CL1002 Programming Fundamentals

LAB 11

Nested Structure, Introduction to Pointers, Accessing Arrays using Pointer, Dynamic Memory Management

NATIONAL UNIVERSITY OF COMPUTER AND EMERGING SCIENCES

Learning Objectives

1. Structures
2. Nested Structures
3. Introduction to Pointers
4. Accessing Array using Pointers
5. Dynamic Memory Allocation

1.0 Nested Structures

Nested structure in C is nothing but structure within structure. One structure can be declared inside other structure as we declare structure members inside a structure. The structure variables can be a normal structure variable ,array or a pointer variable to access the data. You can learn below concepts in this section.

```c
#include <stdio.h>
#include <string.h>

struct UniversityDetails
{
    int UniversityRanking;
    char UniversityName[90];
};

struct student_detail
{
    int id;
    char name[20];
    float percentage;
    // structure within structure
    struct UniversityDetails data;
};

int main()
{
    struct student_detail std_data = {1, "Arif", 80.5, 285,
                                      "National University of Computer & Emerging Sciences"};
    printf(" Id is: %d \n", std_data.id);
    printf(" Name is: %s \n", std_data.name);
    printf(" Percentage is: %f \n\n", std_data.percentage);

    printf(" University Ranking is: %d \n",
                std_data.data.UniversityRanking);
    printf(" University Name is: %s \n",
                std_data.data.UniversityName);
    return 0;
}
```

OUTPUT:

```
C:\Users\Shoaib\Documents\struct_example03.exe
 Id is: 1
 Name is: Arif
 Percentage is: 80.500000

 University Ranking is: 285
 University Name is: National University of Computer & Emerging Sciences

--------------------------------
Process exited after 0.1215 seconds with return value 0
Press any key to continue . . .
```

Another example of Nested Structure:

Sample Code:

```
#include <stdio.h>
#include <string.h>

struct Type{
    char TypeName[20];        // Mini, Sedan, Sports, Luxary, SUV
};

struct Car{
    char CarName[20];
    char make[15];
    char model[15];
    char color[10];
    int seats;
    int engine;          // 1800 cc
    int price;

    struct Type CarType;
};

int main()
{
    struct Car myCar;

    puts("------------------------- Example: Nested Structure -------------------------");

    puts("Enter the Name of your Car: ");
    gets(myCar.CarName);
    puts("Enter the type of your Car {Mini, Sedan, Sports, Luxary, SUV}: ");
    gets(myCar.CarType.TypeName);

    puts("Enter the Color of your Car: ");
    gets(myCar.color);
    puts("Enter the make of your Car: ");
    gets(myCar.make);
    puts("Enter the model of your Car: ");
    gets(myCar.model);
    printf("\nEnter the seats of your Car: ");
    scanf("%d",&myCar.seats);
    printf("\nEnter the engine cpacity (cc) of your Car: ");
    scanf("%d",&myCar.engine);
    printf("\nEnter the price of your Car: ");
    scanf("%d",&myCar.price);

 puts("\n\n------------------------- Print  ------------------------");

 printf("\nCarName: %s",myCar.CarName);
 printf("\nCarType: %s",myCar.CarType.TypeName);
 printf("\nColor: %s",myCar.color);
 printf("\nMake: %s",myCar.make);
 printf("\nModel: %s",myCar.model);
 printf("\nSeats: %d",myCar.seats);
 printf("\nEngine (cc): %d",myCar.engine);
 printf("\nPrice: %d", myCar.price);


 return 0;
```

OUTPUT:

C:\Users\Shoaib\Documents\struct_example02.exe

```
------------------------- Example: Nested Structure -------------------------
Enter the Name of your Car:
Picanto 2.0
Enter the type of your Car {Mini, Sedan, Sports, Luxary, SUV}:
Mini
Enter the Color of your Car:
White
Enter the make of your Car:
KIA
Enter the model of your Car:
Picanto

Enter the seats of your Car: 4

Enter the engine cpacity (cc) of your Car: 1300

Enter the price of your Car: 120000



------------------------- Print  -------------------------

CarName: Picanto 2.0
CarType: Mini
Color: White
Make: KIA
Model: Picanto
Seats: 4
Engine (cc): 1300
Price: 120000
--------------------------------
Process exited after 54.59 seconds with return value 0
Press any key to continue . . .
```

