To:
Tenouk

C LAB WORKSHEET 11
C & C++ Pointers Part 1: Playing With Memory Addresses

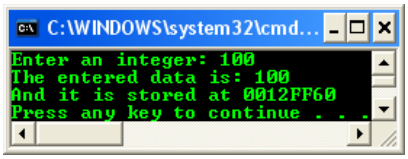
1. Understanding and using C/C++ Pointers.
2. Pointers and array.
3. Pointers and function.
4. Pointers and structure data type.
5. Tutorial references that should be used together with this worksheet are C & C++ pointers part 1 and C & C++ pointers part 2.

- When we create and declare a variable, for example:

```
int i = 77;
```

- The compiler assigns an **address** to that **variable**. i will be stored at one specific address in computer's memory system. The computer memory is normally Random Access memory (RAM). Which part of the RAM will be used to store the variable and other things declared and used in programs determined by the compiler and depending on the addressing mode used by processor. You can get the detail story for C/C++ compiler in Module W and Module Buffer overflow and see how the C/C.++ program elements stored in different part of the memory area. To understand the addressing used, you need to learn about the microprocessor and it is not in our scope of discussion.
- It doesn't matter whether it is variable or other C/C++ element, the most important thing here when you declare a variable, memory storage with address is reserved for it. The memory address is fixed and also depends on the installed size but the **data** place in the variable can be changed from time to time during the execution. For the above C/C++ statement, data is placed in i temporarily and may be used later, during the program execution.
- Here, it can be concluded that the **name of the variable** and its **address is fixed** once the program starts executing, but its **contents** may change. In programming or computer system the memory address normally represented by hexadecimal number. For example you already used the & (address-of operator) to display variable's address and to store data in the specified variable (such as in scanf()/scanf_s()).

```
#include <stdio.h>

void main(void)
{
    int myvar;
    printf("Enter an integer: ");
    scanf_s("%d", &myvar);
    printf("The entered data is: %d\n", myvar);
    printf("And it is stored at %p\n", &myvar);
}
```



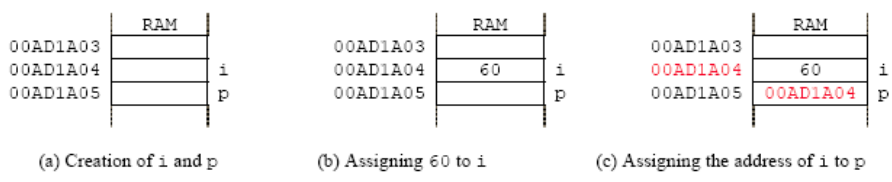
- For 32 bits system the 0012FF60 hexadecimal will be converted to 32 bits binary (one hex character represented by 4 binary digits).
- Just as we can create variables that store integers, characters and floats, we can also create variables that store the memory addresses of other variables. **Variables that store memory addresses instead of the actual data or values are called pointers**. An example of the pointer variable is shown below.

```
int i, *p;
```

- Here we have created two variables i and p. The similarity between them is that they both have something to do with integers. The difference between them, seen by the **asterisk**, is that i stores integer values, whereas p stores addresses of integers' locations. You can store a 60 in i, but not in p. Similarly, you can store the address of i in p, but not the address of an integer in i. **Integer variables** store **integer values** and **integer pointers** store the **addresses of integers**. Both of the following statements are legal:

```
i = 60; // storing an integer value
p = &i  // storing an address of an integer variable
```

- Let depict the previous statements in graphical manner.



(a) Creation of i and p (b) Assigning 60 to i (c) Assigning the address of i to p

- In the Figures, (a) shows that when the variables are created, i is stored at location AD1A04 and p is stored at location AD1A05. In (b) 60 is assigned to i, and in (c) the address of i, which is AD1A04 is assigned to p.

