Lecture 11

CSE 110

16 July 1992

1 Examples of Pointer Arithmetic

In each of the following examples, suppose that `array` is an <array of 3 `int`s>, whose elements contain the values 5, 23, and 119, respectively. Suppose `p` and `q` are pointer variables of type <pointer to `int`>, which point initially to the first element of the array `array`, and that `d` is an <`int`> variable. Then:

- This statement

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

  computes the value of `p + 1`, which is a pointer to `array`'s second element (by the pointer arithmetic rule), and assigns that pointer value to the variable `q`. `q` now points to `array`'s second element. `p` has not changed.

- The statement

  ```
  q = p++;
  ```

  gets the value of `p`, assigns that value to `q`, and makes a note to bump up the value of `p` by one before the statement is over. So `q` ends up pointing to the first element of `array`, and `p`, which got bumped up, points to the the second element of `array`.

- This statement is illegal:

  ```
  q = *(p + 1);
  ```

  because `p` is a <pointer to `int`>, and so `p + 1` is also a <pointer to `int`>— it points to the element after the one that `p` points to. Thus `*(p + 1)`

is the thing that `p + 1` points to, and is therefore an <`int`>. `q`, on the other hand, is a pointer variable. You can't store an <`int`> in a pointer variable, because pointers are not <`int`>s. The compiler will refuse to compile this statement.

- Perhaps what we meant to write in the last example was this:

```
*q = *(p + 1);
```

This gets the value of the element after the one `p` is pointing to, and assigns that value to the element that `q` is pointing to. `q` is pointing to the first element of `array`, so that first element gets assigned the value 23. Neither `p` nor `q` changes; both are still pointing to the first element of `array`. But `array` itself now contains the values {`23, 23, 119`}.

- Or, perhaps, what we meant to say in the incorrect example was this:

```
d = *(p + 1);
```

`d` is an <`int`>, `*(p + 1)` is an <`int`>, so things work out. `p + 1` points to the second element of `array`, and `*(p + 1)` is this second element itself. Since the expression `*(p + 1)` is in a value context, it's evaluated to produce a value, and the result is the contents of the second element of `array`, which is 23. This value is assigned to the <`int`> variable `d`. The value of `p` does not change.

- On the other hand, this is something else again:

```
d = *p + 1;
```

`p` points to the first element of `array`, so `*p` gets the value of that element, 5. Then the computer adds 5 and 1 and assigns the result, 6, to the variable `d`. Neither `p` nor `*p` change.

- Unary operators like `++` and `*` take precedence from right to left. That means that if you see `*p++`, it means `*(p++)` and not `(*p)++`. What's the difference between these two expressions?

```
d = *(p++);
```

says to get the value of `p`, which is a pointer to the first element of `array`, and to make a note to bump up `p` by the end of the statement. Then, the computer finds the thing that `p` is pointing to, which has value 5, and assigns that value to `d`. After the statement, `d` got the value 5 and `p` was bumped up to point to the second element of `array`. None of the values in `array` changed. On the other hand,

Copyright ©1992 Mark-Jason Dominus All rights reserved.

```
d = (*p)++;
```

says to get the thing that `p` points to, namely the first element of `array`, get its value, and make a note to bump that value up by 1 before the end of the statement. The value of the first element of `array` gets assigned to `d`, and then the value gets bumped up by 1. So `d` gets the value 5, and `p` hasn't changed—it still points to the first element of `array`, whose value is now 6 instead of 5.

In short: `p++` means to bump up `p` so that it points to the next element in the array. `(*p)++` means to bump up the value of the thing `p` points to, but to leave `p` itself unchanged so that it points to the same place.

2 Too Much Work

C has a philosophy that it won't perform any operation unless that operation is completely trivial. For example, there is no way to say "Add the value 3 to every element of the array `a`." You can do it, but you have to code the loop yourself. That's because there is usually more than one way to perform any nontrivial operation, and C does not want to be in the position of having to figure out what way is best for your particular application. C lets the programmer decide what method is best.[1]

Accordingly, there is no operation in C that operates on all the elements of an array at once. You always have to use a loop. Operating on all the elements of an array at once is "Too Much Work." You can get C to do a little bit of work at a time, working on one array element, and then the next, and then the next, but that is the way you have to say it.

