Discussion #2: Arrays and Pointers

Written by Justin Hsia (6/23/2011)

Array Basics:

Can initialize in the following ways:

1) `type array_name[NUM];`
   Sets aside enough consecutive memory to hold `num*sizeof(type)`.
   `NUM` here must be a constant like 5. Arrays cannot be declared dynamically.
2) `type array_name[] = {data1,data2,data3,...,dataN};`
   Sets aside enough consecutive memory to hold `data1` through `dataN`.

You can think of the name of an array as `const type *array_name`, that is, a pointer you cannot change that holds the address of the head of a block of memory (the array). This means you can dereference it and use it as an address in pointer arithmetic, but you cannot assign to it. So once you declare an array, you cannot change its size.

The data access command `array_name[n]` is equivalent to `*(array_name+n)` and is valid for `n` between `0` and `NUM-1`. There is no bounds checking! No warnings will be thrown for trying to access `array_name[n]` for `n>=NUM` or `n<0`, but will generally cause errors.

Though sparingly used, C can do multi-dimensional arrays. See K&R p.110-112 if interested.

Pointer Basics:

Though kind of silly, this short Youtube video is a good intro visualization of pointers: http://www.youtube.com/watch?v=6pmWojisM_E.

What is a pointer?

`type *p, q, *r;` Here `p` and `r` are pointers and `q` is a normal variable.

A pointer is a variable that holds an address. The address in `p` is supposed to point to a location in the address space that holds data of type `type`. All pointers take up the same amount of space in memory, but the data they point to vary in size.

Pointers are a major source of coding errors. Assigning the address of a variable with an incompatible data type will generally lead to Bus Errors when you attempt to access that data. Attempting to access bad addresses (outside of address space or memory you don't own) usually leads to Segmentation Faults.

The address-of operator (&):

First of all, I'm not sure this is an official name, but "address-of operator" makes sense. Goes in front of a variable name and returns the address of that variable in memory, such as `p = &q`.

The dereference operator (*):

Note: Here I am referring to when the symbol '*' is used as an *operator*, that is, when it acts on a variable. When used in a variable declaration, such as `int *p`, the '*' symbol is part of the variable type (pointer to an `int`) and is NOT being used as an operator.

The dereference operator is used to access the data that a pointer points to (an address is a *reference* to something, so *dereferencing* it fetches the data).

NULL:

`NULL` is a symbolic constant that is specifically used for pointers. NULL evaluates to zero and essentially indicates a pointer to nothing.

Pointers and functions:

This is one of the big reasons for using pointers. To bypass the pass-by-value property of C functions, we can make permanent changes within a function by passing addresses to data and then directly accessing that data. In this case, the arguments, which are now addresses, are still passed by value, but assigning to the locations of those addresses makes changes that persist even after the function returns.

Check: Write the function `swap` that a caller could use to exchange the values of two `int` variables. Give a sample call to `swap`.

Pointer arithmetic:

Here we see the benefits of declaring variable and pointer types. Pointer arithmetic automatically takes the size of the data type into account, so incrementing a pointer by 1 actually increments the address by the size of the data that the pointer points to. All pointer arithmetic operations are automatically sized by the data size.

Valid operations include:

1) adding integers (+/-) to pointers,
2) subtracting two pointers within the same continuous chunk of memory,
3) comparing two pointers , and
4) comparing to `NULL`.

Pointers and structs:

Pointers are used often with structs, since structs have a tendency to be dynamically allocated. Two caveats about pointers and structs: (1) accessing a member of a struct that a pointer points to has its own operator (`->`) and (2) a struct is not allowed to include an instance of itself, so self-referential structs instead use pointers.

For `struct node {int val; struct node *next;} n` and `struct node *p = &n`: `p->val` is equivalent to `(*p).val` (we will see why the parentheses are necessary).

Check: Assume we have `int *p, int arr[], struct node {int val; struct node *next;} n, and struct node *sp = &n`.

Address or data?

```
p           /* address */          n           /* data */
&n          /* address */          arr[1]      /* data */
arr         /* address */          *sp         /* data */
&sp         /* address */          *(p + 2)    /* data */
&arr[9]     /* address */          *&*&n       /* data */
```

Pointers to pointers:

Like any other data type, pointers can point to other pointers. A declaration looks like:

`type **pp`, in which case `*p` is an address and `**p` is data.

Precedence and associativity of operators:

See Table 2-1 on K&R p.53 for full list. Of note:

1) Accessing operators (`[]`, `.`, and `->`) have higher precedence than addressing operators (`*` and `&`).
2) Addressing operators (`*` and `&`) have the same precedence as increment operators (`++` and `--`), but associate **right to left** (so `++*p` increments data, `*++p` increments pointer before returning data, and `*p++` returns data before incrementing pointer).

"Strings"

I use quotations here because C does not support an explicit data type called `string`, though it somewhat confusingly does have a library header called `string.h` and uses terminology such as "string constants." What "strings" actually are in C are arrays of characters that are terminated by the null character, such as `char str[] = "hello"`.

The null character (`'\0'`):

The null character, similar to `NULL`, evaluates to zero and is specifically used in character arrays. When `printf` is given the conversion character `%s` and a pointer to an array of characters, it will print out the characters until it finds the null character. This fact can be used to manipulate strings that exist within a large array of characters.

String constants:

A string constant (or string literal) is a sequence of characters surrounded by double quotes, as in `"hello, world!"`. A string constant is stored in memory as an array of characters, terminated by `'\0'` (so the array length is the # of characters + 1).

Note: String constants are storage class `static`, so attempting to change them will cause a seg fault.

Check: The function `strcpy` used below is defined in `string.h` (see K&R p.249).
Write different versions of the code below to achieve the following:

```
char str[30];
strcpy(str,"hello, world!");
/* insert line here */
printf("%s\n",str);
```

1) Prints "hello, world."
2) Prints "hello"
3) Potentially causes a memory error