C Programming

Recitation 5

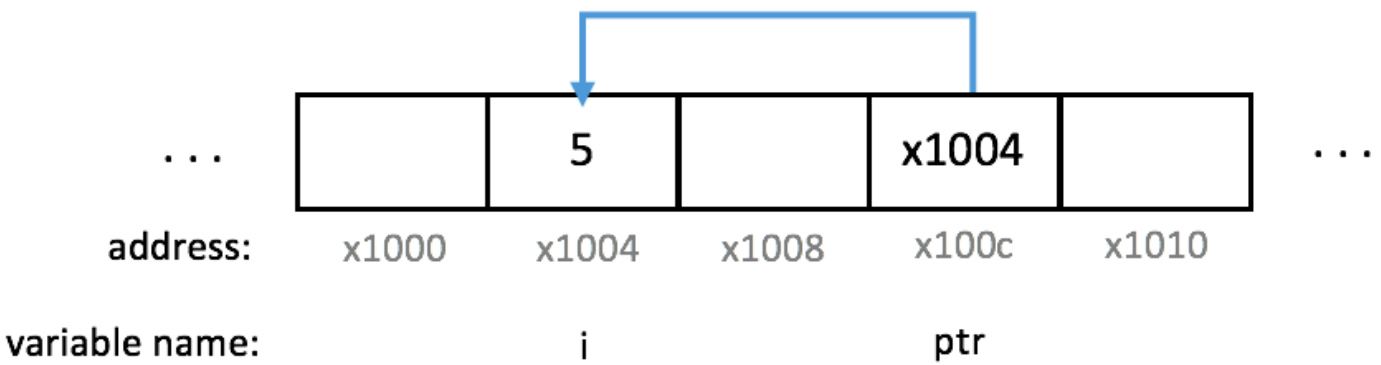
April 2017

Why do we need pointers?

- Enable us to achieve parameter passing by reference
- Dealing effectively with arrays
- Creating complex data structures
- Working with dynamically allocated memory (next week's topic)

What is a pointer?

- Each variable is stored in a storage location.
- Storage locations has addresses.
- We can store these address values as well.
- If a variable holds an address value, it is a pointer variable.



Address and Dereferencing Operators

- In order to work with pointers, we should know about these two operators:
  - & (Address operator): yields address when applied to a variable
  - * (Dereferencing operator): fetches the value when applied to an address
    - We also use * when declaring pointer variables
- $\&(address)$ and $*(dereference)$ are inverse of each other

```
int i=5;
int *ptr;                    /* pointer declaration */
ptr = &i;
printf("%p %d", ptr, *ptr);   /* pointer dereferencing */
```

A SIMPLE TASK

- Let's download *ex1.c* and try to complete the tasks.

```c
#include <stdio.h>

int main() {
int *ptr1, *ptr2,  m = 100, n ;

/*  Store address of m in ptr1
Store address of n in ptr2  */

/*  n = m + 3;
do the equivalent operations using pointers */

/*Print the value addresses and the values */
printf("ptr1 address: %p ptr2 address: %p \n", ptr1, ptr2);
printf("*ptr1: %d *ptr2: %d\n", *ptr1, *ptr2);
printf("m: %d, n: %d\n", m,n);

return 0;
}
```