In fact, C goes even farther than that. Even manipulating an entire array at once is "Too Much Work". In C, there are no array values.

2.1 No Array Values

This is a little surprising. If `i` is an <`int`> variable, `i`'s value is an <`int`> value. If `pc` is a <pointer to `char`> variable, `pc`'s value is a <pointer to `char`> value.

[1] This philosophy is why there is no automatic run-time array bounds checking in C: If it were there and your program needed to run quickly, you would be out of luck, because there would be no way to take the bounds checking out. On the other hand, if you really did want it there, and you were willing to pay the speed penalty, you could write the code for the array bounds checking yourself. There is a trade-off between speed and safety, and C does not presume to know which one you value more highly.

Copyright ©1992 Mark-Jason Dominus All rights reserved.

But: If `af` is an <array of `float`> variable, `af`'s value is *not* an <array of `float`> value, because there are no array values.

So suppose `af` is an <array of `float`>s; what is the value of the expression `a`?

2.2 The Array Value Rule

No expression has a value which is an array. If `a` is an <array of thing>, then the value of `a` is a pointer to the first element of the array `a`, and has type <pointer to thing>.

2.3 Implications for Pointer Arithmetic

The array value rule says that the value of an array `a` is a pointer to `a`'s first element. That means that in a value context, where `a` is being evaluated to produce a value, the expressions `a` and `&a[0]` are interchangeable. The first denotes the array itself, and the second is a pointer (`&`) to `a`'s first element (`a[0]`), but the values of the two expressions are the same. The two expressions therefore behave differently from one another only in an object context.

Now, the pointer arithmetic rule tells us that if `p` points to the first element of an array, say to `a[0]`, then the expression `p + 1`, by definition, is a pointer which points to the second element of that array, namely `a[1]`, and that `p + n`, in general, points to `a[n]`.

The expression `a` is just such a pointer—the array value rule says that it points to the first element of the array `a`. Therefore, `a + n` points to the element `a[n]`.

Since `a + n` is a pointer to the element `a[n]`, the expression `*(a + n)` denotes the element `a[n]` itself. The expression `*(a + n)` is an lvalue expression, because it represents an object: the element of array `a` with index `n`.

Here's the punch line: The expression `a[n]` is in all ways completely identical with the expression `*(a + n)`. When the compiler needs to compile an array reference such as `a[n]`, it compiles it as though you had written `*(a + n)`. Array subscripting is only a notation convenience, and everything we want to do with array can be done with pointers instead, because of C's powerful pointer arithmetic convention.

Copyright ©1992 Mark-Jason Dominus All rights reserved.

3 Functions that Operate on Strings

We'll apply our powerful theory of arrays and pointers, and write some functions that operate on strings. First we'll write `strlen`, a function which takes a string as an argument and returns the number of characters in the string.

We'll be calling `strlen` this way:

```
char a[] = "Jean Ogrinz";
int length;

length = strlen(a);
```

Now, if we want to write `strlen`, we need to decide what its header looks like. Clearly it'll return an <`int`>. What's its argument? A string, which is an <array of `char`>, but that's not quite right. The argument to a function is in a value context, and so the `a` in `strlen(a)` is in a value context and is being evaluated to produce a value. The array value rule says that the value of an array in a value context is a pointer to the array's first element. That means that when `strlen` is really being passed is a pointer to `a`'s first element, which is a <pointer to `char`>. So `strlen` will begin like this:

```
int strlen(char *s)
```

here's the rest of `strlen`:

```
int strlen(char *s)
{
  int i = 0;

  while (*s != '\0') {
    i++;
    s++;      /* Make s point to the next character */
  }

  return i;
}
```

Copyright ©1992 Mark-Jason Dominus All rights reserved.

3.1 How `strlen` Works

When `strlen` starts up, `s` is initialized to point to the first character of the string we want to examine, as we've said. Since `s` is a <pointer to `char`>, `*s` is the <`char`>that `s` is pointing to.

We run through the `while` loop as long as `*s` is not the NUL character. Each time through, we increment `i`, which is a counter of how many characters we've seen so far, and we increment `s`, which makes it point to the next character in the string. When that character is the NUL character, we quit the loop and return the count.

Copyright ©1992 Mark-Jason Dominus All rights reserved.