

# Pointers

Instructor:

## Agenda

- 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
- Explain function pointers

## 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
- Pointers 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?

Some situations where pointers can be used are -

- 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 \*

**General declaration syntax is :**

`type *name;`

**For Example:**

`int *var2;`

## Pointer Operators

- There are 2 special operators which are used with pointers

▪ **&** and **\***

- The **&** operator is a unary operator and it returns the memory address of the operand

**var2 = &var1;**

- The second operator **\*** is the complement of **&**. It is a unary operator and returns the value contained in the memory location pointed to by the pointer variable's value

**temp = \*var2;**

## Assigning Values To Pointers-1

- Values can be assigned to pointers through the & operator.

```
ptr_var = &var;
```

- Here the address of var is stored in the variable ptr\_var
- It is also possible to assign values to pointers through another pointer variable pointing to a data item of the same data type

```
ptr_var = &var;  
ptr_var2 = ptr_var;
```

## Assigning Values To Pointers-2

- Variables can be assigned values through their pointers as well

**\*ptr\_var = 10;**

- The above declaration will assign 10 to the variable var if ptr\_var points to var

## Pointer Arithmetic-1

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

```
int var, *ptr_var;  
ptr_var = &var;  
var = 500;  
ptr_var++;
```

- Let us assume that **var** is stored at the address **1000**
- Then ptr\_var has the value 1000 stored in it. Since integers are 2 bytes long, after the expression "ptr\_var++;" ptr\_var will have the value as 1002 and not 1001

## Pointer Arithmetic-2

<code>++ptr_var or ptr_var++</code>	points to next <b>integer</b> after var
<code>--ptr_var or ptr_var--</code>	points to <b>integer</b> previous to var
<code>ptr_var + i</code>	points to the <i>i</i> th integer after var
<code>ptr_var - i</code>	points to the <i>i</i> th integer before var
<code>++*ptr_var or (*ptr_var)++</code>	will increment <b>var</b> by 1
<code>*ptr_var++</code>	will fetch the value of the next integer after var

- Each time 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

## Pointer Comparisons

- Two pointers can be compared in a relational expression provided both the pointers are pointing to variables of the same type
- Consider that ptr\_a and ptr\_b are 2 pointer variables, which point to data elements a and b. In this case the following comparisons are possible:

ptr_a < ptr_b	Returns true provided a is stored before b
ptr_a > ptr_b	Returns true provided a is stored after b
ptr_a <= ptr_b	Returns true provided a is stored before b or ptr_a and ptr_b point to the same location
ptr_a >= ptr_b	Returns true provided a is stored after b or ptr_a and ptr_b point to the same location.
ptr_a == ptr_b	Returns true provided both pointers ptr_a and ptr_b points to the same data element.
ptr_a != ptr_b	Returns true provided both pointers ptr_a and ptr_b point to different data elements but of the same type.
ptr_a == NULL	Returns true if ptr_a is assigned NULL value (zero)

## **Pointers and Single Dimensional Arrays-1**

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

- By writing the actual array element preceded by the ampersand sign (&)
- By writing an expression in which the subscript is added to the array name

## Pointers and Single Dimensional Arrays-2

```
#include<stdio.h>
void main()
{
    static int ary[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int i;
    for (i = 0; i < 10; i++)
    {
        printf("\ni=%d,ary[i]=%d,*(ary+i)=%d",i,
               ary[i],*(ary + i));
        printf("&ary[i]= %X,ary+i=%X",&ary[i],ary+i);
        /* %X gives unsigned hexadecimal */
    }
}
```

## Pointers and Single Dimensional Arrays-3

### Output

```
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
```

## Pointers and Multi Dimensional Arrays-1

- 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 ptr_var [expr1] [expr 2];`

## Pointers and Strings-1

```
#include <stdio.h>
#include <string.h>
void main () {
    char a, str[81], *ptr;

    printf("\nEnter a sentence:");
    gets(str);
    printf("\nEnter character to search for:");
    a = getche();
    ptr = strchr(str,a);
    /* return pointer to char*/
    printf( "\nString starts at address: %u",str);
    printf("\nFirst occurrence of the character is at
address: %u ",ptr);
    printf("\n Position of first occurrence(starting from
0)is: % d", ptr-str);
}
```

## Pointers and Strings-2

### 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

## Allocating Memory-1

The **malloc()** function is one of the most commonly used functions which permit allocation of memory from the pool of free memory. The parameter for **malloc()** is an integer that specifies the number of bytes needed.