Quick Tip

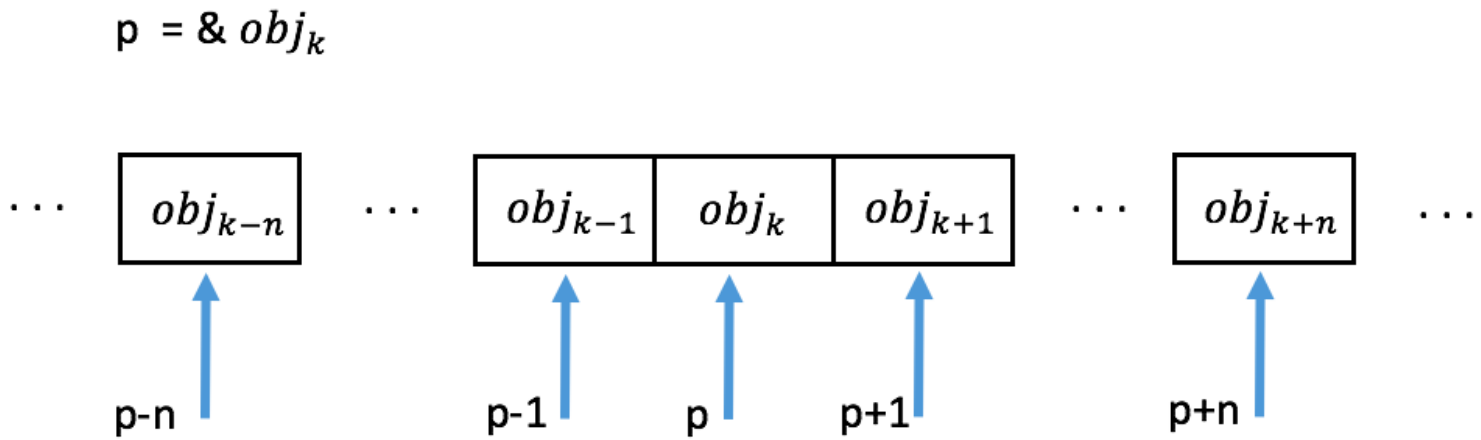
- You can remember the functionalities of the address and dereferencing operators as:
  - & (address of)
  - * (the value at that address)
- Also,please note that below lines are equal to each other. We can say that & is the inverse of *.

```
int *ptr, a=3, b;
ptr = &a;       /* ptr points to a   */
b = *ptr;       /* fetch the value that ptr referencing    */
b = *(&a);      /* same as above line    */
b = a;
/* last three lines are equivalent*/
```

Pointer Arithmetic - Format

- Addition/Subtraction an integer from a pointer.
- Subtraction of a pointer from a pointer.
- We can compare two pointers.

- We will see the examples of pointer arithmetic on arrays.



Operator Precedence

- Address & and dereferencing * operators have equal precedence
- You have to be careful when you mix * with ++ or - -
  - *++cp interpreted as *(++cp)
  - *cp++ interpreted as *(cp++)
- Second one fetches the value then increment pointer, the same way with the j = i++;
- (*cp)++ increments the value pointed by cp.
- Have a look at ▸ Operator Precedence to see full reference for C.

```
int c[2] = {3,5};
int *cp = c;

int d = *++cp; /* try them*/

printf("%d %d\n", d, *cp);
```

Pointer - Array Equivalence

- We can use pointer arithmetic to perform array operations.
- An array variable is actually a pointer that holds the address of first element in the array.
- These are basically same:
  - array[n]
  - *(array+n)
- Let's download *ex2.c* and examine it.

Functions and Pointers

- The examples that we saw until today was pass by value.
- We can use pointers to pass parameters by reference.
- Let's examine the codes below:

Code Samples

```
void inc(int *p){
   (*p)++; /* increment what p ↩
      is pointing to */
}

int main( ) {
   int x = 7;
   inc(&x);
   printf("%d",x);
   return 0;
}
```

- What if p is assigned to null after increment in the function?

```
void swap(int *xp, int *yp){
   int temp = *xp;
   *xp = *yp;
   *yp = temp;
}

int main( ) {
   int x = 3, y = 5;
   swap(&x, &y);
   printf("%d %d", x, y);
   return 0;
}
```

- What if int x, int y in function definition (instead of pointer)?

Sending Array to Functions

- We can also send arrays as function parameters by using this property.
- Let's download *ex*4.*c* and *ex*5.*c* and examine it. Note that below lines are basically the same thing:
  - double getAverage(int *arr, int size);
  - double getAverage(int arr[], int size);

POINTER TO A POINTER

- We can also define a pointer to another pointer

```
int main( ) {
   int num = 5;
   int *x   = &num;
   int **y  = &x;

   printf("%d %d", *x, **y);
   return 0;
}
```