Pointers

SESSION 8

Objectives

- Explain what a pointer is and where it is used
- Explain how to use pointer variables and pointer operators
- Assign values to pointers
- Explain pointer arithmetic
- Explain pointer comparisons
- Explain pointers and single dimensional arrays
- Explain Pointer and multidimensional arrays
- Explain how allocation of memory takes place

What is a pointer

- A pointer is a variable, which contains the address of a memory location of another variable
- If one variable contains the address of another variable, the first variable is said to point to the second variable
- A pointer provides an indirect method of accessing the value of a data item
- Pointer can point to variables of other fundamental data types like int, char, or double or data aggregates like arrays or structures

What are pointers used for ?

- To return more than one value from a function
- To pass arrays and strings more conveniently from one function to another
- To manipulate arrays easily by moving pointers to them instead of moving the arrays itself
- To allocate memory and access it (Direct Memory Allocation)

Pointer variables

- A pointer declaration consists of a base type and a variable name preceded by an *
- Syntax:

```
type *name;
```

- Example :

```
int *p1;
float *p2;
char *p3;
```

Pointer operators

- There are 2 special unary operators which are used with pointers: **&** and *****
- The **&** operator returns the memory address of the operand:

```
int n = 10;
int *p;
p = &n;
```

- The operator ***** is the complement of **&**. It returns the value contained in the memory pointed to by the pointer variable's value.

```
int x = *p;
```

Assign values to pointers

- By the **&** operator

```
char  c = 'a';
char *p;
p = &c;
```

- By another pointer variable

```
char *p2;
 p2 = p;
```

- Variables can be assigned values through their pointers

```
*p = 'b';
```

Pointer arithmetic – 1/2

- *Addition* and *subtraction* are the only operations that can be performed on pointers

```
int n, *p;
p = &n;
n = 20;
p++;
```

- Let us assume that **n** is stored at the address **1000**
- Then **p** has the value 1000.
  Since integers are 2 bytes long, after the expression "p++;" **p** will have the value as 1002.

Pointer arithmetic – 2/2

- When a pointer is incremented, it points to the memory location of the next element of its base type
- Each time it is decremented, it points to the location of the previous element
- All other pointers will increase or decrease depending on the length of the data type they are pointing to

++p or p++	**point to next integer after n**
--p or p--	point to next integer previous to n
p + i	point to the i^{th} integer after n
p - i	point to the i^{th} integer before n
++*p or (*p)++	Increment n by 1
*p++	get (fetch) value of the next integer after n

Pointer comparisons

- Two pointers can be compared provided the both are pointing to variables of the same type
- Consider that pointers pa and pb, which point to data elements a and b. Following comparisons are possible:

pa < pb	**Return true if a is stored before b**
pa > pb	Return true if a is stored after b
pa <= pb	Return true if a is stored before b / pa, pb point to the same location
pa >= pb	Return true if a is stored after b / pa, pb point to the same location
pa == pb	Return true if the both point to the same element
pa != pb	Return true if the both point to the different elements
pa == NULL	Return true if pa is assigned NULL value (zero)

Pointer & single dimension array -1/2

The address of an array element can be expressed in two ways :

- By writing an expression in which the subscript is added to the array name
  ex: a+i

- By writing the actual array element preceded by the ampersand sign (&)
  ex: &a[i]

Pointer & single dimension array -2/2

```
void main()
{
  static int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
  int i;
  for (i = 0; i < 10; i ++) {
     printf("\n i=%d,a[i]=%d,*(a+i)=%d", i, ary[i], *(ary + i));
     printf("&a[i]= %X, a+i= %X\n", &ary[i], ary+i); // %X unsigned hexa
  }
}
```

```
i=0   ary[i]=1   *(ary+i)=1   &ary[i]=194   ary+i = 194
i=1   ary[i]=2   *(ary+i)=2   &ary[i]=196   ary+i = 196
i=2   ary[i]=3   *(ary+i)=3   &ary[i]=198   ary+i = 198
i=3   ary[i]=4   *(ary+i)=4   &ary[i]=19A   ary+i = 19A
i=4   ary[i]=5   *(ary+i)=5   &ary[i]=19C   ary+i = 19C
i=5   ary[i]=6   *(ary+i)=6   &ary[i]=19E   ary+i = 19E
i=6   ary[i]=7   *(ary+i)=7   &ary[i]=1A0   ary+i = 1A0
i=7   ary[i]=8   *(ary+i)=8   &ary[i]=1A2   ary+i = 1A2
i=8   ary[i]=9   *(ary+i)=9   &ary[i]=1A4   ary+i = 1A4
i=9   ary[i]=10  *(ary+i)=10  &ary[i]=1A6   ary+i = 1A6
```

