Pointer arithmetic

I was looking at what the '+' and '-' operators actually performs when applied to pointers and arrays, as well as other tricks, and wrote some programs to test it:

```
#include <stdio.h>

main()

{

    int Arr[16];

    unsigned int a0 = (unsigned int) &Arr[0];

    unsigned int a3 = (unsigned int) &Arr[3];

    printf("%u\n", a3 - a0);

    printf("%u\n", &Arr[3] - &Arr[0]);

}
```

The result is this:

```
12

3
```

This other program explores this feature and some others:

```
#include <stdio.h>

int main()

{

    int Arr[16];

    Arr[0] = 9;

    Arr[3] = 5;
```

```
    printf("Arr[3]:      \t%d\n",Arr[3]);

    printf("*(Arr + 3):  \t%d\n",*(Arr + 3));

    printf("*(3 + Arr):  \t%d\n",*(3 + Arr));

    printf("*(&Arr[0] + 3):\t%d\n",*(&Arr[0] + 3)); // &Arr[0] is the same as
Arr

    printf("3[Arr]:      \t%d\n",3[Arr]); // gets converted by compiler to
*(3 + Arr)

    printf("\n");

    // showing how + operation behaves when used with addresses of pointers
or arrays

    printf("Arr:         \t%u\n", Arr);

    printf("&Arr[0]:     \t%u\n", &Arr[0]);

    printf("&Arr[0] + 1: \t%u\n", &Arr[0] + 1);

    printf("&*(&Arr[0] + 1):%u\n", &*(&Arr[0] + 1));

    printf("sizeof(Arr): \t%d\n", sizeof(Arr));

    printf("sizeof(Arr[0]):\t%d\n", sizeof(Arr[0]));

    printf("sizeof(int):\t%d\n", sizeof(int));

    printf("\n");

    printf("sizeof(&Arr[0]):%d\n", sizeof(&Arr[0]));

    printf("sizeof(unsigned long):%d\n", sizeof(unsigned long));

    printf("\n");

   unsigned long ArrAddr = (unsigned long)&Arr[0];

    printf("ArrAddr: %u\n", ArrAddr);
```

```
    int *p;

    p = ArrAddr;

    printf("*p: %d\n", *p);

    printf("*(p+3): %d\n", *(p+3));

    printf("\n");

   // breaking operator overloading of +

    unsigned long ArrAddr3 = ArrAddr + 3;

    p = ArrAddr3;

    printf("ArrAddr3: %u\n", ArrAddr3);

    printf("*p: %d\n", *p);

    printf("\n");

    // doing manually what overloading of + does automatically

    ArrAddr3 = ArrAddr + 3*sizeof(Arr[0]);

    p = ArrAddr3;

    printf("ArrAddr3: %u\n", ArrAddr3);

    printf("*p: %d\n", *p);

    printf("\n");

    return 0;

}
```

The result is this:

```
Arr[3]:         5
```

```
*(Arr + 3):     5

*(3 + Arr):     5

*(&Arr[0] + 3): 5

3[Arr]:         5

Arr:            1587159312

&Arr[0]:        1587159312

&Arr[0] + 1:    1587159316

&*(&Arr[0] + 1):1587159316

sizeof(Arr):    64

sizeof(Arr[0]): 4

sizeof(int):    4

sizeof(&Arr[0]):8

sizeof(unsigned long):8

ArrAddr: 1587159312

*p: 9

*(p+3): 5



ArrAddr3: 1587159315

*p: 0

ArrAddr3: 1587159324

*p: 5
```

Posted on*April 15, 2013*Tagged*C interview questionsstructures in Cunions in C*Comments*No Commentson Unions and Structures*

Unions and Structures

The main difference between an union and a structure is that for a structure, all the structure members are stored contiguously. However, for an union the members are stored overlapping, i.e. one of top of each other.

```
#include <stdio.h>

int main()

{

    struct ex_struct {

        char element1;  // 1 byte

        int element2;   // 4 bytes

        long element3;  // 8 bytes

    };

    union ex_union {

        char element1;

        int element2;

        long element3;

    };

    struct ex_struct a;

    union ex_union b;

    printf("Size of structure: %d\n", sizeof(a));

    printf("Size of union: %d\n", sizeof(b));

    a.element1 = 'D';
```

```
    a.element2 = 59;

    b.element1 = 'D';

    b.element2 = 59;

    printf("a.element1 = %c\n", a.element1);

    printf("a.element2 = %d\n", a.element2);

    printf("b.element1 = %c\n", b.element1);

    printf("b.element2 = %d\n\n", b.element2);

    b.element3 = 0x333333333;

    b.element1 = 0x11;

    printf("b.element3 = 0x%x\n", b.element3);

    printf("b.element1 = 0x%x\n", b.element1);

    return 0;

}
```

This is the output:

```
Size of structure: 16

Size of union: 8

a.element1 = D

a.element2 = 59

b.element1 = ;

b.element2 = 59
```

```
b.element3 = 0x33333311


b.element1 = 0x11
```

Unions are useful when a storage location is needed to represent different data types, which is similar to the concept of polymorphism in C++.