## Allocating Memory-2

```
#include<stdio.h>
#include<malloc.h>
void main()
{
    int *p,n,i,j,temp;
    printf("\n Enter number of elements in the array :");
    scanf("%d",&n);
    p=(int*)malloc(n*sizeof(int));
    for(i=0;i<n;++i) {
        printf("\nEnter 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));
}
```

### Example

## free()-1

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

### Syntax:

```
void free(void *ptr );
```

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

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

## free()-2

```
#include <stdio.h>
#include <stdlib.h> /*required for the malloc and free functions*/
int main() {
    int number;
    int *ptr;
    int i;

    printf("How many ints would you like store? ");
    scanf("%d", &number);
    ptr = (int *) malloc (number*sizeof(int)); /*allocate memory
*/
    if (ptr != NULL) {
        for(i=0 ; i<number ; i++) {
            *(ptr+i) = i;
        }
    }
}
```

Contd...

## free()-3

```
for (i=number; i>0 ; i--) {  
    printf("%d\n",*(ptr+(i-1)));  
    /* print out in reverse order */  
}  
    free(ptr); /* free allocated memory */  
    return 0;  
} else {  
    printf("\nMemory allocation failed - not enough  
memory.\n");  
    return 1;  
}  
}
```

## calloc()-1

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

- **calloc** requires two arguments
- The first is the number of variables you'd like to allocate memory for
- The second is the size of each variable

Syntax :

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

## calloc()-2

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    float *calloc1, *calloc2;
    int i;

    calloc1 = (float *) calloc(3, sizeof(float));
    calloc2 = (float *)calloc(3, sizeof(float));

    if ((calloc1 != NULL) && (calloc2 != NULL)) {
        for(i=0 ; i<3 ; i++) {
            printf("calloc1[%d] holds %05.5f ", i, calloc1[i]);
            printf("\ncalloc2[%d] holds %05.5f ", i,
                  *(calloc2+i));
        }
    }
}
```

Contd.....

## calloc()-3

```
    free(calloc1);
    free(calloc2);
    return 0;
} else {
    printf("Not enough memory\n");
    return 1;
}
}
```

## realloc()-1

You've allocated a certain number of bytes for an array but later find that you want to add values to it. You could copy everything into a larger array, which is inefficient, or you can allocate more bytes using **realloc**, without losing your data.

- **realloc** takes two arguments
- The first is the pointer referencing the memory
- The second is the total number of bytes you want to reallocate

Syntax:

```
void *realloc( void *ptr, size_t size );
```

## realloc()-2

```
#include<stdio.h>
#include <stdlib.h>
int main() {
    int *ptr;
    int i;

    ptr = (int *)calloc(5, sizeof(int *));
    if (ptr != NULL) {
        *ptr = 1; *(ptr+1) = 2;
        ptr[2] = 4; ptr[3] = 8; ptr[4] = 16;
        ptr = (int *)realloc(ptr, 7*sizeof(int));
        if (ptr != NULL) {
            printf("Now allocating more memory... \n");
            ptr[5] = 32; /* now it's legal! */
            ptr[6] = 64;
```

## realloc()-3

```
for (i=0 ; i<7 ; i++) {  
    printf("ptr[%d] holds %d\n", i, ptr[i]);  
}  
realloc(ptr,0); /* same as free(ptr); - just fancier! */  
return 0;  
} else {  
    printf("Not enough memory - realloc failed.\n");  
    return 1;  
}  
} else {  
    printf("Not enough memory - calloc failed.\n");  
    return 1;  
}  
}
```

## Function Pointers

- Address is the entry point of the function
- Function has a physical location in memory that can be assigned to a pointer
- By using function pointers, a function can be sent as a parameter to another function.
- This feature enables the C program to load function dynamically at runtime.

Note: Function pointer is very important concept for understanding dynamic-binding concept, delegate, event, or polymorphism in OO languages such as Java, C++\C#

## Function Pointers - Example

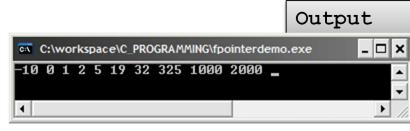
```
/* Quick sort example */
#include <stdio.h>
#include <stdlib.h>

int compare (const void * a, const void * b);

int main () {
    int values[] = { 2, 5, -10, 1000, 19, 32, 325, 2000, 0, 1 };
    int n;

    qsort (values, 10, sizeof(int), compare);
    for (n=0; n<10; n++) {
        printf ("%d ",values[n]);
    }
    return 0;
}

int compare (const void * a, const void * b) {
    return ( *(int*)a - *(int*)b );
}
```





© FPT Software

31