len=0
  3. Assign address of array to pointer variable //p=&a[0] equal to p=a
  4. while char value(*p) !='\0'
  5. Count len++ and p++
  6. print
  7. End

Practical-10.5

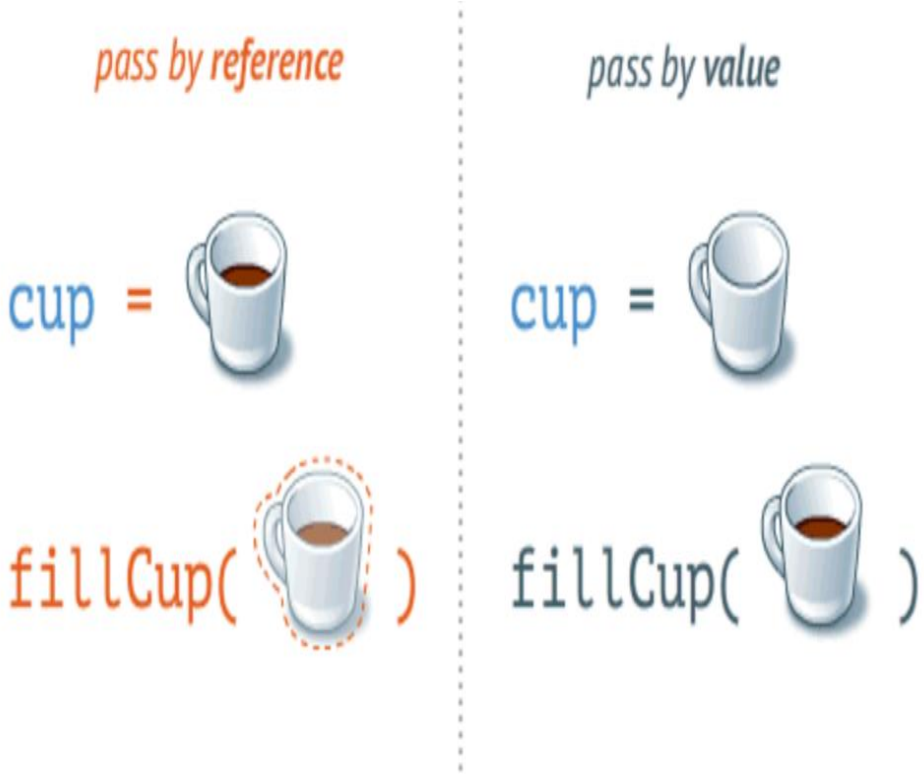
Explain the difference between “call/pass by value” & “pass by address” with a example.

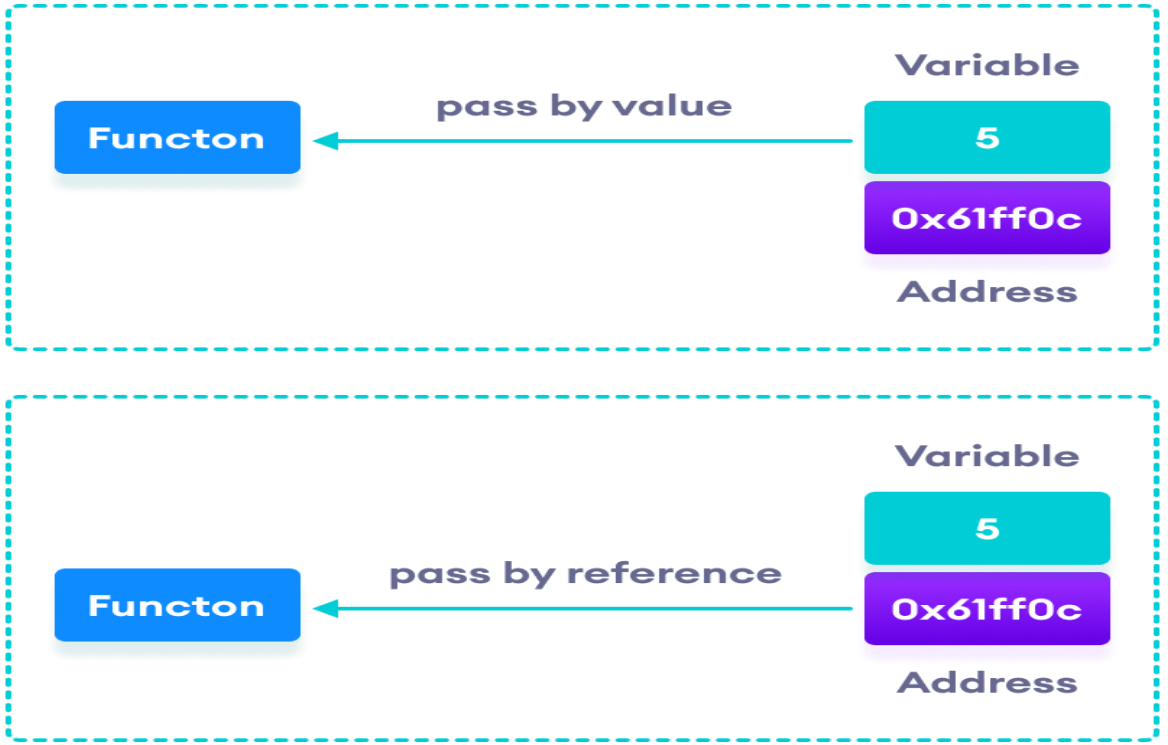
Pass By Value Vs Pass By Reference

- **Pass By Value**: In **Pass by value**, function is called by directly **passing** the **value** of the variable as an argument.
- So any changes made inside the function is made to the copied value not to the original value