- We can also define pointer variables that store addresses to other **data types**, such as the following:

```
char a[1], *pa;
float x, *px;
```

- Here pa, can store addresses to character data types and px can store addresses to floats. Other combinations are not allowed. Here, both are correct.

```
pa = &a[0]; // storing a character address in a character pointer.
px = &x;    // storing a float address in a float pointer
```

- However the following statements are not legal.

```
pa = &x;    // pa is not a pointer variable for floats.
px = 3.40;  // px cannot store floating point values!
```

  - pa is a pointer to characters and can't store addresses of type characters. Also, x can store only addresses to floats. 3.40 is not an address.
  - Conventionally, names for pointer variables usually begin with the letter p, although they don't have to. Pointer names beginning with p reminds the programmer that this variable may be a pointer and can store only addresses.
  - 60 is an **integer constant**. The value of 60 will always be 60. i is an integer variable. Its value may be 60 for now, but later it could change. Similarly, AD1A04 is a **pointer constant**; it is the address of a location in RAM. The address of that location cannot be changed; it will always be AD1A04. However, p is a pointer variable; the address stored in it may change. Typically, the word "pointer" by itself means a pointer variable.

Pointer Arithmetic

- We can add to or subtract from pointer variables. However, we must remember that adding a number to a pointer does not necessarily add that many bytes but adds that **number of storage units**. In this text we will **assume** that characters are stored in one byte, integers in two and floats in four (you can use the sizeof() function to view the real size of the data type stored in your system). Bytes are used to measure the amount or size of memory.

```
int i, *p;
```

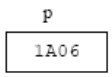
- Using our assumptions above, if 1 is added to a pointer integer, such as p, then 2 bytes will be added to the address that is already in it. Similarly if 1 is subtracted from p, then 2 bytes are subtracted from its contents. However, for every integer added to px, a pointer to floats, four is added to its contents because we are assuming that floats are stored using four bytes. In our case, one storage unit is 4 bytes for floats, 2 for integers and 1 for characters.

Arrays And Pointers Relation

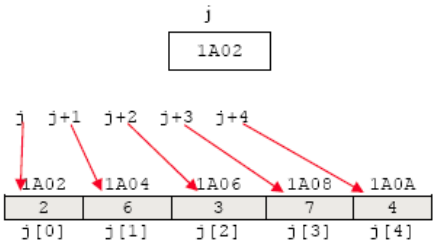
- When arrays are defined, the **array name** actually holds the **starting address of the array**. This string address is fixed because we can't move the array to some other location in memory that easily once the array is created. For example:

```
int *p, j[ ] = {2, 6, 3, 7, 4};
```

- Here, we have a pointer p and an array j[ ]. As seen in the following Figure the value of j is fixed at the address of 1A02 and can't be changed.
- **Pointer variable** can be changed.



- However **pointer constant** cannot be changed.



- The integer array is initialized with each slot and the initialization requires two bytes. When we execute this statement:

```
p = &j[2];
```

- The value of p becomes 1A06, the address of j[2]. We can change the value of p because it is a **pointer variable**, but we can't change the value of j, a **pointer constant**. Notice that the value of p is an address, whereas the value of j[0] is an integer. If we do this:

```
p = p + 1;
```

- Then the value of p will become 1A08 and if we do this:

```
p = j + 1;
```

- Then the value of p will become 1A04 because j is fixed at 1A02. However, we can't change j like:

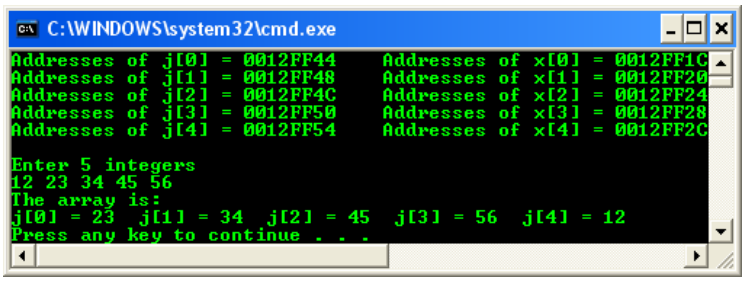
```
j = j + 1; // not legal
```

- Because j is an array and arrays are pointer constants.