3.0 Introduction to Pointers

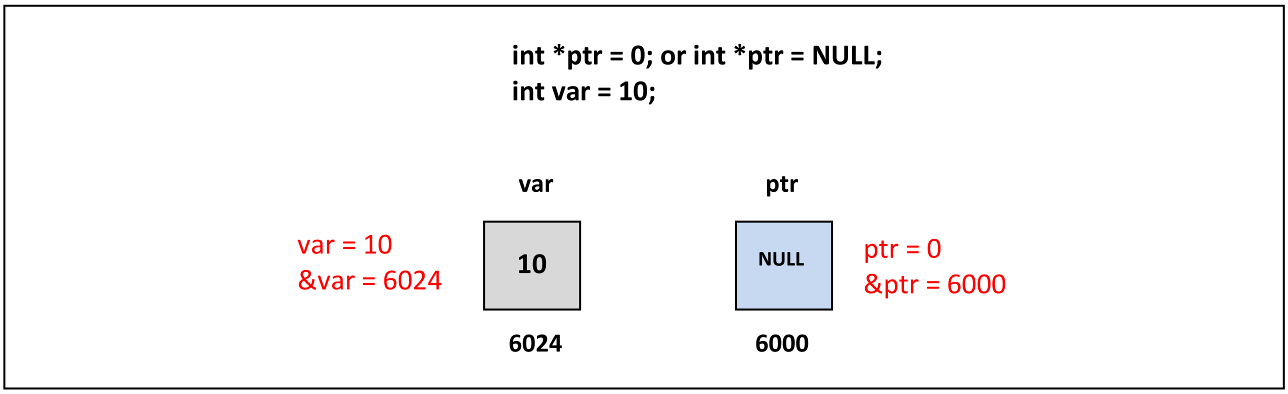
Pointer is a variable whose value is a memory address. Normally, a variable directly contains a specific value. A pointer contains the memory address of a variable that, in turn, contains a specific value. In this sense, a variable name directly references a value, and a pointer indirectly references a value.

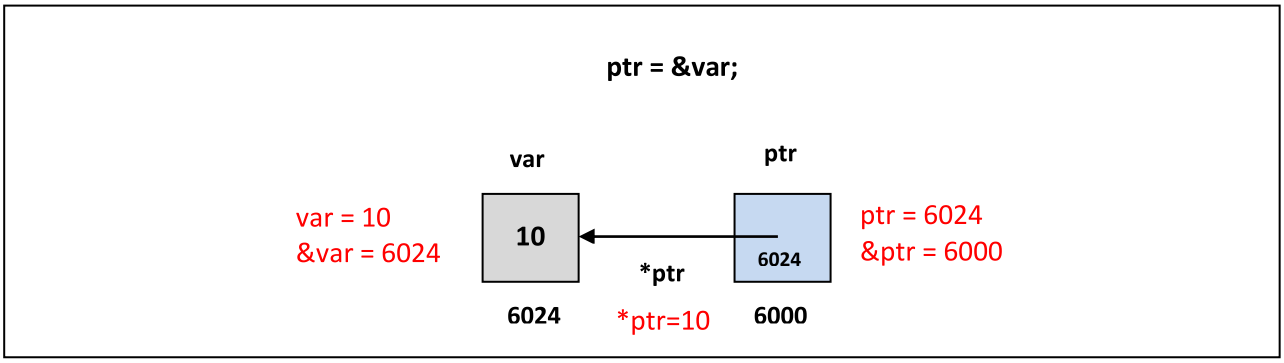
3.1 Pointer Declaration & Initialization

Syntax: type * variable;

Code:
```
int *ptr = 0;   // Pointer Declaration
int var= 10;
ptr =  &var;     // Pointer Initialization
```

The value of the pointer variable ptr is a memory address. A data item whose address is stored in this variable must be of the specified type.





Sample Code:

```c
#include <stdio.h>

int main()
 {
 int a; // a is an integer
 int *aPtr; // aPtr is a pointer to an integer

 a = 7;
 aPtr = &a; // set aPtr to the address of a

 printf( "The address of a is %p \nThe value of aPtr is %p", &a, aPtr);

 printf( "\n\nThe value of a is %d \nThe value of *aPtr is %d", a, *aPtr);

 printf( "\n\n Showing that * and & are complements of each other \n &*aPtr = %p \n *&aPtr = %p\n", &*aPtr,*&aPtr );
 } // end main
```

Select C:\Users\Shoaib\Documents\pointer_2.exe

```
The address of a is 000000000062FE1C
The value of aPtr is 000000000062FE1C

The value of a is 7
The value of *aPtr is 7

 Showing that * and & are complements of each other
 &*aPtr = 000000000062FE1C
 *&aPtr = 000000000062FE1C

--------------------------------
Process exited after 3.97 seconds with return value 110
Press any key to continue . . .
```

3.2 POINTER ARITHMETICS

- A limited set of arithmetic operations may be performed on pointers. A pointer may be incremented (++) or decremented (--), an integer may be added to a pointer (+ or +=), an integer may be subtracted from a pointer (- or -=) and one pointer may be subtracted from another.
- When an integer is added to or subtracted from a pointer, the pointer is incremented or decremented by that integer times the size of the object to which the pointer refers.
- Two pointers to elements of the same array may be subtracted from one another to determine the number of elements between them.

