Array Names and Pointers

Matt Bishop
Research Institute for Advanced Computer Science
NASA Ames Research Center
Moffett Field, CA 94035

Beginning C programmers are often told that an array name is a pointer. While true, this comparison is usually misunderstood, because array names cannot be manipulated in the same way pointers can.

Recall that a pointer to a character is defined by

```c
char *pointer;
```

and an array is defined by

```c
char arrayname[SIZE];
```

where SIZE is the number of elements of the array. The first declaration allocates space for the pointer, and none for the storage; if declared as a global, the value contained in pointer will be 0. The second declaration allocates SIZE characters for storage, and the name arrayname is the address of the zeroth element of that array. Note no storage is reserved for a pointer. This is the crux of the distinction: the name pointer contains the address of a character, and is a variable, whereas the name arrayname is itself the address of a character and is a constant, the value of the constant being determined at compile or run time (depending on how the compiler allocates storage.) So, a good rule of thumb is: an array name is a constant; a pointer is a variable.

What this means is that pointer arithmetic may be performed on pointer, since the arithmetic changes the value stored in that location. But since the name arrayname is a constant, its value cannot be changed by pointer arithmetic (or any other kind of manipulation, for that matter.) This distinction is often skimmed over, and beginning C pointers usually discover this property by accident.

It is time for some examples! First, we will look at a function to initialize a 4×1 matrix:
/* this is part of a larger program in file x.c... */

float vector[4];  /* the matrix */

vecinit()
{
    /*
    * put 1.0 into each of the elements
    */
    *vector++ = 1.0;
    *vector++ = 1.0;
    *vector++ = 1.0;
    *vector++ = 1.0;
    /*
    * now reset the value of "vector"
    * to what it was originally
    */
    vector = 4;
}

Notice that vector is declared as an array, with enough storage for 4 floating point numbers allocated. As we described above, the name vector refers to the address of the first element in the array, and so is a constant. But in lines 10 through 13, and again in line 18, this routine changes the value of vector by using pointer arithmetic! Hence, this part of the program will not even compile, and the resulting error messages are:

"x.c", line 10: illegal lhs of assignment operator
"x.c", line 11: illegal lhs of assignment operator
"x.c", line 12: illegal lhs of assignment operator
"x.c", line 13: illegal lhs of assignment operator
"x.c", line 18: illegal lhs of assignment operator

Let us change the routine slightly:
/* this is part of a larger program in file x.c... */

float vector[4]; /* the matrix */

vecinit()
{
  float *v; /* used to load vector */

  /*
    * set v to point to the first element in vector
    */
  v = vector;

  /*
    * put 1.0 into each of the elements
    */
  *v++ = 1.0;
  *v++ = 1.0;
  *v++ = 1.0;
  *v = 1.0;
}

Now we use a pointer, v, to load the initial values into vector. The value of vector is never manipulated or changed; only the value of v is. Since v is a pointer, it is a variable, and this manipulation is legal. In fact, the program now compiles without errors.

There is one exception to the “array name constant” rule: when an array is declared as a parameter in a function definition, in that specific case, it is identical to a pointer declaration. This seems like an annoying inconsistency, but in fact is not at all inconsistent. Recall that the compiler does not allocate storage for array parameters in function definitions; instead, it allocates a pointer to the base of the array. Let’s look at a sample program to be more specific.

Consider the following C function; it copies a string from its second argument to its first:
/* strcopy.c */

/*
 * copy the string in the array "from" to the array "to"
 */

strcpy(to, from)

cchar to[100]; /* copy to this string */
cchar from[100]; /* copy from this string */
{
  /*
   * just copy until you hit the end
   */
  while((*to++ = *from++) != '\0');
}

Note the declarations of the parameters from and to; they are declared as arrays. However, only a pointer to each is allocated (and as you would expect, the pointers are named to and from, respectively.) (In fact, unless your C compiler checks array bounds, it is customary to leave the size out of the declaration, so lines 7 and 8 would be written

    char to[]; /* copy to this string */
    char from[]; /* copy from this string */

rather than as above.) Now, since to and from are pointers to characters, they can be manipulated as pointers, and so the pointer arithmetic in line 13 is legitimate; compiling this function will produce no error messages.

In practise, since to and from are being treated like pointers rather than array names, C programmers would be most likely to write the declarations to reflect this:

    char *to; /* copy to this string */
    char *from; /* copy from this string */

However, this is a matter of taste, not a part of the language.

It is very easy to confuse array names and pointers, and figuring out the difference is something many C programmers have trouble doing. This article has tried to speed that process by explicitly describing the differences.