Examples

Create an empty Win32 console application project named **mypointer**. Add a C++ source file named **mypointersrc**. Set your project **compiled as a C code**. Build and run the following code.

```
#include <stdio.h>

// function prototype
void ReadArr(int *x, int y);

void main(void)
{
    // p used to store addresses for array j[ ].
    int i, j[5], *p;
    // q used to store addresses for array x[ ]
    float x[5], *q = &x[0];
    // printing out the addresses of the two arrays.
    p = &j[0];  // line 1
    for(i = 0; i <= 4; ++i)
        printf("Addresses of j[%d] = %p\tAddresses of x[%d] = %p \n", i,  p + i, i,
&q[i]);
    // reading numbers into the integer array.
    printf("\nEnter 5 integers\n");
    // for older compiler may use scanf("%d", &j[4]); etc
    // notice there is no &
    scanf_s("%d", &j[4], 4);
    scanf_s("%d", j, 1);
    scanf_s("%d", j + 1, 1);      // j can't change and it hasn't
    p = &j[1];  //  same as p = j + 1;
    p = p + 1;
    // notice there is no &
    scanf_s("%d", p, sizeof(int));
    ReadArr(j, 3);
    // print the array
    printf("The array is: \n");
    for(i = 0; i <= 4; ++i)
        printf("j[%d] = %d  ", i, j[i]);
    printf("\n");
}

// function definition, to read a number into x[y]
// receive a pointer integer and integer arguments
// return nothing...
void ReadArr(int *x, int y)
{
    x = x + y;
    scanf_s("%d", x, 1);
}
```



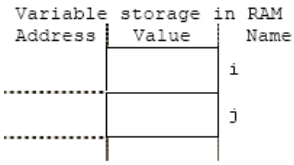
- The loop varies i from 0 to 4. In each iteration, it prints the address of the next slot in each of the two arrays. It does this by printing the value of p, which has the address of j[ ] and adding i to it each time. The addresses of the q[ ] array are printed by printing &q[i]. This gives the address of each slot. Notice that in the output, the addresses of the integer array go up by two and that of the float array go up by 4. On your computer the storage unit of each data type and the actual addresses may be different for example if you use and compile on 64 bits machine.
- Next, the program uses different ways of specifying the address of each slot in the j[ ] array. The addresses are passed to the scanf_s() function to read in values in the array slots. First, we read 4 into j[4] by passing &j[4]. Then a 2 is read into j[0] because j is passed to scanf_s(). j is the same as &j[0]. Similarly, j + 1 is passed to scanf_s() which is really &j[ ]. This reads in j[1]. Notice that you can add 1 to j, which will evaluate the expression to 1A04 as in the previous Figure. However, you can't change j to that.
- Now, the address of j[1] is stored in p, which is changed to p + 1 because p is a **pointer**, unlike j, which is a **pointer constant**. Since p contains the address of j[1], 3 is read into that slot. Last, j and 3 are passed to the function ReadArr(). The address given by j is now assigned to the pointer variable x. We could not change j in main(), but here, x is a variable and so it can be changed. y is 3, so storage units are added to 1A02 and x becomes 1A08. Reading a 7 at this address places the 7 in j[3]. Finally the program prints the array to show its contents.

More Pointers Practice

1. Remember that &i will give the address of variable i. Build and run the following program, show the output and answer the question.

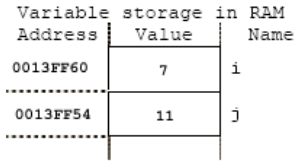
```
#include <stdio.h>

void main(void)
{
    int i = 7, j = 11;
    printf("i = %d, j = %d\n", i, j);
    printf("&i = %p, &j = %p\n", &i, &j);
}
```

As you may recall, each variable has a data type, name, value and address associated with it. In the following Figure, inside the boxes, under the label of "value", show the **values** of i and j. Likewise, on the lines under the label marked "Address", show the **addresses** of i and j in RAM from your output.



