C Pointers

1. Objective

- What is C structure?
- When to use structures.
- Syntax of a structure.
- How to declare variable of type structure?
- Fields of a structure and how to initialize them.
- How to manipulate structure type

2. Introduction

- Static vs. Dynamic variables
    - Static variables:
        - Size is fix throughput the execution
        - Size is known at compile time
        - Memory is allocated at execution
    - Dynamic Variables
        - Creation during the run time
        - Size may vary and it is not known at compile time
        - Generally, the user is responsible for freeing the allocated memory
- Clearly there is a need to **allocate memory** at run time instead of compile time.

- This raises two important issues:
  - You have to be able to give the memory back when you are done with it
  - If the memory isn't allocated yet, how should you refer to it in your program?

3. Pointer Variable Declarations and Initialization

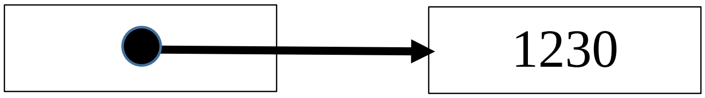
- Pointer variables:
  - Pointers store the address of a variable. They Contain memory addresses as their values
  - They are called pointers because storing the address of another variable is essentially a way of referring to, or pointing to, the variable
  - Normal variables contain a specific value (direct reference):

```
int var = 10234;
```

$\boxed{10234}$

- Pointers contain address of a variable that has a specific value (indirect reference)

A pointer to an integer



- Indirection – referencing a pointer value

- Pointer declarations
  - ***** used with pointer variables:

```
int * pi;     // declares a pointer to an integer
char *pc1, *pc2;  // pointers to characters
```

  - Multiple pointers require using a ***** before each variable declaration

```
int *p1, *p2;
```

  - Can declare pointers to any data type
- Pointer Initialization:

- A **NULL** pointer is a special pointer value that is known not to point anywhere.
- Initialize pointers to:
  - **0**,
  - **NULL (**preferred)
  - an address
- Example:

```
//gcc 5.4.0

#include  <stdio.h>

int main(void)
{
   int *ip = NULL;
   int i = 10;

   printf("Address of ip = %u\n", ip);

   ip = &i;

   printf("Content of ip = %d\n", *ip);
   printf("Address of ip = %u\n", ip);

   return 0;
}
```

4. Reference operator (&) and Dereference operator (*)

- **Reference operator (&): How to get the address of a variable**
  - Use & address operator

```
//gcc 5.4.0

#include  <stdio.h>

int main(void)
{
    int i = 5;
    int *ptr;    //  declare a pointer variable
    ptr = &i;   //  ptr stores the ADDRESS of i
    printf("The content of the memory location pointed to
by ptr = %d\n", *ptr);   // refer to referee of ptr


    return 0;
}
```

- **Accessing the content of a pointer:**
  - There are two ways to get the content of a pointer:
    - Dereference operator (*):

        - Using -> operator for sturctures
    - Use the dereference operator *:
        - It is used to refer the content of a pointer
        - Example:

```
//gcc 5.4.0

#include  <stdio.h>

int main(void)
{
   int * pi;
   int i=8;
   pi = &i;  // pi stores the ADDRESS of i
   printf("pi points to a memory location with %d\n",
*pi);
   *pi = 14;
   printf("Now pi points to a memory location with
%d\n", *pi);

   return 0;
}
```

- Exercise, what is the output of the following?

```
int *ip1, *ip2;
int x=34, y= 7;
ip1=&x;
ip2=ip1;
```

```
*ip2=12;
```

5. Relation between Arrays and Pointers

- In fact, you have already been using pointers when you were working with arrays.
- Given the following array:

```
int arr[4];
```

  - Then the name arr is actually a pointer to the first element in the array:

    arr[0] is equivalent to *arr

    *arr = 5;   is the same as   arr[0] = 5;

    arr[1] is equivalent to *(arr + 1)
    arr[2] is equivalent to *(arr + 2)
    arr[3] is equivalent to *(arr + 3)

- Example:

```
//gcc 5.4.0

#include  <stdio.h>

int main(void)
{

    int i;
    int arr[4];

    for (i=0;i<4;++i)
        *(arr + i) = 100;

    for (i=0;i<4;++i)
        printf("arr[%d] = %d\n", i, arr[i]);
   // printf("Address of ip = %u\n", ip);

    return 0;
}
```

6. Dynamic Memory Allocation

- Dynamic memory allocation allows your program to obtain more memory space while running.
- It allows to allocate/free memory during the execution of you program.

Function	Use of Function
malloc()	Allocates requested size of bytes and returns a pointer first byte of allocated space
calloc()	Allocates space for an array elements, initializes to zero and then returns a pointer to memory
free()	deallocate the previously allocated space
realloc()	Change the size of previously allocated space

- C malloc():
  - The name **malloc** stands for "memory allocation".