- **Pass by Reference:** In Pass by Reference, Function is called by directly passing the reference/address of the variable as an argument.
- So changing the value inside the function also change the original value

Pass By Value Vs Pass By Reference





Example

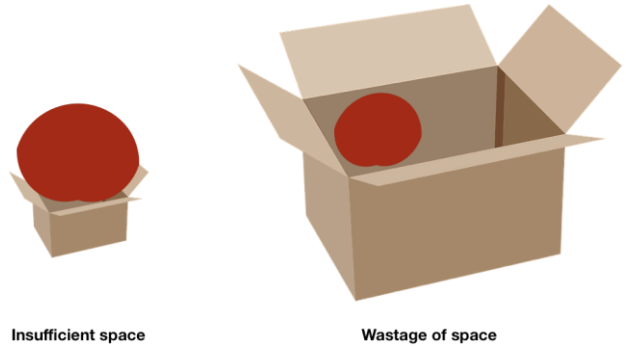
- void swapx(int*, int*);
- int main()

```
  {
        Read values a and b;
        swapx(&a, &b);  print a and b;
   }
void swapx(int*x, int*y)
   {
Int t;t=*x;*x=*y;*y=t;
      swap value using temp
   }
```

Practical-10.6

- Write a program that uses a table of integers whose size will be specified interactively at run time.

Dynamic Memory Allocation



- C is a structured language; it has some fixed rules for programming.
- In array Once the size of an array is declared, you cannot change it.
- Sometimes the size of the array you declared may be insufficient.
- To solve this issue, you can allocate memory manually during run-time
- This is known as dynamic memory allocation in C programming.

Dynamic Memory Allocation

- To allocate memory dynamically, library functions

  are malloc(), calloc(), realloc() and free() are used

- These functions are defined in the <stdlib.h> header file.

Malloc()

- **“malloc”** or **“memory allocation”** method allocates memory at runtime.
- It takes the size in bytes and allocates that much space in the memory.
- It means that malloc(50) will allocate 50 byte in the memory.
- It returns a void pointer and is defined in stdlib.h.
- **Syntax:**
- ptr = (cast-type*) malloc(byte-size)
- **ptr = (int*) malloc(100 * sizeof(int));//100*4=400**
- It initializes each block with default garbage value.

Example

```c
int main()
{
    char name[20];
    char *address;

    strcpy(name, "Harry Lee");
    address = (char*)malloc( 50 * sizeof(char) );  /* allocating memory dynamically  */
    strcpy( address, "Lee Fort, 11-B Sans Street");

    printf("Name = %s\n", name );
    printf("Address: %s\n", address );
    return 0;
}
```

Example

```
scanf("%d", &n);//100
ptr = (int*) malloc(n * sizeof(int));//400
if(ptr == NULL)
 {
   printf("Error! memory not
allocated.");
   exit(0);
    }
```

```
printf("Enter elements: ");
   for(i = 0; i < n; ++i)
   {
       scanf("%d", ptr + i);
       sum += *(ptr + i);
   }
printf("Sum = %d", sum);
free(ptr);
// deallocating the memory
```

Calloc()

- The name "calloc" stands for contiguous allocation.
- The malloc() function allocates memory and leaves the memory uninitialized, whereas the calloc() function allocates memory and initializes all bits to zero.
- ptr = (float*) calloc(25, sizeof(float));
- This statement allocates contiguous space in memory for 25 elements each with the size of the float.
- calloc initializes the allocated memory to zero value whereas malloc doesn’t.
- calloc is used to allocate memory to mostly arrays and structures

Example

```
int main()
{
    int n,i,*p;
    printf("Enter number of elements: ");
    scanf("%d",&n);
    p=(int*)calloc(n, sizeof(int));  //memory allocated using malloc
    if(p == NULL)
    {
        printf("memory cannot be allocated\n");
    }
    else
    {
      printf("Enter elements of array:\n");
      for(i=0;i<n;++i)
      {
        scanf("%d",&*(p+i));
      }
      printf("Elements of array are\n");
      for(i=0;i<n;i++)
      {
        printf("%d\n",*(p+i));
      }
    }
    return 0;
}
```

Free()

- Dynamically allocated memory created with

  either calloc() or malloc() doesn't get freed on their own.

- You must explicitly use free() to release the space.

Practical-10.6

1. Start
2. Declare int size and pointer variable table and p
3. Read size//5
4. Table=Allocate memory using malloc//5*4=20//table=1000-1020
5. If table==0 print message memory full or no space available
6. Read input values using for loop(P=table to p<table+size)//1000—40-1040
7. Print values using for loop
8. End

Practical-10.7

- Write a program to store a character string in block of memory space created by malloc and then modify the same to store a large string.

Realloc

- If suppose we allocated more or less memory than required, then we can change the size of the previously allocated memory space using **realloc**
- **void *realloc(pointer, new-size);**

Realloc()

```
int main()
 {
      char *p1;
      int m1, m2; m1 = 10; m2 = 20;
      p1 = (char*)malloc(10);
      strcpy(p1, “UVPCE");
      p1 = (char*)realloc(p1, 10);
       strcpy(p1, “Ganpat University");
       printf("%s\n", p1);
}
```