Pointer with multi-dimension array

- A two-dimensional array can be defined as a pointer to a group of contiguous one-dimensional arrays
- A two-dimensional array declaration can be written as

data_type (*ptr_var) [expr 2];

instead of

data_type [expr1] [expr 2];

Pointer with string

Output

```
Enter a sentence: We all live in a yellow submarine
Enter character to search for: Y
String starts at address: 65420.
First occurrence of the character is at address: 65437.
Position of first occurrence (starting from 0) is: 17
```

```
void main() {
   char a, str[81], *ptr;
   printf("\n Enter a sentence:");  gets(str);
   printf("\n Enter character to search for:");  a=getche();
   ptr = strchr(str,a); /* return pointer to char*/

   printf("\n String starts at address: %u",str);
   printf("\n First occurrence of the char is at address: %u", ptr);
   printf("\n Position occurrence (from 0) is: %d", ptr - str);
}
```

Allocating memory

- The function **malloc()** permits allocation of memory from the pool of free memory.
- The parameter for **malloc()** is an integer that specifies the number of bytes needed.
- Syntax:

```
void* malloc(int size_t);
```

malloc() - example

```
#include <stdio.h>
#include <stdlib.h>     /*required for malloc and free functions*/
int main() {
   int n, i,j, temp, *p;
   printf("enter number of elements: "); scanf("%d", &n);
   p = (int *) malloc (n*sizeof(int));
   if(p!=NULL) {
     for(i=0; i<n ; i++) {
        printf("input element no. %d", i+1); scanf("%d", p+i);
     }
     for(i=0; i<n-1; i++)
       for(j=i+1; j<n; j++)
         if(*(p+i)> *(p+j)){
             temp = *(p+i); *(p+i )= *(p+j); *(p+j)= temp;
         }
     for(i=0; i<n; i++) printf("%d \n", *(p+i));
  }
}
```

free()

free() function can be used to de-allocates (frees) memory when it is no longer needed.

Syntax:

```
void  free(void *p);
```

- This function deallocates the space pointed to by p, freeing it up for future use.

- p must have been used in a previous call to **malloc(), calloc(),** or **realloc().**

free() - example

```
#include <stdio.h>
#include <stdlib.h>     /*required for malloc and free functions*/
int main() {
   int n, i, *p;
   printf("How many ints would you like store? ");
   scanf("%d", &n);
   p = (int *) malloc (n*sizeof(int));
   if(p!=NULL) {
     for(i=0; i<n ; i++) *(p+i) = i;
     for(i=n-1; i>0; i--)
          printf("%d\n",*(p+i));  /*print out in reverse*/
     free(p);                      /* free allocated memory */
   }
   else {
     printf("\n Memory allocation failed - not enough memory.\n");
  }
}
```

calloc()

calloc() is similar to **malloc()**, but the main difference is that the values stored in the allocated memory space is zero by default

Syntax:

```
void *calloc(int size, int size_t);
```

- The first parameter is the number of variables you'd like to allocate memory for
- The second is the size of each variable

calloc() - example

```
#include <stdio.h>
#include <stdlib.h>
int main() {
  float *p1, *p2;
  int i;
  p1 = (float *) calloc(3, sizeof(float));
  p2 = (float *) calloc(3, sizeof(float));
  if(p1!=NULL && p2!=NULL) {
     for(i=0 ; i<3 ; i++)  {
         printf("\n p1[%d] holds %05.2f ", i, p1[i]);
         printf("\n p2[%d] holds %05.2f ", i, *(p2+i));
     }
      free(p1);
      free(p2); return 0;
  } else {
     printf("Not enough memory\n"); return 1;
  }
}
```

realloc()

- You've allocated a certain number of bytes for an array but later find that you need to add more values to it. You could copy everything into a larger array: inefficient !
Instead of, using **realloc()**, you can get more bytes without losing data.
- Syntax

```
void *realloc(void *p, int size_t)
```

  - The first parameter is the pointer referencing the memory
  - The second is the total number of bytes you want to reallocate

realloc() - example

```
int main() {
  int *p, i;
  p = (int *)calloc(3, sizeof(int *));
  if(p!=NULL) {
      *p = 1;  *(p+1) = 2; p[2]= 4;
      p= (int *)realloc(ptr, 5*sizeof(int));
      if(p!=NULL){
         printf("Now allocating more memory.\n");
         p[3]= 8; p[4]= 32      // now it's legal!
         for(i=0; i<5; i++)
             printf("p[%d] = %d\n", i, p[i]);

         realloc(p,0);          // same as free(ptr)
      } else {
         printf("Not enough memory - realloc failed.");
      }
  } else {
      printf("Not enough memory - calloc failed"); }
}
```