Unit #: 2

Unit Name: Counting, Sorting and Searching

Topic: Pointers

Course objectives:

The objective(s) of this course is to make students

CObj1: Acquire knowledge on how to solve relevant and logical problems using computing machine

CObj2: Map algorithmic solutions to relevant features of C programming language constructs

CObj3: Gain knowledge about C constructs and it's associated eco-system

CObj4: Appreciate and gain knowledge about the issues with C Standards and it's respective behaviors

CObj5: Get insights about testing and debugging C Programs

Course outcomes:

At the end of the course, the student will be able to

CO1: Understand and apply algorithmic solutions to counting problems using appropriate C Constructs

CO2: Understand, analyze and apply text processing and string manipulation methods using C Arrays, Pointers and functions

CO3: Understand prioritized scheduling and implement the same using C structures

CO4: Understand and apply sorting techniques using advanced C constructs

CO5: Understand and evaluate portable programming techniques using preprocessor directives and conditional compilation of C Programs

Team – PSWC,
Jan - May, 2022
Dept. of CSE,
PES University

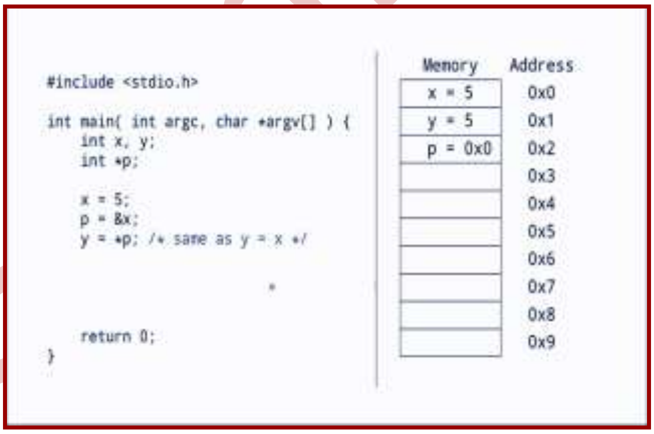
Pointers

- Pointer is a variable which contains the address. This address is the location of another object in the memory
- Pointers can be used to access and manipulate data stored in memory.
- Pointer of particular type can point to address any value of that particular type.
- Size of pointer of any type is same /constant in that system.
- Not all pointers actually contain an address
  Example: NULL pointer // Value of NULL pointer is 0.

Pointer can have three kinds of contents in it

1. The address of an object, which can be de referenced.
2. A NULL pointer
3. Undefined value // If p is a pointer to integer, then – int *p;

Note: A **pointer** is a **variable** that **stores** the **memory address** of **another variable** as **its value. The address of the variable we are working with is assigned to the pointer**



We can get the value of the variable the pointer currently points to, by dereferencing pointer by using the * operator. When used in declaration like int* ptr, it creates a pointer variable. When not used in declaration, it act as a dereference operator w.r.t pointers

Pointer Declaration:

Syntax: Data-type *name;

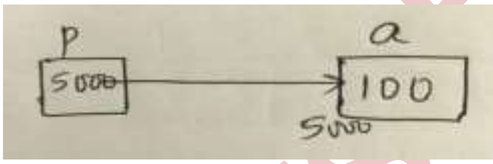
Example: int *p; // Compiler assumes that any address that it holds points to an integer type.

p= ∑ // Memory address of sum variable is stored into p.

Coding Example_1:

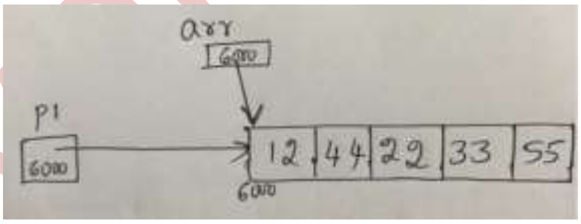
int *p; // p can point to anything where integer is stored. int* is the type. Not just int.

```
int a = 100;
p=&a;
printf("a is %d and *p is %d", a,*p);
```



Coding Example_2: Pointer pointing to an array

```
Now, int arr[] ={12,44,22,33,55};
int  *p1 = arr;  // same as int *p1;
p1 = arr;    // same as int *p1; p1 = &arr[0];
int arr2[10];
arra2 = arr;     // Arrays are assignment incompatible. Compile time Error
```



Pointer Arithmetic:

Below arithmetic operations are allowed on pointers

- Add an int to a pointer
- Subtract an int from a pointer
- Difference of two pointers when they point to the same array.

Integer is not same as pointer. We get warning when we try to compile the code where integer is stored in variable of int* type.

