To:
Tenouk

C LAB WORKSHEET 11a
C & C++ Pointers Part 2: Pointers, Array and Functions

1. Relation between pointers and array.
2. Pointers and functions.
3. Tutorial references that should be used together with this worksheet are C & C++ pointers part 1 and C & C++ pointers part 2.

6. Now let us see what operations we can do with variables that store memory addresses. First, let us have a little lesson in counting memory addresses by 1 in **hex**. Counting is done from 0 to 9, as we do normally. After 9, we count up to A, B, C, D, E, F and then 10 (one zero). Here, counting in memory addresses is done as a sample starting at FFDE. Adding 1 to FFDE gives us FFDF and adding 1 to that gives us FFE0 and so on.

... 00FFDE 00FFDF 00FFE0 00FFE1 00FFE2 00FFE3 00FFE4 00FFE5 00FFE6 00FFE7 00FFE8 00FFE9 00FFEA 00FFEB 00FFEC 00FFED 00FFEF 00FFF0 00FFF1 ...

Run the following program, show the output and answer the questions.

```
#include <stdio.h>

void main(void)
{
    char a, *pa;      // Statement 1
    pa = &a;          // Statement 2
    printf("pa = &a --> pa = %p \n", pa);
    pa = pa + 1;      // Statement 3
    printf("pa = pa + 1 --> pa = %p \n", pa);
    pa = pa + 3;      // Statement 4
    printf("pa = pa + 3 --> pa = %p \n", pa);
    pa = pa - 1;      // Statement 5
    printf("pa = pa - 1 --> pa = %p \n", pa);
}
```

a. Can we add an integer to a variable such as pa that stores addresses?
b. Can we subtract integers from a variable such as pa that store addresses?
c. Can we multiply integers to a variable such as pa that stores addresses as shown below? What error do you get?

```
pa = pa * 3;
```

d. What was the first address stored in pa in Statement 3?
e. After 1 was added to pa in Statement 3, what address was stored in pa?
f. After 3 was added to pa in Statement 4, what address was stored in pa?
g. After 1 was subtracted from pa in Statement 5, what address was stored in pa?
h. Now change only Statement 1 to the following statement:

```
int a, *pa;
```

i. Rerun the program and answer the same questions, namely:

   i. What was the first address stored in pa in Statement 3?
   ii. After 1 was added to pa in Statement 3, what address was stored in pa?
   iii. After 3 was added to pa in Statement 4, what address was stored in pa?
   iv. After 1 was subtracted from pa in Statement 5, what address was stored in pa?

j. Adding 1 to a variable that holds character addresses adds what number to the address?
k. Adding 1 to a variable that holds integer addresses adds what number to that address?
l. Since the answer to question j is 1, characters are stored in memory using only 1 byte. A byte is a unit of measure of memory. How many bytes are used to store integers on your computer? Take note that the number of bytes used to store characters, integers and floats varies depending on the type of computer or platform (e.g. 32 bits, 64 bits system etc.) and/or the target platform of your program. You can use the sizeof() function to check the size of variables as shown in the following program example.

```
#include <stdio.h>

void main(void)
{
    char a = 'W';
    int b = 100;
    float c = 1.234;
    double d = 100000.34;
    printf("Size of a = %d byte(s).\n", sizeof(a));
    printf("Size of b = %d byte(s).\n", sizeof(b));
    printf("Size of c = %d byte(s).\n", sizeof(c));
    printf("Size of d = %d byte(s).\n", sizeof(d));
}
```

```
pa = &a --> pa = 0013FF63
pa = pa + 1 --> pa = 0013FF64
pa = pa + 3 --> pa = 0013FF67
pa = pa - 1 --> pa = 0013FF66
Press any key to continue . . .
```

a. Yes, we can.
b. Yes, we can.
c. No we can't. The following error generated.

"error C2296: '*' : illegal, left operand has type 'char *'"

d. The address is 0013FF63.
e. The address is 0013FF64 (0013FF63 + 1).
f. The address is 0013FF67 (0013FF64 + 3).
g. The address is 0013FF66 (0013FF67 - 1).
h.

```
pa = &a --> pa = 0013FF60
pa = pa + 1 --> pa = 0013FF64
pa = pa + 3 --> pa = 0013FF70
pa = pa - 1 --> pa = 0013FF6C
Press any key to continue . . .
```

i. (i) The address is 0013FF60.

