Array access | 9.3 Arrays | UT.6.01x Coursew... (https://www.edx.org) https://courses.edx.org/courses/UTAustinX/UT...

UTAustinX: UT.6.01x Embedded Systems - Shape the World KarenWest (/dashboard) ▼

Courseware (/courses/UTAustinX/UT.6.01x/1T2014/courseware) **Course Info (/courses/UTAustinX/UT.6.01x/1T2014/info)**

Discussion (/courses/UTAustinX/UT.6.01x/1T2014/discussion/forum) **Progress (/courses/UTAustinX/UT.6.01x/1T2014/progress)**

Questions (/courses/UTAustinX/UT.6.01x/1T2014/a3da417940af4ec49a9c02b3eae3460b/)

Syllabus (/courses/UTAustinX/UT.6.01x/1T2014/a827a8b3cc204927b6efaa49580170d1/)

We will need a place to put data. One of the simplest and fastest places to store data is in RAM memory. The TM4C123 has 32 kibibytes of RAM, and we can use it to store temporary data. If the information is a constant, and we know its values at compile time, we can place the data in ROM. The TM4C123 has 256 kibibytes of flash ROM, and we can use it to store constant data.

Random access means one can read and write any element in any order. Random access is allowed for all indexable data structures. An indexed data structure has elements of the same size and can be accessed knowing the name of the structure, the size of each element, and the element number. In C, we use the syntax `[]` to access an indexed structure. Arrays, matrices, and tables are examples of indexed structures available in C.

Sequential access means one reads and writes the elements in order. Pointers are usually employed in these types of data structures. Strings, linked-lists, stacks, queues, and trees are examples of sequential structures. The first in first out circular queue (**FIFO**) is useful for data flow problems.

Help

An **array** is made of elements of equal precision and allows random access. The **precision** is the size of each element. Typically, precision is expressed in bits or bytes. The **length** is the number of elements. The **origin** is the index of the first element. A data structure with the first element existing at index zero is called **zero-origin indexing**. In C, zero-origin index is almost always used. For example, `Data[0]` is the first element of the array `Data`.

Just like any variable, arrays must be declared before they can be accessed. The number of elements in an array is determined by its declaration. Appending a constant expression in square brackets to a name in a declaration identifies the name as the name of an array with the number of elements indicated. Multi-dimensional arrays require multiple sets of brackets. These examples illustrate valid declarations of arrays. Because there is no `const` modifier, the arrays will be defined in RAM. The C compiler will add code that runs before your main, which will initialize all RAM-based variables to zero.

```
short Data[5];      // allocate space for 5 16-bit integers
long scores[20];    // allocate space for 20 32-bit integers
int Width[6];       // 6 signed, precision depends on compiler
char Image[5][10];  // allocate space for 50 8-bit integers
short Points[5][5][5];// allocate space for 125 16-bit ints
```

If you would like the compiler to initialize to something other than zero, you can explicitly define the initial values. Again, this initialization will occur before your main is executed. