

ECE 264 Advanced C Programming

Contents

1	Pointer	1
2	Swap Function	2
3	Pointer and Array	4

1 Pointer

```
#include <stdio.h>

int main(void)
{
    int    i = 7, *p;

    p = &i;
    printf("%s%d\n%s%p\n",
           "    Value of i: ", *p,
           "Location of i: ", p);
    return 0;
}
```

Output:

```
Value of i: 7
Location of i: 0xbf8328e0
```

The location (address) may change every time the program is executed.

```
/* reviewpointer.c */
/* based on page 250 and 251 in ABoC */
#include <stdio.h>
int main(int argc, char * argv[])
{
```

```

int i = 3;
int j = 61;
int * p = & i; /* p points to i */
double x = 7.82;

/* int * r = & x; */
/* warning: initialization from incompatible pointer type */

double * s = & x;
printf("*p = %d\n", (*p)); /* *p = 3 */
printf("*s = %f\n", (*s)); /* *s = 7.820000 */

* p = 42;
printf("i = %d\n", i); /* i = 42 */

p = & j;
* p = 96;
printf("j = %d\n", j); /* j = 96 */

* s = 0.15;
printf("x = %f\n", x); /* x = 0.150000 */

/* p = 2; */
/* warning: assignment makes pointer
   from integer without a cast */

/* p = & 3; */
/* error: lvalue required as unary '&' operand */

/* p = & (j + 11); */
/* error: lvalue required as unary '&' operand */

return 0;
}

```

2 Swap Function

```

void swap(int *p, int *q)
{
    int    tmp;

    tmp = *p;
    *p = *q;
    *q = tmp;
}

#include <stdio.h>

void    swap(int *, int *);

int main(void)
{
    int    i = 3, j = 5;

    swap(&i, &j);
    printf("%d  %d\n", i, j);    /* 5  3 is printed */
    return 0;
}

```

Using

```

void swap(int *p, int *q)

```

is important. The following swap function does not work

```

/* swap2.c */
#include <stdio.h>
void swap(int p, int q)
{
    int tmp;
    printf("%d  %d\n", p, q);
    /* 3 5 */
    tmp = p;
    p = q;
    q = tmp;
    printf("%d  %d\n", p, q);
    /* 5 3 */
}
int main(void)

```

```

{
    int    i = 3, j = 5;
    swap(i, j);
    printf("%d  %d\n", i, j);
    /* 3 5 */
    return 0;
}

```

3 Pointer and Array

When we want to pass an array to a function, we are effectively passing a pointer to the first element of the array. The following two examples are the same.

```

/* sum12.c */
#include <stdio.h>
double sum1(double doubleArray[], int numElem)
{
    int index;
    double arraySum = 0.0;
    /* do not assume sum will initialize to 0 */
    for (index = 0; index < numElem; index ++)
        {
            /* always use bracket, even for only one statement */
            arraySum += doubleArray[index];
        }
    return arraySum;
}

double sum2(double * doubleArray, int numElem)
{
    int index;
    double arraySum = 0.0;
    for (index = 0; index < numElem; index ++)
        {
            arraySum += doubleArray[index];
        }
    return arraySum;
}

int main(int argc, char * argv[])
{

```

```

double doubleArray[] = { 0.1, 1.5, 3.8, 2.7};
/* calculates the size of the array */
int numElem = sizeof(doubleArray) / sizeof(double);
printf("numElem = %d\n", numElem);
printf("sum1 = %f\n", sum1(doubleArray, numElem));
/* Do not use "sum1 = %d\n" */
printf("sum2 = %f\n", sum2(doubleArray, numElem));
return 0;
}
/*
Output
numElem = 4
sum1 = 8.100000
sum2 = 8.100000
*/

```

How does this work? Remember earlier we talked about data types. Each type has a size. C knows how what *distance* to move between elements based on the type. The following example shows that `char`, `int`, and `double` have different sizes. The base addresses may change when the program is executed but the distances (the last three numbers in each line) are always the same.

```

/* arrayaddr.c */
#include <stdio.h>
int main(int argc, char * argv[])
{
    const int arraySize = 10;
    int cnt;
    char   cha[arraySize];
    int    ina[arraySize];
    double doa[arraySize];
    unsigned int chbase = (unsigned int) cha;
    unsigned int inbase = (unsigned int) ina;
    unsigned int dobase = (unsigned int) doa;
    unsigned int chaddr;
    unsigned int inaddr;
    unsigned int doaddr;
    printf("base = %x %x %x\n", chbase, inbase, dobase);
    for (cnt = 0; cnt < arraySize; cnt++)
    {
        chaddr = (unsigned int) & cha[cnt];
        inaddr = (unsigned int) & ina[cnt];
        doaddr = (unsigned int) & doa[cnt];
    }
}

```

```

        printf("addr = %x %x %x diff = %d %d %d\n",
               chaddr, inaddr, doaddr,
               chaddr - chbase, inaddr - inbase,
               doaddr - dobase);
    }
    return 0;
}
/*
base = bf80d030 bf80cff0 bf80cf90
addr = bf80d030 bf80cff0 bf80cf90 diff = 0 0 0
addr = bf80d031 bf80cff4 bf80cf98 diff = 1 4 8
addr = bf80d032 bf80cff8 bf80cfa0 diff = 2 8 16
addr = bf80d033 bf80cffc bf80cfa8 diff = 3 12 24
addr = bf80d034 bf80d000 bf80cfb0 diff = 4 16 32
addr = bf80d035 bf80d004 bf80cfb8 diff = 5 20 40
addr = bf80d036 bf80d008 bf80cfc0 diff = 6 24 48
addr = bf80d037 bf80d00c bf80cfc8 diff = 7 28 56
addr = bf80d038 bf80d010 bf80cfd0 diff = 8 32 64
addr = bf80d039 bf80d014 bf80cfd8 diff = 9 36 72
*/

```

C's arrays do **not** know their sizes. This is a common problem We have to remember passing the size of the array. This is fixed in some newer languages, such as Java. Java's arrays know their own sizes. If you want to learn more about Java, you can take ECE 462 next semester.

Please remember, however, that C does **not** explicitly define the size of char or int (page 123 in ABoC). C does guarantee the following

```

sizeof(char) = 1
sizeof(char) <= sizeof(int).

```

The previous examples change single integers (swap) or pass arrays as arguments. What happens if we want to change the elements of an array inside a function? Consider the following example:

```

/* vectorscalar .c */
#include <stdio.h>
void scalarProduct(double doubleArray[], int numElem, double s)
{
    int index;
    for (index = 0; index < numElem; index ++)
```

```

        {
            doubleArray[index] *= s;
        }
    }

void printArray(double doubleArray[], int numElem)
{
    int index;
    for (index = 0; index < numElem; index ++)
        {
            printf("%7f ", doubleArray[index]);
        }
    printf("\n\n");
}

int main(int argc, char * argv[])
{
    double doubleArray[] = { 0.1, 1.5, 3.8, 2.7};
    /* calculates the size of the array */
    int numElem = sizeof(doubleArray) / sizeof(double);
    printArray(doubleArray, numElem);
    scalarProduct(doubleArray, numElem, 5.5);
    printArray(doubleArray, numElem);
    return 0;
}

/*
    0.100000 1.500000 3.800000 2.700000

    0.550000 8.250000 20.900000 14.850000
*/

```

As you can see, the scalar product can modify the elements because the **addresses** of the elements are passed to the function so the change is visible to the caller.

When you pass arguments to a function, consider whether the function changes any argument. If so, make sure the argument is passed as a pointer so that the change is visible to the caller of the function.