4.0 Accessing Array using Pointers

Arrays and pointers are intimately related in C and often may be used interchangeably.

- An array name can be thought of as a constant pointer.
- Pointers can be used to do any operation involving array subscripting.
- When a pointer points to the beginning of an array, adding an offset to the pointer indicates which element of the array should be referenced, and the offset value is identical to the array subscript. This is referred to as pointer/offset notation.
- An array name can be treated as a pointer and used in pointer arithmetic expressions that do not attempt to modify the address of the pointer.
- Pointers can be subscripted exactly as arrays can. This is referred to as pointer/subscript notation.

Sample Code:

```
#include <stdio.h>

int main()
{
	int *ptr = NULL; // pointer variable of type "pointer to int" / null pointer
	int intVariable1 = 10; // Declare an integer variable and initialize it with 10

	// Use address-of operator & to assign memory address of intVariable1 to a pointer
	ptr = &intVariable1;
	// Pointer ptr now holds a memory address of intVariable

	// Print out associated memory addresses and their values
	printf("The memory address allocated to ptr at the time of its creation:%d \n",&ptr);
	printf("\nptr is pointing to memory address or value contained in ptr:%d\n",ptr);
	printf("\nThe memory address allocated to intVariable at the time of its creation:%d\n",&intVariable1);
	printf("The value contained in intVariable:%d\n" ,intVariable1);
	printf("\nptr is pointing to the value:%d\n" , *ptr);

	int array[3] = {1,2,3},offset,i; //Initialize an array of three elements
	printf("\nThe number of bytes in the array is %d\n", sizeof(array));

	ptr = array; // Assign memory address of arr to pointer
	printf("\nThe number of bytes in the ptr is %d\n", sizeof(*ptr));

	printf("\nThe total number of elements in the array is %d\n", sizeof(array)/sizeof(*ptr));

	// Print out associated memory addresses and their values
	printf("\nThe memory address allocated to array at the time of it's creation:%d\n", array);
	printf("\nptr is now pointing to memory address array[0] or value now contained in ptr:%d\n",ptr);
	printf("\nThe value at array[0]:%d\n",*ptr);

	ptr++; //Adds 4 to the value(address) contained in ptr i.e to address of array[0] and now contains the address of array[1]
	printf("\nptr is now pointing to memory address array[1] or value now contained in ptr:%d\n",ptr);
	printf("\nThe value at array[1]:%d\n",*ptr);

	ptr--; //Subtracts 4 from the value(address) contained in ptr i.e from address of array[1] and now contains the address of array[0]
	printf("\nptr is now pointing to memory address array[0] or value now contained in ptr:%d\n",ptr);
	printf("\nThe value at array[0]:%d\n",*ptr);

	ptr = ptr+2; //Adds 4 to the value(address) contained in ptr i.e to address of array[0] and now contains the address of array[2]
	printf("\nptr is now pointing to memory address array[2] or value now contained in ptr:%d\n",ptr);
	printf("\nThe value at array[2]:%d\n",*ptr);

	ptr = ptr - 2;

	// Displaying array using array subscript notation
	printf( "\nArray printed with:\nArray subscript notation\n" );
	for ( i = 0; i < 3; ++i )
	{
		printf( "array[ %d ] = %d\n", i,array[ i ]);
	}

	// Displaying array using array name and pointer/offset notation
	printf( "\nPointer/offset notation where the pointer is the array name\n" );
	for ( offset = 0; offset < 3; ++offset )
	{
		printf( "*( array + %d ) = %d\n", offset, *(array + offset) );
	}


	// Displaying array using ptr and pointer/offset notation
```

```
    printf( "\nPointer/offset notation\n" );
    for ( offset = 0; offset < 3; ++offset )
    {
            printf( "*( ptr + %d ) = %d\n", offset, *(ptr + offset) );
    }

    // Displaying array using ptr and array subscript notation
    printf( "\nPointer subscript notation\n" );
    for ( i = 0; i < 3; ++i )
    {
            printf( "ptr[ %d ] = %d\n", i,ptr[ i ]);
    }

    return 0;
}
```

