C Pointers & Abstraction

**Pointer**
- Pointing to a variable and, pointing to the address of the variable that stores the value.
- The pointer is a variable itself. By changing the value of the pointer, You can make the variable A point to variable B.
- Only the address value can be stored in the pointer variable.
- int *p : p is an integer type points.

**Basic Example**

```c
#include <stdio.h>

int main()
{
    int i, j;
    int *p1, *p2;   // int type pointer p1, p2 declaration

    i = 7;
    p1 = &i;        // assign the address of i to p1

    printf("*p1 : %d\n",*p1);  // *p1 : 7
    printf("*&i : %d\n",*&i);  // *&i : 7 (something pointed by the address of i)

    j = 10;
    p2 = &j;        // assign the address of j to p2

    printf("*p2 : %d\n", *p2);
    printf("*&j : %d\n", *&j);

    *p1 = 3;        // assign 3 to the something pointed by p1(i)
    *p2 = *p1;      // assign 3 to the something pointed by p2(j)

    printf("i, j : %d, %d\n", i, j);
}
```

DCSLab
CSE, SNU
Pointer and Array

- In C/C++, array name is equal to the address of variable. The first element address of array A is &A[0], and it means A=&A[0].
- If it is declared like “int A[3]={1,2,3};”, compiler translates this command like that.
  1. Secure the three consecutive 4byte.
  2. Input 1,2,3 to the each 4 byte.
  3. Create int pointer A
  4. Initialize the value of the pointer A to the first array element of the array
- If pointer value increase one, the address increases as much as size times data type pointed by pointer, so that pointer points the next first element of the array.
- Pointer + n = address increase by n (X)
- Pointer + n = the location moved from the current location of array by n elements (if it is int type, then 4)
- Current location + n*sizeof(int)
  - \( \text{A}[i] = *(\text{A}+i) \)

```c
#include <stdio.h>

int main()
{
    int i, *p;
    int a[5] = {1, 2, 3, 4, 5};
    int sum1 = 0, sum2 = 0, sum3 = 0;

    for (i = 0; i < 5; i++) {
        sum1 += a[i];          // 1 + 2 + 3 + 4 + 5 = 15
        printf("&a[%d]=%x\n", i, &a[i]); // the address of each element in array a
    }

    for (p = a; p < &a[5]; p++) // assign the first address of a to pointer p
        sum2 += *p;

    for (i = 0; i < 5; i++)
        sum3 += *(a + i);
    // add the space occupied by one array element (Since the type is 4 byte type, the address of (a+1) is “the address of a+4”, that is, a[1])
    printf("%d %d %d\n", sum1, sum2, sum3);

    return 0;
}
```
## Pointer operators (Use of the * and ++/-- Operators)

<table>
<thead>
<tr>
<th>Pointer</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>*p+1</td>
<td>Add the value 1 to the data store in the address pointed by *p</td>
</tr>
<tr>
<td>*(p+1)</td>
<td>Data stored in the address that is added the value sizeof(type) to the address pointed by *p</td>
</tr>
<tr>
<td>*p++</td>
<td>Get the data stored in the address pointed by *p, and go to next address</td>
</tr>
<tr>
<td>*p += 2</td>
<td>By increasing 2 from *p, pointing to the element that is behind 2 from current position</td>
</tr>
<tr>
<td>*p -= 3</td>
<td>By decreasing 3 from *p, pointing to the element that is ahead 3 from current position</td>
</tr>
</tbody>
</table>

```c
#include <stdio.h>
```
int main() {
    int a[5] = {1, 3, 5, 7, 9};
    int *p, *q;
    p = a;

    /* Unary Operator : ++/-- */
    /* ++i: added first, and then using value */
    /* i++: used first, and then add operation is executed */

    /* Right to Left Association Rule */
    printf("%d %d\n", *++p, *p);      /* = *(++p), *p */
    printf("%d %d\n", ++*p, a[1]);    /* = ++(*p), a[1] */
    printf("%d %d\n", *p++, a[1]);    /* = *(p++), a[1] */
    printf("%d %d\n", (*p)++, a[1]);  /* != *p++, a[1] */
    printf("%d %d\n", *p, a[2]);
    return 0;
}