Coding Example_3:

```
int arr[ ] = {12,33,44};
int *p2 = arr;
printf("before increment %p %d\n",p2, *p2);
```

```
p2++;  //same as p2 = p2+1
          // This means 5000+sizeof(every element)*1 if 5000 is the base address
          //increment the pointer by 1. p2 is now pointing to next location.
printf("after increment %p %d\n",p2, *p2);
```

Coding Example_4: Example on Pointer Arithmetic :

```
int *p, x = 20;
p = &x;
printf("p     = %p\n", p);
printf("p+1 = %p\n", (int*)p+1);
printf("p+1 = %p\n", (char*)p+1);
printf("p+1 = %p\n", (float*)p+1);
printf("p+1 = %p\n", (double*)p+1);
```

Sample output:

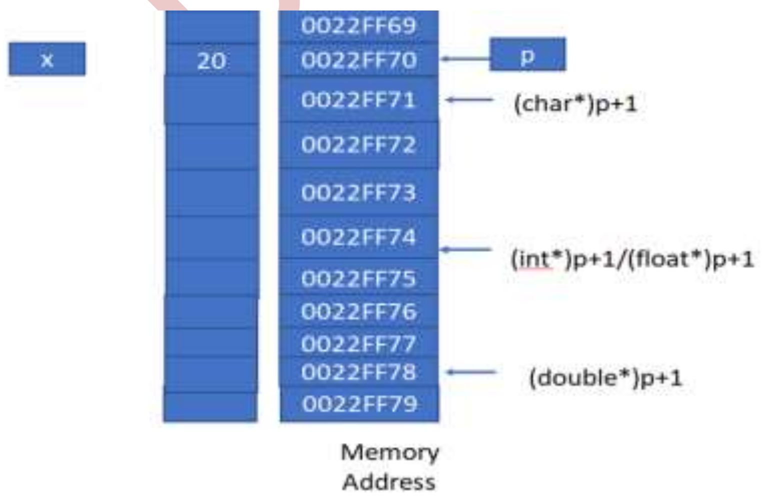
p   = 0022FF70

p+1 = 0022FF74

p+1 = 0022FF71

p+1 = 0022FF74

p+1 = 0022FF78



Coding Example_5:

```
int main()
{
        int *p;
        int a = 10;
        p = &a;
        printf("%d\n",(*p)+1);        // 11 ,p is not changed
        printf("before *p++ %p\n",p);          //address of p
        printf("%d\n",*p++);          // same as *p   and then p++ i.e 10
        printf("after *p++ %p\n",p);
        //address incremented by the size of type of value stored in it
        return(0);
}
```

Coding Example_6:

```
int main()
{
        int *p;
        int a = 10;
        p = &a;
        printf("%d\n",*p);//10
        printf("%d\n",(*p)++);// 10 value of p is used and then value of  p is incremented
        printf("%d\n",*p); // 11
        return 0;
}
```

Array Traversal using pointers:

Version 1: Index operator can be applied on pointer. Array notation

```
for(i = 0;i<5;i++)
        printf("%d \t",p3[i]); // 12 44 22 33 55
// every iteration added i to p3 .p3 not modified
```

Version 2: Using pointer notation

```
for(i = 0;i<5;i++)
        printf("%d\t",*(p3+i));        // 12   44    22    33    55
// every iteration i value is added to p3 and content at that address is printed.
// p3 not modified
```

Version 3:

```
for(i = 0;i<5;i++)
        printf("%d \t",*p3++); // 12 44 22 33 55
// Use p3, then increment, every iteration p3 is incremented.
```

Version 4: **undefined behavior if you try to access outside bound**

```
for(i = 0;i<5;i++)
        printf("%d    \t",*++p3);    // 44   22    33    55    undefined value
// every iteration p3 is incremented.
```

Version 5:

```
for(i = 0;i<5;i++)
                printf("%d \t",(*p3)++); // 12 13 14 15 16
// every iteration value at p3 is used and then incremented.
```

Version 6:

```
for(i = 0;i<5;i++,p3++)
        printf("%d \t",*p3); // 12  44  22  33 55
// every iteration value at p3 is used and then p3 is incremented.
```

Version 7: **p3 and arr has same base address of the array stored in it. But array is a constant pointer. It cannot point to anything in the world.**

```
for(i = 0;i<5;i++)
	printf("%d	\t", *arr++);	// Compile Time Error
```

Arrays and Pointers:

An array during compile time is an actual array but degenerates to a constant pointer during run time. Size of the array returns the number of bytes occupied by the array. But the size of pointer is always constant in that particular system.

Coding Example_8:

```
int *p1;
float *f1 ;
char *c1;
printf("%d%d%d ",sizeof(p1),sizeof(f1),sizeof(c1)); // Same value for all
int a[] = {22,11,44,5};
int *p = a;
a++;	// Error constant pointer
p++;	// Fine
p[1] = 222; //  allowed
a[1] = 222 ; // Fine
```

Note: If variable i is used in loop for the traversal, a[i], *(a+i), p[i], *(p+i), i[a], i[p] are all same.

Differences between array and pointer:

1. The sizeof operator:
   - sizeof(array) returns the amount of memory used by all elements in array
   - sizeof(pointer) only returns the amount of memory used by the pointer variable itself
2. The & operator:
   - &array is an alias for &array[0] and returns the address of the first element in array
   - &pointer returns the address of pointer

3. String literal initialization of a character array – Will be discussed in detail in next lecture

- char array[] = “abc” sets the first four elements in array to ‘a’, ‘b’, ‘c’, and ‘\0’
- char *pointer = “abc” sets pointer to the address of the “abc” string (which may be stored in read-only memory and thus unchangeable)
- Pointer variable can be assigned a value whereas array variable cannot be.

4. Pointer variable can be assigned a value whereas array variable cannot be.

```
int a[10];
int *p;
p=a; //allowed
a=p; //not allowed
```

5. An arithmetic operation on pointer variable is allowed.

```
int a[10];
int *p;
p++; /*allowed*/
a++; /*not allowed*/
```

Happy Coding using Arrays and Pointers!!