(ii) The address is 0013FF64 (0013FF60 + 4).
(iii) The address is 0013FF70 (0013FF64 + C where C = 12 in decimal). In this case 4 x 3 = 12 in decimal).
(iv) The address is 0013FF6C (0013FF70 - 4).

j. 1.
k. 4.
l. A character will take 4 bytes.

```
C:\WINDOWS\system32\cmd...
Size of a = 1 byte(s).
Size of b = 4 byte(s).
Size of c = 4 byte(s).
Size of d = 8 byte(s).
Press any key to continue . . .
```

7. The compiler knows that if we were to add 1 to a variable that holds addresses to integers, it will actually add 4 because the next integer will be four bytes away. Now find out how many bytes are used to store floating point numbers by observing the number added when 1 is added to variable that holds the addresses of floats. Change here only the keyword float in Statement 1.

```
#include <stdio.h>

void main(void)
{
    float a, *pa;     // Statement 1
    pa = &a;          // Statement 2
    printf("pa = &a --> pa = %p \n", pa);
    pa = pa + 1;      // Statement 3
    printf("pa = pa + 1 --> pa = %p \n", pa);
    pa = pa + 3;      // Statement 4
    printf("pa = pa + 3 --> pa = %p \n", pa);
    pa = pa - 1;      // Statement 5
    printf("pa = pa - 1 --> pa = %p \n", pa);
}
```

```
pa = &a --> pa = 0013FF60
pa = pa + 1 --> pa = 0013FF64
pa = pa + 3 --> pa = 0013FF70
pa = pa - 1 --> pa = 0013FF6C
Press any key to continue . . .
```

a. What was the first address that was stored in pa in Statement 2?
b. After 1 was added to pa in Statement 3, what address was stored in pa?
c. After 3 was added to pa in Statement 4, what address was stored in pa?
d. After 1 was subtracted from pa in Statement 5, what address was stored in pa?
e. How many bytes are used to store floats on your computer?
f. Adding 5 adds how much to a variable that holds addresses for a character?
g. Adding 5 adds how much to a variable that holds addresses for an integer?
h. Adding 5 adds how much to a variable that holds addresses for a float?
i. Adding 5 adds how much to a variable that holds addresses for a short int?
j. Adding 5 adds how much to a variable that holds addresses for a long int?
k. What about adding 5 to unsigned int?

a. The address is 0013FF60.
b. The address is 0013FF64 (0013FF60 + 4).
c. The address is 0013FF70 (0013FF64 + C).
d. The address is 0013FF6C (0013FF70 - 4).
e. 4 bytes.
f. 5 bytes (5 x 1 bytes).
g. 20 bytes (5 x 4 bytes).
h. 20 bytes (5 x 4 bytes).
i. 10 bytes (5 x 2 bytes).
j. 20 bytes (5 x 4 bytes).
k. 20 bytes (5 x 4 bytes).

8. Pointer and structure data type. Build, run and show the output. Then answer the questions.

```
#include <stdio.h>

// structure data type, you will learn this in another worksheet...
struct room
{
    float width;
    float length;
};

void main(void)
{
    struct room dining = {10.5f, 12.7f}, *d;
    d = &dining;
    printf("The width = %.2f The length = %.2f \n", dining.width, dining.length);
    printf("Variable pointer, d pointed to %p\n", d);
    d = d + 2;
    printf("d = d + 2, then variable pointer, d pointed to %p\n", d);
}
```

```
The width = 10.50 The length = 12.70
Variable pointer, d pointed to 0013FF5C
d = d + 2, then variable pointer, d pointed to 0013FF6C
Press any key to continue . . .
```

a. What is the name of the pointer variable?
b. This pointer holds addresses of what data type?
c. Variables of this data type take how many bytes to store? How can you tell by the definition of the structure?
d. What is the initial value of d, the pointer?
e. What is the value of d after 2 was added to it?
f. Do the answers in c, d and e agree? Please explain?