Print example

1. *++p = 3, *p=1
   The current value of array pointed by p *p=3
   a[5]=1 3 5 7 9

2. ++*p = 4, a[1]=3
   The current value of array pointed by p *p=4
   a[5]=1 4 5 7 9

3. *p++ = 4, a[1]=4
   The current value of array pointed by p *p=5
   a[5]=1 4 5 7 9

4.
Computer Programming  
(2014 Spring)

\[
(*p)++ = 5, \text{a[1]} = 4  
\text{The current value of array pointed by } p *p = 6  
\text{a[5]} = 1 4 6 7 9
\]

\[
*p = 6, \text{a[2]} = 6
\]

**Command-Line-Argument Example**

```c
/*arg.c*/
#include <stdio.h>

int main(int argc, char* argv[]) {
    int i;
    for (i = 0; i < argc; i++)
        printf("%s %c\n", *(argv + i), *argv[i]);
    printf("\n");  
    printf("%c\n",***argv[0]);
    printf("%c\n",*(++argv[0]));
    printf("%c\n",**(++argv[0]));
    printf("%c\n",**(++argv[0]));
    return 0;
}
```

**Pointer and Function Argument**

- **Call by Value**
  - If function is invoked, memory area is created for delivering argument value, and then argument value is copied to new area.
  - Even though delivered argument value is changed in callee, it doesn’t change the value of caller.

```c
/* Call by Value */
#include <stdio.h>

void swap(int, int);
```
```c
int main() {
    int x = 1, y = 2;
    printf("x : %d, y : %d\n", x, y);
    swap(x, y);
    printf("x : %d, y : %d (caller)\n", x, y);
    return 0;
}

void swap(int x, int y) {
    int temp;
    temp = x;
    x = y;
    y = temp;
    printf("x : %d, y : %d (callee)\n", x, y);
}

◆ Call by Reference
1. If function is invoked, the address of argument is delivered.
2. So, In Callee, the variable declared in Caller can be manipulated.

/* Call by Reference */

#include <stdio.h>

void swap(int*, int*);

int main() {
    int x = 1, y = 2;
    printf("x : %d, y : %d\n", x, y);
    swap(&x, &y);
}```
```c
void swap(int* x, int* y) {
    int temp;

    temp = *x;
    *x = *y;
    *y = temp;

    printf("x : %d, y : %d (callee)\n", *x, *y);
}
```

```c
printf("x : %d, y : %d (caller)\n", x, y);
return 0;
}
```
Practice

1. Module Example: Check the output of program, by completing push and pop function.

(No need to pointer. By using return value of pop function, print the number in order of 'pop')

```c
/* Interface */
/* stack.h */
void make_empty();
int is_full();
void push(int);
int pop();

/* Implementation */
/* stack.c */

int contents[100];
int top = 0;
void make_empty(){
    top = 0;
}
int is_full(){
    return (top<100)? 0 : 1;
}
void push(int i){
    /* blank */
}
int pop(){
    /* blank */
}

/* main.c */
#include "stack.h"
Main(){
    int i;
    make_empty();
    for(i = 0; i<10; i++) if(!is_full())push(i+1);
}
```
for(i = 0; i<10; i++) printf("%d\n",pop()) ;
}

2. Make the mystrlen, mystrcpy function.

```
#include <stdio.h>

int mystrlen(char*);
void mystrcpy(char*, char*);

void main()
{
    char *str1="This is easy."
    char str2[256];
    int result;
```
3. Find out the sum of elements in the given array.
(except, manage the addition, by receiving the array with pointer directly.)

```c
#include <stdio.h>

int sumUp(int *A, int size)
{
    /*blank*/
}
```
```c
void main()
{
    int total;
    int score[5]={98,100,50};
    total=sumUp(score,3);

    printf("sum = %d\n",total);
}
```