Keep in mind that those addresses are different from the shown on the left and may also different with your friends.

```
i = 7, j = 11
&i = 0013FF60, &j = 0013FF54
Press any key to continue . . .
```



2. Build and run the following program, show the output and answer the questions.

```
#include <stdio.h>

void main(void)
{
    int i = 7, j = 11;
    printf("i = %d, j = %d\n", i, j);
    printf("&i = %p, &j = %p\n", &i, &j);
    // reassign new values...
    i = 4;
    j = 5;
    // reprint...
    printf("\ni = %d, j = %d\n", i, j);
    printf("&i = %p, &j = %p\n", &i, &j);
}
```

a. After the initialization statement, were we able to change the values of the variables?
b. After the assignment statements, were we able to change the addresses of the variables? Try &i = 4;
c. Now at the end of main(), add the following statement and try running it: j = &i; Were we able to store the address of i into j?

```
i = 7, j = 11
&i = 0013FF60, &j = 0013FF54

i = 4, j = 5
&i = 0013FF60, &j = 0013FF54
Press any key to continue . . .
```

a. Yes.
b. No we can't. The &i = 4; generate the following error during the build time.

"...error C2106: '=' : left operand must be l-value..."

c. We add the following code at the end of main().

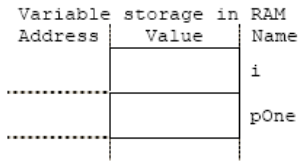
```
// assigning the address of variable i to normal variable j
j = &i;
printf("\ni = %d, j = %d\n", i, j);
printf("&i = %p, &j = %p\n", &i, &j);
```

```
i = 7, j = 11
&i = 0013FF60, &j = 0013FF54

i = 4, j = 5
&i = 0013FF60, &j = 0013FF54

i = 4, j = 1310560
&i = 0013FF60, &j = 0013FF54
Press any key to continue . . .
```

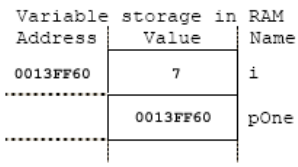
Yes we can store the address of i into j.

3. We need a new way to define a variable if that variable will store memory addresses or RAM locations. Notice the declaration of pOne using an asterisk. *pOne is a variable that can hold the memory addresses of integers. Show the output and answer the questions.

```
#include <stdio.h>

void main(void)
{
    int i = 7;      // Statement 1
    int *pOne;  // Statement 2
    float x = 0.00;
    printf("&i = %p \n", &i);
    pOne = &i;  // Statement 3
    printf("pOne = %p\n", pOne);
}
```




```
&i = 0013FF60
pOne = 0013FF60
Press any key to continue . . .
```



We can assign addresses or an integer to pOne.

```
&i = 0013FF60
pOne = 00000004
Press any key to continue . . .
```

However assigning an integer to pOne has no valuable purpose in programming because we don't know what is stored at the address and programming wise it is illegal.

d. We need to use an asterisk (*) to denote the variable can only hold the address of integer.

a. In the above diagram show the value of i inside its box and next to its address.
b. After Statement 3 is executed, what is the value of pOne? Show the value in the diagram.
c. Can you assign only addresses to pOne? Can you assign an integer as shown below to pOne?

```
pOne = 4;
```

d. From Statement 1 and 2, how can we tell beforehand that i can hold only integers and pOne can hold only the addresses of integer?
e. Try assigning an address of a float as shown below to pOne. Does it work?

```
pOne = &x;
```

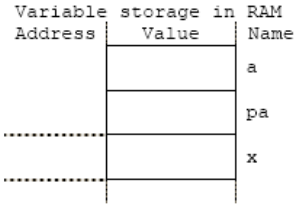
f. Does pOne have its own memory location in RAM, as shown in the diagram? If so, show the address where pOne is stored. Try:

