Arrays

Passing arrays to functions

A big topic for beginners is how to write a function that can be passed an array. A very common way of achieving this is done using pointers. This method can be seen all through the C core in functions like memcpy(). Another way, which is the more natural choice for C++, but not as popular with beginners and C programmers is to pass the variable as an actual array. Both methods have their advantages, however each has their drawbacks. How you need to use the array in your own code should determine which is best.

Pass an array using a pointer

An arrays name in code, without subscript operators [], is implicitly convertible to a pointer of its first element. An example of this is shown below. The pointer to arr[0] is assigned to the variable ptr. Contrary to popular belief, this is not a pointer to the array, and the array has implicitly been cast to a pointer, which is commonly referred to as *decayed pointer*, as it has been stripped of the information an array type provides. The example also shows how you can explicitly get the pointer of any array element.

```
char arr[] = { 1, 2, 3, 4, 5 };

char *ptr = arr;

//Or the equivalent ( change index to start mid-array ):
char *ptr2 = &arr[0];
```

Once we have a pointer to an element, you can also use pointer arithmetic to navigate the array. When passing an array to a function using pointers, you can convert it to a pointer using the two methods explained above. The next example shows this basic principle in the function setup():

```
#define MAX_LEN 5
char arr[ MAX_LEN ] = { 1, 2, 3, 4, 5 };

void Func( char *ptr ){

  for( int idx = MAX_LEN - 1 ; idx >= 0 ; --idx ){
    Serial.println( ptr[ idx ], HEX );
  }
}
```

```
void setup(){
  Serial.begin( 9600 );
  Func( arr );
}

void loop() {}
```

The above setup works, however the function is limited in the fact it can not be reused easily. To make the code more generic and increase its reuseability, we can remove the hard coded length and pass it through the function also.

```
char arr[] = { 1, 2, 3, 4, 5 };

void Func( char *ptr, int length ){

  while( length ){
    Serial.println( ptr[ --length ], HEX );
  }
}

void setup(){
  Serial.begin( 9600 );
  Func( arr, 5 );
  Serial.println( "----------------" );
  Func( &arr[2], 3 );
}

void loop() {}
```

Now the function allows arrays of any length you want. This method is almost always required to deal with an array of unknown size as the typical methods of retrieving the length of an array do not work with the pointer.

Pass a multidimensional array using pointer

The above example is a common and accepted way of dealing with array data. However this method is not as straight forward when using multidimensional arrays, for example, two dimensional array types do not convert to a 2D pointer (something like char **ptr). To pass a multidimensional array you must know the value of all dimensions except the first, so things become a little more restrictive.

This short example shows how a multidimensional array can be passed. The functions Func() & Func2() are two different styles used to pass a multidimensional array, and just like the examples above you can add in extra parameters to tell the function the length of each

dimension:

```
void Func( char array[][3] ){
  return;
}
void Func2( char (*array)[3] ){
  return;
}

void setup() {
  char arr[][3] = {
    { 0, 1, 2 },
    { 3, 4, 5 },
    { 6, 7, 8 }
  };
  Func( arr );
  Func2( arr );
}

void loop() {}
```

As you can see, with each extra dimension, the functions accepting the pointer become less and less generic and more specific to a particular array. This is where templates could be used to gain a more generic interface. However templates can provide many more capabilities and these are explained in the next section below.

Pass an array using array types

Rather than having an array decay to a pointer every time you need to use it, you can utilize a reference object. Arrays themselves are types, which means we can create pointers and references to them just like any other type. The syntax becomes a little obscure due to the nature of the array declaration, however it allows for a different style of programming to the pointer methods in the section above.

For example, we cannot simply add an & symbol to mark the array as a reference like: char &array[5] as this declares an array of references, which is illegal, and not what we want (a reference to an array). The & needs to be inside brackets to show its part of the array variable, and not the array data type: char (&array)[5].

In the following example, a function taking a reference to an array can replace the first two examples in the pointer method above:

```
void Func( char (&array)[5] ){

  for( int idx = 0 ; idx < sizeof( array ) ; ++idx ){
    Serial.println( array[ idx ], HEX );
```

```
  }
}

void setup() {
  Serial.begin( 9600 );

  char arr[] = { 1,2,3,4,5 };
  Func( arr );
}

void loop(){}
```

If you haven't already noticed, the function does not need the length to be passed in as sizeof will work correctly with a reference. The same function can be rewritten using an actual *pointer to array*, the only difference is the need to *dereference* the pointer back to an array before using it:

```
void Func( char (*array)[5] ){

  for( int idx = 0 ; idx < sizeof( *array ) ; ++idx ){
    Serial.println( (*array)[ idx ], HEX );
  }
}

//...
char arr[] = { 1,2,3,4,5 };
Func( &arr ); //Pass the 'address of' the array.
```

These examples, while useful, are still bound to an array of a certain length and are not quite as flexible as one may need. To write a function which will accept any size array, we can start to adopt some template features. The first change we can make is to let the compiler deduce the length of the array, so we can write one piece of code for any sized array (single dimension).

```
template< size_t N >
  void Func( char (&array)[N] ){

  for( int idx = 0 ; idx < N ; ++idx ){
    Serial.println( array[ idx ], HEX );
  }
}
```

The template parameter N will be replaced with a usable value when called with an array. And in the declaration we can replace the hard coded value with N. As this value is accessible as a

constant within the function, it also eliminates the need for sizeof.

The next modification we can add is to allow arrays of any data type to use the function. This requires an additional template parameter, which we can use to replace the array type char:

```
template< typename T, size_t N >
  void Func( T (&array)[N] ){
    //code here
}
```

To write a version of this function for multidimensional arrays we can simply add a new template parameter for each dimension:

```
template< typename T, size_t N, size_t X >
  void Func( T (&array)[N][X] ){
    //Code here
}
template< typename T, size_t N, size_t X, size_t Y >
  void Func( T (&array)[N][X][Y] ){
    //Code here
}
```

The benefit is each dimension is known and can be easily iterated. Heavy use of template style functions with different sized arrays could lead to code bloat, so these functions could be used as wrappers for pointer style functions, or to map multidimensional arrays onto functions accepting single dimension, which can be optimized very well by the compiler.

Unique solution ID: #1031
Author: Christopher Andrews
Last update: 2014-10-26 06:02