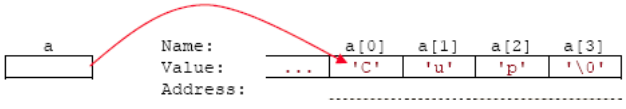
a. The name is d.
b. A structure data type that contains 2 float type elements.
c. It will take 8 bytes (2 x 4 bytes of floats).
d. Initial value is 0013FF5C.
e. The value is 0013FF6C (0013FF5C + 10 where 10 in hex = 16 in decimal). The 16 comes from 2 x 8 bytes of struct room.
f. Yes they agree. In c the struct room data type will take 8 bytes to store 2 floats of 4 bytes each. Then we add 2 to the pointer of struct room data type. In this case we need to add another 2 x 8 = 16 bytes to the original address. So 0013FF5C + 10 = 0013FF6C. 10 is in hex that equal to 16 in decimal.

9. Next, let turn our attention to the addresses of array elements. Show the output and answer the questions.

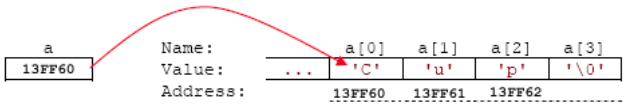
```
#include <stdio.h>

void main(void)
{
    char a[4] = "Cup";
    printf("a = %p\n&a[0] = %p\n&a[1] = %p\n&a[2] = %p\n", a, &a[0], &a[1], &a[2]);
}
```

```
a = 0013FF60
&a[0] = 0013FF60
&a[1] = 0013FF61
&a[2] = 0013FF62
Press any key to continue . . .
```

a. Does a print an address? Show its value in the following Figure.



Answer for a. and b.



b. Fill in the addresses in the Figure.
c. Is a the address of the first slot of the array, namely, &a[0]? Is the following equal:

a == &a[0]?

d. How many bytes is a[1] away from a[0]?
e. Elements in an array are stored next to each other. Hence, how many bytes does it take to store each character in an array of characters?
f. If the number of bytes needed to store one character is 1 as known previously, then adding 1 to the address of a should give you &a[1], the address of a[1]. Are these two addresses the same?

printf("a + 1 = %p\n&a[1] = %p\n", a + 1, &a[1]);

If they are, then the condition a + 1 == &a[1]? is true.

g. And since that is true, this should also be true:

printf("a + 2 = %p\n&a[2] = %p\n", a + 2, &a[2]);

Is this also true?

10. Build and run the following code. Answer the questions.

```
#include <stdio.h>

void main(void)
{
    char a[4] = "Cup";
    printf("a = %p\n&a = %p\n", a, &a);
}
```

a. You have learnt previously that we passed addresses to the scanf()/scanf_s () function and when dealing with character strings, we could send either the name of the string or its address. Then according to the experiment, is the following true?

a == &a?

b. In conclusion, which of the following are true?

a == &a?
a == &a[0]?
a + 1 == &a[1]?

11. Let change the data type. Build and run the following code, show the output and answer the questions.

```
#include <stdio.h>

void main(void)
{
    int a[4];
    printf("a = %p\n&a[0] = %p\n&a[1] = %p\n&a[2] = %p\n", a, &a[0], &a[1], &a[2]);
    // notice the difference between value/data and address
    printf("\n&a[2] - &a[1] = %d, &a[1] - &a[0] = %d, &a[0] - a = %d\n", &a[2] - &a[1], &a
[1] - &a[0], &a[0] - a);
}
```

a. Which of the following are still true?

a == &a?
a == &a[0]?
a + 1 == &a[1]?

b. How many bytes are used to store a float on your computer? Does this agree with previous result from experiment # 6
c. Change the int to float in this experiment. Run it. Which of the following are still true?

a == &a?
a == &a[0]?
a + 1 == &a[1]?

d. How many bytes are used to store a float on your computer? Does this agree with previous results from experiment # 6?
e. Can you tell if the following statement is also true without running it? Write down your answer and then try it.

a + 1 == &a[0] + 1

12. For the following experiment, the data type used will not affect our conclusions. Here, we see that **arrays** and **pointers** are similar. Then we see if there is any difference between them. Build, run, show the output and answer the questions.

```
#include <stdio.h>

void main(void)
{
    char a[4] = "Cup";      // Statement 1
    printf("a = %p\na+1 = %p\n&a = %p\n&a[0] = %p\n&a[0] + 1 = %p\n&a[1] = %p\n",
 a, a+1, &a, &a[0], &a[0] + 1, &a[1]);
}
```

a. a[ ] is an array, so a can store addresses. We can also change the array variable into a pointer variable. Replace Statement 1 with the following two

c. Yes a is the address of the first slot of the array. In this case a == &a[0].
d. 1 byte.
e. In this case is 3 bytes (not include the null character).
f. Yes those addresses are same as shown in the following output.

```
a = 0013FF60
&a[0] = 0013FF60
&a[1] = 0013FF61
&a[2] = 0013FF62
a + 1 = 0013FF61
&a[1] = 0013FF61
Press any key to continue . . .
```

g. Yes it is true as shown in the following output.

```
a = 0013FF60
&a[0] = 0013FF60
&a[1] = 0013FF61
&a[2] = 0013FF62
a + 2 = 0013FF62
&a[2] = 0013FF62
Press any key to continue . . .
```

```
a = 0013FF60
&a = 0013FF60
Press any key to continue . . .
```

a. Yes based on the output as shown above.
b. All is true as shown in the following output.

```
a = 0013FF60,  &a = 0013FF60
a = 0013FF60,  &a[0] = 0013FF60
a + 1 = 0013FF61,  &a[1] = 0013FF61
Press any key to continue . . .
```

```
a = 0013FF54
&a[0] = 0013FF54
&a[1] = 0013FF58
&a[2] = 0013FF5C

&a[2] - &a[1] = 1, &a[1] - &a[0] = 1, &a[0] - a = 0
Press any key to continue . . .
```

a. All still true.

```
a = 0013FF54
&a[0] = 0013FF54
&a[1] = 0013FF58
&a[2] = 0013FF5C

a = 0013FF54,  &a = 0013FF54
a = 0013FF54,  &a[0] = 0013FF54
a + 1 = 0013FF58,  &a[1] = 0013FF58

&a[2] - &a[1] = 1, &a[1] - &a[0] = 1, &a[0] - a = 0
Press any key to continue . . .
```

b. 4 bytes. Yes it agrees with the previous result.
c. All still true.

```
a = 0013FF54
&a[0] = 0013FF54
&a[1] = 0013FF58
&a[2] = 0013FF5C

a = 0013FF54,  &a = 0013FF54
a = 0013FF54,  &a[0] = 0013FF54
a + 1 = 0013FF58,  &a[1] = 0013FF58

&a[2] - &a[1] = 1, &a[1] - &a[0] = 1, &a[0] - a = 0
Press any key to continue . . .
```

d. 4 bytes. Yes it agrees with the previous result.
e. Yes the statement is true.

```
a + 1 = 0013FF58,  &a[0] + 1 = 0013FF58
Press any key to continue . . .
```

```
a = 0013FF60
a+1 = 0013FF61
&a = 0013FF60
&a[0] = 0013FF60
&a[0] + 1 = 0013FF61
&a[1] = 0013FF61
Press any key to continue . . .
```

statements.

```
int *a, b;
a = &b;     // assign address of variable b to variable pointer a
```

Is there any difference in the printouts?

b. Between the two printouts, which of the following conditions are true? Complete the table.

Which conditions are true?	"a" is an array
	`int a[5];`
		"a" is a pointer
		`int *a, b;`
		`a = &b;`

`&a[1] == &a[0] + 1`?		
`a + 1 == &a[1]`?		

a.

```
a = 0013FF54
a+1 = 0013FF58
&a = 0013FF60
&a[0] = 0013FF54
&a[0] + 1 = 0013FF58
&a[1] = 0013FF58
Press any key to continue . . .
```

Yes there is a different. However both printouts are consistent in term of multiplier. In the first printout it is multiplier 1 that is for char and the second printout is multiplier 4 that is for int.

b.

Which conditions are true?	"a" is an array
	`int a[5];`
		"a" is a pointer
		`int *a, b;`
		`a = &b;`

`&a[1] == &a[0] + 1`?	TRUE	TRUE
`a + 1 == &a[1]`?	TRUE	TRUE

13. We saw in the table above one difference between declaring a variable as a pointer and declaring a variable as an array. Here is one more experiment.

```
#include <stdio.h>

void main(void)
{
    float a[4], *b, c;
    b = &c;     // Statement 1
    printf("b = %p\n", b);
}
```

a. Did Statement 1 execute or did it give an error?
b. Change Statement 1 to the following:

```
a = &c;
```

Try running it. Did it give an error? What was that error?
As a conclusion, both a and b store addresses, that is they are both pointers. However, the address stored in b (a pointer) can be changed, it is a **pointer variable**. The address of a (an array) cannot be changed, it is a **pointer constant**.

a.

```
b = 0013FF3C
Press any key to continue . . .
```

Statement 1 executed without error.

b. Yes there was error as shown below.