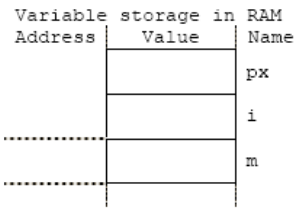
```
printf("&pOne = %p\n", &pOne);
```

4. The following experiment is a review of the previous experiments. Show the output and answer the questions.

```
#include <stdio.h>

void main(void)
{
    char a = 'Q', *pa;
    // the F modifier force the value to float
    // else by default it is a double
    float x = 7.3F, *px;
    int i = 2, *m;
    pa = &a;
    px = &x;
    m = &i;
    printf("pa = %p\npx = %p\n m = %p\n", pa, px, m);
}
```

a. Out of the six variables declared, which may store addresses?
b. In the following Figure, show the values of a, x and i.
c. From the output, show the values of pa, px and m in the Figure below.





d. Now that you know the values of pa, px and m, what are the addresses of a, x, and i? Show them in the Figure.
e. Must variable that store addresses begin with the letter p? If not, how can we tell from the declaration of these variables that they can store addresses?
f. Which variable(s) may store only (single) characters?
g. Which variable(s) may store addresses to characters?
h. Can you store the address of a float in pa? Can you store the address of an integer in pa? If not, what error or warning message is obtained? For example:

```
pa = &x;
pa = &i;
```

i. Can you store a character in pa? If not, what error message is obtained?

```
pa = 'Q';
```

j. Can you change the address of a variable? If not, what error message is obtained?

```
&a = 4440;
```

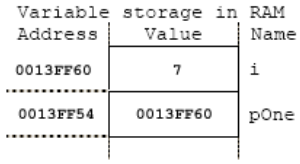
k. Try storing the addresses of a and i in px. Does px hold the addresses of

e. Yes it works with warning during the building.

"warning C4133: '=' : incompatible types - from 'float *__w64 ' to 'int *'"

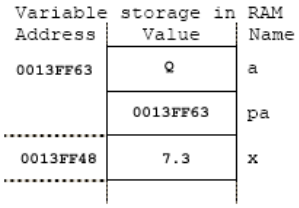
f. Yes pOne have its own memory location in memory.

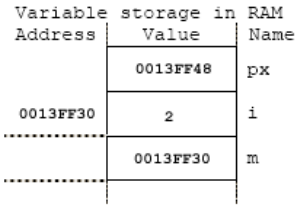
```
&i = 0013FF60
&pOne = 0013FF54
pOne = 0013FF60
Press any key to continue . . .
```




```
pa = 0013FF63
px = 0013FF48
 m = 0013FF30
Press any key to continue . . .
```

a. *pa, *px and *m will store addresses.





b.

e. Not really. However as a convention, pointer variables prefix with 'p' to denote it is a pointer variable. The asterisk (*) will denote the variable is a pointer variable.
f. Variable a.
g. Variable *pa.
h. Yes we can assign other pointer types. However this will deviate the purpose of declaring the original variable and programming wise it is illegal. The following are the warning generated. The code run without any error.

"warning C4133: '=' : incompatible types - from 'float *__w64 ' to 'char *'"
"warning C4133: '=' : incompatible types - from 'int *__w64 ' to 'char *'"

i. Same as (h) with the following warning message.

"warning C4047: '=' : 'char *' differs in levels of indirection from 'int'"

j. No we can't. The code generates the following error.

"error C2106: '=' : left operand must be l-value"

k. No it doesn't. However the following warnings generated.

"warning C4133: '=' : incompatible types - from 'char *__w64 ' to 'float *'"
"warning C4133: '=' : incompatible types - from 'int *__w64 ' to 'float *'"

l. No it doesn't. However the following warnings generated.

"warning C4133: '=' : incompatible types - from 'char *__w64 ' to 'int *'"

floats only?

```
px = &a;
px = &i;
```

l. Try storing the addresses of a and x in m. Does m hold the addresses of integers only?

