Module 5: Pointers

Overview of Chapter

Lesson 1: Pointer Basics

Learning Outcomes

- To declare and initialize pointers
- To dereference the content of variables pointed by pointers

Discussion

So far, the variables discussed would store a numerical value to a particular memory address. However, a pointers are variables would store an address (memory location) of some value to an address. Pointer would not only reference other data types but functions as well.

Declaring and Assigning Addresses to Pointers.

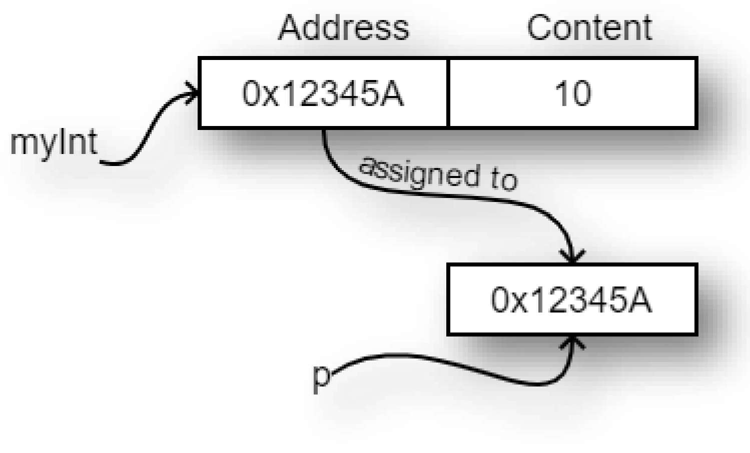
To declare a pointer variable that would point to some type, place an asterisk (*) before the variable name. For instance an integer pointer is declared as:

```
int *p;
```

A declared pointer without any assigned value would reference to an NULL address. So, to initialize a pointer with an address you use the & operator before the variable name to assign it to an existing and similar (i.e. int variable to an int pointer) memory address. For instance, the pointer from *p can be assigned with an address by:

```
int myInt = 10;
p = &myInt;
```

Now, p would not contain the value of 10 instead it would contain the address where the value of integer 10 is located. The illustration below shows this operation.

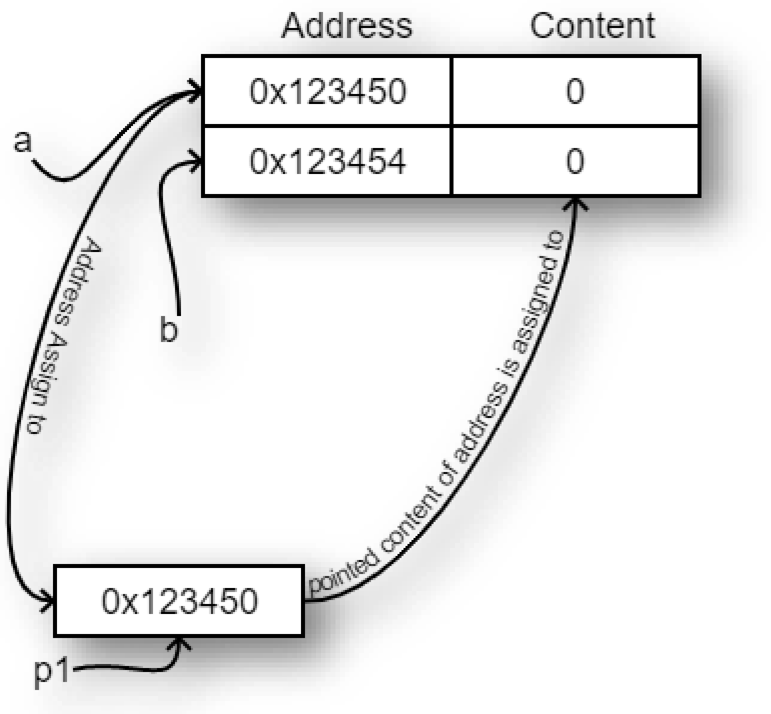


Pointer Dereferencing

The value of which the pointer points is accessed by using the * operator. For example:

```
int a = 0, b;
int *p1= &a;//declares p1 pointer and assign address a to it.
b= *p1;//assigns the content pointed to by p1 to b
printf ("%d ",b);//output is 0
```

This entire process of dereferencing is illustrated below.



A pointer can also be modified to change its reference, be sure that you would not reference a non-existing address since this would make you program terminate abruptly. For example:

```
int d=1, e=5;
int *p2;
p2 = &d; //pointer p2 points to integer d
p2 = &e;//pointer p2 now points to integer e
// p2 = &w; //erroneous since integer w does not exist
```

Lesson 2: Pointer Applications

Learning Outcome(s)

- ➢ To dynamically allocate memory during runtime using the calloc function
- ➢ To determine the array size during runtime.
- ➢ To apply the aforementioned techniques to solve dynamic memory allocation related problems.

Discussion

One of the most common application of array is dynamic memory allocation. Unlike static arrays from the Module 3 in which the size of the array is known during compile time, dynamic arrays would be allocated with memory during runtime. This results to appropriate memory allocation since only the needed memory size would be provided to the array when the program executes rather than reserving fixed memory allocation prior to program execution.

Dynamic memory allocation

The `calloc` function from the `<stdlib.h`> library would allocate a block of memory for an array of n elements during runtime. The syntax format is as follows:

```
pointer=(pointerDataType*) calloc (numOfElements, sizeOfElements);
```

In which:

- numOfElements - unsigned integer size of elements
- sizeOfElements - unsigned integer size (in bytes) per element.

Now, when a lot of memory is dynamically allocated it without releasing it, it would cause the computer's memory to run out as a consequence. So, the free statement would be used to release unused allocated memory.

```
free (pointer);
```

For example you would allocate n number of elements during runtime, initialize it all with 0, then print each element.

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

int main()
{
    int *dynMemInt;
    int i,n;
    printf ("Size of Array: ");
    scanf ("%d",&n);
    dynMemInt = (int*) calloc (n,sizeof(int));

    for (i=0;i<n;i++)
    {
        dynMemInt[i] = 0;
        printf("dynMemInt[%d]: %d \n",i,dynMemInt[i]);
    }
    free (dynMemInt);
}
```

The `sizeof` operator

The previous example used the `sizeof` operator to determine the allocation per element of the dynamic array. This is because the sizeof operator would return the size in bytes of a particular variable or datatype. For example:

```
sizeof(int); //would return 4 since each int is 4 bytes
int x[10];
sizeof(x);//would return 40 since 10 ints is 40 bytes
```

A more practical example is using the sizeof together with #define is to determine the size of the array during runtime. For example, you would copy the content of x array with unknown size during compile time to y array.

```
#include <stdio.h>
#include <stdlib.h>
//a definition to determine the size of the array during runtime
#define NUM_ELEM(x) (sizeof (x) / sizeof (*(x)))

int main()
{
    int x[] = {1,2,3,4,5,6,7,8,9,10};
    int *y;
    int i=0;
    y = (int*) calloc(NUM_ELEM(x),sizeof(int));

    for (i=0;i<NUM_ELEM(x);i++)
    {
        y[i]=x[i];
        printf("Y[%d]: %d\n",i,y[i]);
    }
    free (y);
}
```