Lecture 10

CSE 110

15 July 1992

1 Miscellaneous Details About Arrays

Here are some fine points of array use that we didn't talk about before.

1.1 Out-of-Bounds Array References

Question: `arr` has only 23 elements. In this code we're trying to store the value 5 into the 120th element. What happens?

```
. . .
int arr[23];

arr[119] = 5;
```

Answer: You get undefined behavior, so the compiler might choose to teleport elephants into the room, or to generate an executable program, which, when run, will teleport elephants into the room. Most likely, however, is that the compiler will say nothing at all, and will generate an executable program that will assign the value 5 to the place in memory where `arr[119]` *would* have been if `arr` had had that many elements. However, `arr` doesn't extend that far, and chances are that the compiler put some other variable at that spot instead, so that the `5` will obliterate some other piece of data.

You might ask why the compiler doesn't catch this for you, and the answer is that catching out-of-bounds array references at the time the program is compiled is impossible.[1]

[1] One can prove mathematically that there is *no* algorithm which, given the source code for a C program, can determine whether or not that C program will generate an out-of-bounds array reference when it is run.

You can catch out-of-bounds array references at the time the program is run, so that when you run the program, it aborts in the middle rather than doing a bogus array reference. Some languages, such as Pascal, do this automatically. But to detect the bogus array reference the computer must test the index value, every time it looks at any element of any array, to make sure it is not too big or too small, and that will make your programs slow. The C compiler takes the point of view that if you wanted to check to index value, and accept the corresponding slowness, then you'd write code yourself that tests the index value, and that if you didn't you wouldn't, and so washes its hands of the problem.

An out-of-bounds array reference can create a bug that is very difficult to find. If you store a value into an array element that isn't there, you'll clobber some other variable's value, and you might not find out about that until much later when you try to use the variable and discover it is full of garbage. So be careful.

1.2 The Expression `a[i]` is an Lvalue

The expression `a[i]` represents an object, namely the `i`'th variable in the array `a`, and is therefore an lvalue, and may appear on the left-hand-side of an assignment statement, or as the operand of the `&` operator. We've already seen array references in both contexts.

1.3 `[···]` has High Precedence

The `[···]` operator has higher precedence than *everything* else, so that `&a[i]` means `&(a[i])` and not `(&a)[i]`.

1.4 More About Strings

Suppose we have an array, and we want to perform some operation on every element of the array. We'd write a loop, of course; and loop over the elements in the array, performing the operation on the first one, and then on the second one, and soforth.

The question: How do we know when to stop?

Either, we have to remember how long the array was, or, we have to arrange that the last element of the array contains a special *sentinel* value that we can look for so that when we see the sentinel, we know we're done. Both methods are widely used. For strings, however, we always use a the sentinel value method, and the C language provides a little support for this.

When you write something like

```
char name[] = "Jean Ogrinz";
```

the compiler creates an array of *twelve* characters, not eleven. It initializes the first element of the array with the letter `J`, the second with `e`, and soforth, and it initializes the last element with a special sentinel value: the *NUL character*.

The C library provides many functions that operate on strings, and every one of them assumes that the strings you pass to it as arguments will end with the NUL character; that's how these functions know where to stop. For example, there's a function **`strlen`** which takes a string as an argument and returns the length of the string, and it does this by counting the characters in the string one at a time until it sees the NUL character.

So when I said a string was an array of characters, I fibbed a little. A string is an array of characters that is terminated with a NUL character.

Every character is represented internally by some integer between 0 and 255; for example, on typical machines, the character `A` is represented with the number 65. The details about what number represents what character are None of Your Business, with one exception: The NUL character is always, always, always the character which is represented by the number 0.

If you need to write the NUL character in your C program, you can write `'\0'`.

2 Declarations

The C declaration syntax was designed to be intuitive, but isn't.

The declaration

```
int i, *pi, ai[5], ii;
```

declares four things: `i`, an <`int`>; `pi`, a <pointer to `int`>; `ai`, an <array of 5 `int`s>; and `ii`, another <`int`>.

The way to remember this is that if you were to take any one of the expressions from the declaration and actually write it in your program, you'd have an <`int`>. So:

- `i` is an <`int`>.
- `*pi` is an <`int`>, and therefore `pi` itself is a <pointer to `int`>.
- `ai[5]` is an <`int`> (or would be if it weren't out-of-bounds; in any case, `ai[`something`]` is an <`int`>.), and therefore `ai` itself is an <array of `int`>)
- `ii` is an <`int`>.

This interpretation may come in handy when you're trying to understand a more complicated declaration, such as

```
char *argv[];
```

which says that `*argv[]` is a <`char`>, so therefore `argv[`something`]` must be a <pointer to `char`>, and `argv` itself must be an <array of pointer to `char`>.

C's declaration syntax has been widely criticized already, so let's try to live with it. We won't be seeing declarations much more complicated than this in any case.

3 Pointer Arithmetic

Having just had a section on C's declaration syntax, surely one of its worst features, we should try to recoup a little prestige and have a section on one of C's very best features. Here it is; it gets big type because it is so important:

3.1 The Pointer Arithmetic Rule

If p is a pointer which points to a certain element of an array, then the value of the expression p `+ 1` is, by definition, a pointer which points to the *next* element of the array.

3.2 An Example

```
float arr[53], *p *q;

p = &arr[4];
```

`p` is now pointing to the fifth element of the array `arr`. (Remember that `arr[4]` is the fifth element, because `arr[0]` is first and `arr[1]` is second.)

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

q is now pointing to the sixth element of the array `arr`. (Remember that `arr[5]` is the sixth element.) Then

```
 *q = 119;
  printf("%d\n", arr[5]);
```

will print `119`, regardless of what `arr[5]` was holding before.

3.3 Consequences of the Pointer Arithmetic Rule

Similarly, if *p* is a pointer which points to a certain element of an array, then the value of the expression *p* `+ 2` is, by definition, a pointer which points to the element after the next one in the array.

Similarly, if *p* is a pointer which points to a certain element of an array, then the value of the expression *p* `- 1` is, by definition, a pointer which points to the previous element in the array.

Similarly, if `p` points to the element `a[0]` of the array `a`, and if `n` is an <`int`>, then the value of the expression `p + n` is a pointer to the element `a[n]` of the same array `a`. Note what happens when the value of `n` is 0: `p + 0` points to the same place as `p`, as it should.

Similarly, if `p` points to an element of an array, and you do `p++;`, then afterwards, `p` points to the next element of the array.

This simple convention is at least partly responsible for C's vast popularity. C makes pointer manipulations easy and flexible by using this simple notation for talking about pointers to different parts of the same array.

We'll see how to use this tomorrow, when everything will finally fall into place.