E:\Solat\ITC\Lab10\test.exe

```
The memory address allocated to ptr at the time of it's creation:2293312

ptr is pointing to memory address or value contained in ptr:2293308

The memory address allocated to intVariable at the time of it's creation:2293308
The value contained in intVariable:10

ptr is pointing to the value:10

The number of bytes in the array is 12

The number of bytes in the ptr is 4

The total number of elements in the array is 3

The memory address allocated to array at the time of it's creation:2293296

ptr is now pointing to memory address array[0] or value now contained in ptr:2293296

The value at array[0]:1

ptr is now pointing to memory address array[1] or value now contained in ptr:2293300

The value at array[1]:2

ptr is now pointing to memory address array[0] or value now contained in ptr:2293296

The value at array[1]:1

ptr is now pointing to memory address array[2] or value now contained in ptr:2293304

The value at array[1]:3

Array printed with:
Array subscript notation
array[ 0 ] = 1
array[ 1 ] = 2
array[ 2 ] = 3

Pointer/offset notation where the pointer is the array name
*( array + 0 ) = 1
*( array + 1 ) = 2
*( array + 2 ) = 3

Pointer/offset notation
*( ptr + 0 ) = 1
*( ptr + 1 ) = 2
*( ptr + 2 ) = 3

Pointer subscript notation
ptr[ 0 ] = 1
ptr[ 1 ] = 2
ptr[ 2 ] = 3

--------------------------------
Process exited after 0.06422 seconds with return value 0
Press any key to continue . . .
```

Passing Array Elements to a Function

Array elements can be passed to a function by calling the function by value, or by reference. In call by reference we pass addresses of array elements to the function.

/* Demonstration of call by reference */

```
main( )
{
 int num[ ] = { 24, 34, 12, 44, 56, 17 } ;
 dislpay ( &num[0], 6 ) ;
}
display ( int *j, int n )
{
    int i ;
for ( i = 0 ; i <= n - 1 ; i++ )
 {
printf ( "\nelement = %d", *j ) ;
j++ ; /* increment pointer to point to next element */
 }
}
```

Pointer to Pointer (Double Pointer)

```
int main()
{
    int var = 789;

    // pointer for var
    int *ptr2;

    // double pointer for ptr2
    int **ptr1;

    // storing address of var in ptr2
    ptr2 = &var;

    // Storing address of ptr2 in ptr1
    ptr1 = &ptr2;

    // Displaying value of var using
    // both single and double pointers
    printf("Value of var = %d\n", var );
    printf("Value of var using single pointer = %d\n", *ptr2 );
    printf("Value of var using double pointer = %d\n", **ptr1);
  return 0;
}
```

Pointers and 2-Dimensional Arrays

```
main( )
{
int s[4][2] = {
{ 1, 2 },
{ 3, 4 },
{ 5, 6 },
{ 7, 8 }
} ;
int i, j ;
for ( i = 0 ; i <= 3 ; i++ )
{
printf ( "\n" ) ;
for ( j = 0 ; j <= 1 ; j++ )
printf ( "%d ", *( *( s + i ) + j ) ) ;
}}
```

Structures and pointers

```
#include<stdio.h>

struct Point
{
   int x, y;
};

int main()
{
   struct Point p1 = {1, 2};

   // p2 is a pointer to structure p1
   struct Point *p2 = &p1;

   // Accessing structure members using structure pointer
   printf("%d %d", p2->x, p2->y);
   return 0;
}
```

Passing structures to a function

```
# include <stdio.h>
struct book
{
char name[50] ;
char author[25] ;
int volno ;
} ;

main()
{
struct book b1 = { "Introduction to Computers", "Peter Norton", 100 } ;
display ( &b1 ) ;
}

display ( struct book *b )
{
 printf ( "\n%s \n%s \n%d", b->name, b->author, b->volno ) ;
}
```

Passing to whole array

```
#include<stdio.h>
int main()
{
    // Pointer to an integer
    int *p;
    // Pointer to an array of 5 integers
    int (*ptr)[5];
    int arr[5];
    // Points to 0th element of the arr.
    p = arr;
    // Points to the whole array arr.
    ptr = &arr;

    printf("p = %u, ptr = %u\n", p, ptr);

    p++;
    ptr++;
    printf("p = %u, ptr = %u\n", p, ptr);
    return 0;
}
```

2D-Array with pointers

```
#include<stdio.h>
int main()
{
    int arr[3][4] = {
                        {1,2,3,4},
                        {5,6,7,8},
                        {9,10,11,12}
                    };
    int i, j;
    int (*p)[4];
    p = arr;
    for(i = 0; i < 3; i++)
    {
        printf("Address of %d th array %u \n",i , p + i);
        for(j = 0; j < 4; j++)
        {
            printf("arr[%d][%d]=%d\n", i, j, *( *(p + i) + j) );
        }
        printf("\n\n");
    }
}
```

4.0 Dynamic Memory

The process of allocating memory during program execution is called dynamic memory allocation. The ability for a program to obtain more memory space at execution time to hold new nodes, and to release space no longer needed is known as dynamic memory management.