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

14. Pointers and function. Build, run, show the output and answer the questions.

```
#include <stdio.h>

// function prototype
// this function receive an array
void FlatOut(int x[ ]);  // Statement 1

void main(void)
{
    int a[5] = {1, 2};      // Statement 2
    // sending a pointer to the first address or
    // the address of the first array element...
    FlatOut(a);             // Statement 3
    printf("a[0] = %d, a[1] = %d\n", a[0], a[1]);
}

// the function definition
void FlatOut(int x[ ])     // Statement 4
{
    // assign some value to array's element...
    x[0] = 11;
    x[1] = 12;
}
```

a. Did the FlatOut() function receive the address of a[ ] or copies of its values?
b. Since it received its address, did FlatOut() change copies of array a[ ] or did it change only the locations where the array elements were stored? It changed the original locations where the array was stored because the final values of the array that were printed in main() were 11 and 12.
c. Change Statement 1 to the following:

```
void FlatOut(int *x);
```

And change Statement 4 to the following:

```
void FlatOut(int *x)
```

Run the program. Is there any difference in its execution?

d. Change Statement 3 to the following:

```
FlatOut(&a);
```

Is there any difference in the program's execution?

e. Change Statement 2 to the following:

```
int *a;
```

Is there any difference in the program's execution?

```
a[0] = 11, a[1] = 12
Press any key to continue . . .
```

a. FlatOut() function receives the address of s[ ].
b. It change the locations where the array elements were stored.
c.

```
a[0] = 11, a[1] = 12
Press any key to continue . . .
```

No there is no different from the previous result.

d.

```
a[0] = 11, a[1] = 12
Press any key to continue . . .
```

No different however there are warnings during the compilation as shown below.

"warning C4047: 'function' : 'int *' differs in levels of indirection from 'int (*__w64)[5]'"
"warning C4024: 'FlatOut' : different types for formal and actual parameter 1"

e. There are warnings during the compile time and the program cannot be executed.

"warning C4047: 'function' : 'int *' differs in levels of indirection from 'int **__w64 '"
"warning C4024: 'FlatOut' : different types for formal and actual parameter 1"

More Pointers Questions and Activities

For the following questions, use the declaration of variables shown below. You can try building a simple program using the given specification in order to answer the questions.

```
int i, j[5] = {4, 5, 6, 7, 8}, *ptr1 = &j[0], *ptr3;
float x[5] = {4.0, 5.0, 6.0, 7.0, 8.0}, *ptr2;
```

1. For each statement below, specify which are valid and which aren't.

   a. ptr1 = ptr1 + 3;
   b. j = j + 1;
   c. ptr1 = j + 1;
   d. ptr2 = ptr1;
   e. ptr1 = j[1];
   f. ptr1 = 2;
   g. i = ptr1;
   h. ptr3 = ptr1;
   i. i = j[2];
   j. ptr2 = x;
   k. ptr1 = ptr1[2];
   l. x = &ptr2[2];
   m. j = ptr1 + 3;
   n. ptr1 = &j[1];

   a. Valid.
   b. Valid.
   c. Valid.
   d. Not valid. Incompatible type. ptr2 is a pointer to a float but ptr1 is a pointer to an integer.
   e. Not valid. Try ptr1 = &j[1];
   f. Not valid. Assigning a constant to a pointer variable.
   g. Not valid. Assigning an address to a normal variable.
   h. Valid.
   i. Valid.
   j. Valid.
   k. Not valid. Try ptr1 = &ptr1[2];
   l. Not valid. Left operand must be left-value.
   m. Not valid. Left operand must be left-value.
   n. Valid.

2. Evaluate the address of each expression. The j[ ] array begins at A008 and the x[ ] array begins at 9008.

   a. &j[0]
   b. j
   c. j – 2
   d. j + 4
   e. &x[4]
   f. x + 3
   g. x – 3
   h. &x[2] + 3

   a. A008.
   b. A008.
   c. A006.
   d. A00C.
   e. 900C.
   f. 900B.
   g. 9005.
   h. 900D.