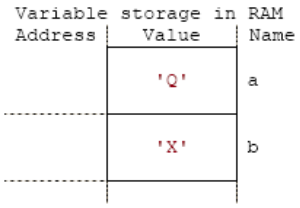
```
m = &a;
m = &x;
```

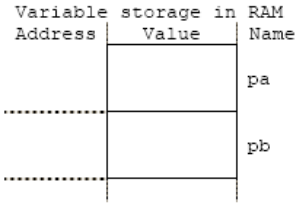
l. What can you store in a? What can you store in pa? What in their declarations identifies the difference?
m. What can you store in x? What can you store in px? What in their declarations identifies the difference?

5. Show the output and answer the questions.

```
#include <stdio.h>

void main(void)
{
    char a = 'Q', b = 'X', *pa, *pb;
    pa = &a;    // Statement 1
    pb = pa;    // Statement 2
    printf("pa = %p, pb = %p\n", pa, pb);     // Statement 3
    pa = &b;
    printf("pa = %p, pb = %p\n", pa, pb);
}
```

a. Can we assign a variable that stores addresses to another variable, like in Statement 2?
b. Can we store the address of any character variable in either pa or pb? How can you tell it?
c. Where is the variable 'a' stored? Where is the variable 'b' stored? Fill in the following diagram.





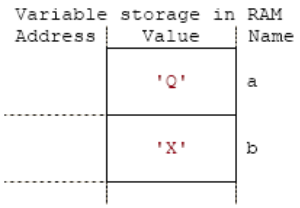
d. What is the first address stored in pa? Show it in the diagram.
e. Strike out that value you just placed for pa and write next to it the new address stored in pa.
f. From this experiment, can you tell where pa and pb are stored? If not, give the Statement that will print these addresses and copy those addresses in the diagram.

"warning C4133: '=' : incompatible types - from 'float *__w64 ' to 'int *'"

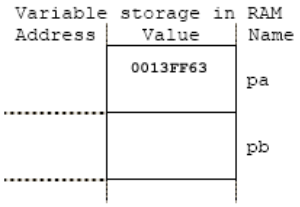
m. By avoiding any warning/error we can store a character in variable 'a' and we can store an address (memory location) in pa that point to a character data type.
n. By avoiding any warning/error we can store a float in variable x and we can store an address (memory location) in px that point to a float data type.

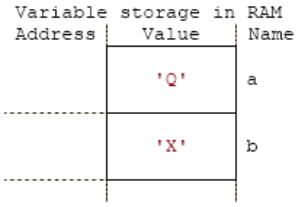
```
pa = 0013FF63, pb = 0013FF63
pa = 0013FF57, pb = 0013FF63
Press any key to continue . . .
```

a. Yes, provided both are pointer variables that point to a similar data type.
b. We can store address of any character variable in pa or pb because both pointer variables point to a similar data type, a char. Programming wise, in this case it is legal.

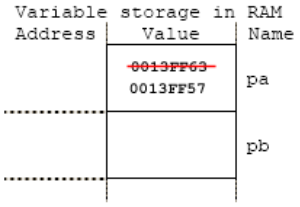


d.





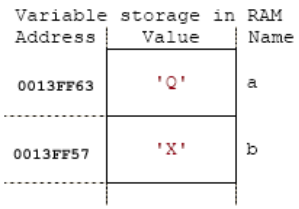
e.

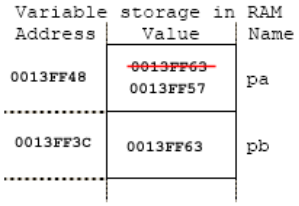


f. By adding the following two line of codes we can see a complete scenario when the program re-build and re-run.

```
// printing the data stored pointed by those pointers
printf("*pa = %c, *pb = %c\n", *pa, *pb);
// printing the addresses of those pointers
printf("&pa = %p, &pb = %p\n", &pa, &pb);
```

```
pa = 0013FF63, pb = 0013FF63
pa = 0013FF57, pb = 0013FF63
*pa = X, *pb = Q
&pa = 0013FF48, &pb = 0013FF3C
Press any key to continue . . .
```





www.tenouk.com