4.1 Importance of Dynamic memory

Many times, it is not known in advance how much memory will be needed to store particular information in a defined variable and the size of required memory can be determined at run time. For example, we may want to hold someone's name, but we do not know how long their name is until they enter it. Or we may want to read in a number of records from disk, but we don't know in advance how many records there are. Or we may be creating a game, with a variable number of monsters (that changes over time as some monsters die and new ones are spawned) trying to kill the player.

C language offers 4 dynamic memory allocation functions. They are,

Function	Syntax
malloc ()	malloc (number *sizeof(int));
calloc ()	calloc (number, sizeof(int));
realloc ()	realloc (pointer_name, number * sizeof(int));
free ()	free (pointer_name);

Malloc()

- malloc function is used to allocate space in memory during the execution of the program.
- malloc does not initialize the memory allocated during execution. It carries garbage value.
- Malloc function returns null pointer if it couldn't able to allocate requested amount of memory.

Calloc()

- calloc function is also like malloc function. But calloc initializes the allocated memory to zero. But, malloc doesn't.

Realloc()

- realloc function modifies the allocated memory size by malloc and calloc functions to new size.
- If enough space doesn't exist in memory of current block to extend, new block is allocated for the full size of reallocation, then copies the existing data to new block and then frees the old block.

Free()

- free function frees the allocated memory by malloc, calloc, realloc functions and returns the memory to the system.

Malloc & free Sample Code:

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
  int n, i, *ptr, sum = 0;

  printf("Enter number of elements: ");
  scanf("%d", &n);

  ptr = (int*) malloc(n * sizeof(int));

  // if memory cannot be allocated
  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);

  // deallocating the memory
  free(ptr);

  return 0;
}
```

C:\Users\Shoaib\Documents\malloc_example_02.exe

```
Enter number of elements: 5
Enter elements: 2
1
2
3
1
Sum = 9
--------------------------------
Process exited after 42.24 seconds with return value 0
Press any key to continue . . .
```

Calloc & free Sample Code:

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
  int n, i, *ptr, sum = 0;
  printf("Enter number of elements: ");
  scanf("%d", &n);

  ptr = (int*) calloc(n, sizeof(int));
  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);
  return 0;
}
```

C:\Users\Shoaib\Documents\calloc_example_02.exe

```
Enter number of elements: 3
Enter elements: 6
9
2
Sum = 17
--------------------------------
Process exited after 21.82 seconds with return value 0
Press any key to continue . . .
```

Realloc Sample Code:

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
  int *ptr, i , n1, n2;
  printf("Enter size: ");
  scanf("%d", &n1);

  ptr = (int*) malloc(n1 * sizeof(int));

  printf("Addresses of previously allocated memory:\n");
  for(i = 0; i < n1; ++i)
    printf("%pc\n",ptr + i);

  printf("\nEnter the new size: ");
  scanf("%d", &n2);

  // rellocating the memory
  ptr = realloc(ptr, n2 * sizeof(int));

  printf("Addresses of newly allocated memory:\n");
  for(i = 0; i < n2; ++i)
    printf("%pc\n", ptr + i);

  free(ptr);

 return 0;
}
```

C:\Users\Shoaib\Documents\realloc_01.exe

```
Enter size: 2
Addresses of previously allocated memory:
0000000000C11400c
0000000000C11404c

Enter the new size: 5
Addresses of newly allocated memory:
0000000000C11400c
0000000000C11404c
0000000000C11408c
0000000000C1140Cc
0000000000C11410c

--------------------------------
Process exited after 5.863 seconds with return value 0
Press any key to continue . . .
```

4.2 Difference between static memory allocation and dynamic memory allocation in C

Static memory allocation	Dynamic memory allocation
In static memory allocation, memory is allocated while writing the C program. Actually, user requested memory will be allocated at compile time.	In dynamic memory allocation, memory is allocated while executing the program. That means at run time.
Memory size can’t be modified while execution. Example: array	Memory size can be modified while execution. Example: Linked list