- The function malloc() reserves a block of memory of specified size and return a pointer of type void which can be casted into pointer of any form.
- **NULL** returned if not enough memory available.
- Syntax:

  ptr = (type) malloc(byte-size)*

- Example: Pointers & Structures
  - We can access the fields of a structure in two different ways:
    - Using Dereference operator *
    - Using -> operator

```
#include <stdio.h>
#include <stdlib.h>
struct person {
   int age;
   float weight;
   char name[30];
};

int main()
```

```
            {
              struct person *ptr;
              int i, num;



              ptr = (struct person*) malloc(sizeof(struct person));
              // Above statement allocates the memory for 1
            structure with pointer ptr pointing to base address */


                 printf("Enter name, age and weight of the person
            respectively:\n");
                 scanf("%s%d%f", &ptr->name, &ptr->age,
            &ptr->weight);


              printf("Displaying Infromation:\n");

                 printf("%s\t%d\t%.2f\n", ptr->name, ptr->age,
            ptr->weight);

              return 0;
      }
```

- C calloc():
  - The name calloc stands for "contiguous allocation".
  - It is used to allocate arrays of memory.

- calloc() allocates multiple blocks of memory each of same size and sets all bytes to zero. Note that malloc() only allocates a single block of memory.
- Syntax:

  ptr = (type) calloc(n, element-size);*

- Example: Dynamic Arrays

```
//gcc 5.4.0

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

int main(void)
{

   int *nums;
   int N;

   printf("Read how many numbers:\n");
   scanf("%d",&N);
   nums = (int *) calloc(N, sizeof(int));
   for (int i=0; i<N; ++i)
      *(nums+i) = i*i;

   for (int i=0; i<N; ++i)
```

```
        printf("Num[%d] = %d\n", i, *(nums+i));

    /* use array nums */

    /* when done with nums: */

    free(nums);

    /* would be an error to say it again - free(nums) */


    return 0;
}
```

- C free():
  - It is used to let the compiler that the allocated memory is no longer needed.
  - Note that the allocated memory is not freed until you explicitly free it up
  - Syntax:

    free(ptr);

  - Example:

    free(nums); // from the previous example.

- C realloc():
  - It is used to let the compiler that the allocated memory is no longer needed.
  - The C library function void *realloc(void *ptr, size_t size) attempts to resize the memory block pointed to by ptr that was previously allocated with a call to malloc or calloc.Syntax:

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

    Where

    ptr: is the pointer to a memory block previously allocated with malloc, calloc or realloc to be reallocated.
    If this is NULL, a new block is allocated and a pointer to it is returned by the function.

    size: is the new size for the memory block, in bytes. If it is NULL and ptr points to an existing block of memory, the memory block pointed by ptr is deallocated and a NULL pointer is returned.

- Example 1:

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

int main()
{
  char *str;

   str = (char *) malloc(20);
   strcpy(str, "C Programming");
   printf("String = %s,  Address = %u\n", str, str);

   /* Requesting more memory: Reallocating memory */
   str = (char *) realloc(str, 30);
   //Concatenate strings
   strcat(str, " language");
   printf("String = %s,  Address = %u\n", str, str);

  free(str);

  return(0);
}
```

- Example 1:

```
#include <stdio.h>
#include <stdlib.h>
void  main() {
 float *myarr;
 int i;
 /* Allocate an array of 5 floating point values */
 myarr = (float *) calloc(5, sizeof(float));
  /* myarr is an array of 5 floating point values */
  for (i = 0; i < 5; i++)
        myarr[i] = 2.0 * i;
  for (i = 0; i < 5; i++)
       printf("myarr[%d] = %lf\n", i, myarr[i]);
   printf("-----------------------------\n");
  /* Increase the size of the array by 5 more floating point
values */
    myarr = (float *) realloc(myarr,10 * sizeof(float));
    for (i = 5; i < 10; i++)
        myarr[i] = 10.0 * i;
    for (i = 0; i < 10; i++)
        printf("myarr[%d] = %lf\n", i, myarr[i]);
}
```

7. Call by reference

- Call by reference is done using pointers.
- Unlike call by value, call by reference passes the address of the variable to the function as parameter.
- The value of the actual parameter can be modified by formal parameter.
- The same memory location is used for both actual and formal parameters
- Example:

```
//gcc 5.4.0

#include  <stdio.h>

void byval(int j){
 j = 0;
}

void byref(int *j){
 *j = 0;
}
int main(void)
{
    int i = 100;
    byval (i);
    printf("i = %d\n", i);

    byref (&i);

    printf("i = %d\n", i);

    return 0;
}
```