Lecture 06 - Arrays (last updated: 2023-09-28)

Array basics

Just like you can declare a single variable as static or automatic variable, you can declare an array as static or automatic array:

```
    int g[10];        // global array of 10 integers, each initialized to 0
                      // (all static variables are initialized to 0)

    void foo()
    {
        int a[4];     // automatic array of 4 integers, from a[0] to a[3],
                      // allocated in foo()'s stack frame

        a[0] = 100;
        a[3] = a[0] + 3;

        printf("%d\n", a[2]);  // a[2] is uninitialized; contains a random value

        a[4] = 104;  // out of bounds; code compiles, but result unpredictable

        // code continues ...
    }
```

Arrays can be explicitly initialized as follows:

```
    int a[3] = { 100, 200, 300 };  // 100, 200, 300 assigned to a[0], a[1], a[2]

    int a[] = { 100, 200, 300 };   // same as int a[3] = { 100, 200, 300 };

    int a[3] = { 100 };            // same as int a[3] = { 100, 0, 0 };

    int a[100] = {0};              // initialize all 100 elements to 0
```

The elements in an array are laid out contiguously in memory:

```
    int a[10];

    printf("%lu\n", sizeof(int));   // byte size of the type 'int' is 4
    printf("%lu\n", sizeof(a[0]));  // byte size of one element of a is 4
    printf("%lu\n", sizeof(a));     // byte size of the whole array is 40

    uint64_t x = (uint64_t) &a[0];  // address of a[0], cast to unsigned long
    uint64_t y = (uint64_t) &a[5];  // address of a[5], cast to unsigned long
    printf("%lu\n", y - x);         // the difference is 20
```

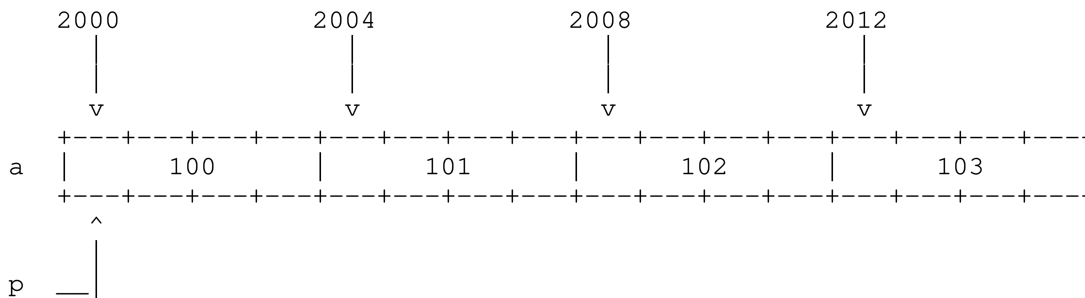
Pointer arithmetic

Consider the following declarations of an array and a pointer to the first element of that array:

```
    int a[4] = { 100, 101, 102, 103 };

    int *p = &a[0];
```

The following diagram shows the array in memory, assuming that a[0] is stored in the 4 bytes starting at the memory address 2000. (In reality, the address of a stack variable would be a much bigger number, but we will use a small number like 2000 for convenience.)



Now consider the following expression, where we add 1 to the pointer variable p:

```
p + 1
```

Adding the integer 1 to a pointer will yield a memory address that is sizeof(*p) bytes ahead of p. That is, in the example above, p + 1 will point to the memory address 2004, not 2001.

In general, adding (or subtracting) an integer N to (or from) a pointer will evaluates to an address, not N bytes away, but N * (the byte size of the underlying element type) bytes away.

In the example above, then, we can access any element a[i] using *(p + i). We can print the four elements of the array in various ways as follows:

```
for (int i = 0; i < 4; i++) { printf("%d\n", a[i]); }

for (int i = 0; i < 4; i++) { printf("%d\n", *(p + i); }

for (int i = 0; i < 4; i++) { printf("%d\n", *p++; }
```

In the last case, where we keep changing the value of p by incrementing it, p will end up at &a[4] at the end of the loop. That is, when the loop terminates, p will point to one PAST the last element of the array. Pointing to the element one past the last element is legal in C as long as the pointer is never dereferenced.

Array name as a pointer to the 1st element

In most expressions, an array name is converted to a pointer to the first element of the array. That is, in the example above, the following equivalences hold in most expressions:

```
a <==> &a[0] <==> p
```

(From now on, we will use "<==>" to say two expressions are equivalent.)

This means that we can also write *(a + i) in place of *(p + i):

```
for (int i = 0; i < 4; i++) { printf("%d\n", *(a + i); }
```

Let’s address a couple of subtle points before we move on to the next section where we will put everything together into the Grand Unified Theory (GUT) of pointers and arrays.

The conversion from an array name to a pointer to the 1st element also takes place when an array name is passed to a function. In fact, it is impossible to pass an array to a function in C because all array parameters are converted to pointer parameters:

```
int foo(int a[10]);
int foo(int a[5]);
int foo(int a[]);    // all 3 array parameter declarations are converted to

int foo(int *a);     // this pointer parameter declaration
```

In other words, when an array is passed to a function, the size of the array gets lost. If the function needs the size of the array, it needs to be passed as a separate argument. The following example illustrates this point:

```
// this function does NOT work
int get_num_elements(int a[])
{
    // does not work as intended because sizeof(a) is always 8;
    // a is a pointer in this function, not an array name

    return sizeof(a) / sizeof(a[0]);
}
```

Finally, here are a couple of expressions where the equivalence does NOT hold:

```
int a[10] = {0};
int *p = &a[0];

printf("%lu\n", sizeof(a));  // byte size of the whole array is 40

printf("%lu\n", sizeof(p));  // byte size of any pointer is 8

p++;                         // p will become &a[1]

// a++;                      // this doesn’t compile; a is an array name,
                             // not a variable, so it cannot be changed
```

Grand Unified Theory (GUT) of pointers and arrays

So we have the following equivalences for a and p, provided that p == &a[0]:

```
a[i] <==> *(p + i)

a    <==> &a[0]
```

The following equivalence can be easily deduced:

```
a[i] <==> *(a + i)  // because a <==> &a[0] <==> p
```

We can go even further to justify the use of "[]" for p:

```
p[i] <==> (&a[0])[i] <==> a[i] <==> *(p + i)
```

We finally arrive at the Grand Unified Theory (GUT) of pointers and arrays:

```
    +----------------------+
    |                      |
    |  u[i] <==> *(u + i)  |  where u is either an array name or a pointer
    |                      |
    +----------------------+
```

In fact, C defines e1[e2] to be *(e1+e2), where one of the two expressions, e1 and e2, is a pointer and the other is an integer. (Note that it does not say e1 is a pointer -- it can't because e1+e2 must be equivalent to e2+e1 -- which means an expression like "0[a]" must be valid!)

Note that i in GUT can be a negative number. Assuming p == &a[5]:

```
    p[-2] <==> *(p - 2) <==> *( &*(a + 5) - 2 )
                        <==> *(   (a + 5) - 2 ) <==> *(a + 3) <==> a[3]
```

The definition of a difference between pointers follows naturally from the addition of a pointer and an integer. Assuming p0 == &a[0] and p2 == &a[2]:

```
    p2 - p0 <==> &a[2] - &a[0] <==> &*(a + 2) - a <==> a + 2 - a <==> 2
```

As an exercise, convince yourself that all of following are equivalent:

```
      a[0]
      *a
      *&a[0]
      *(a+0)
```

So are all of the following:

```
      a+5
      &a[5]
      &a[0]+5
```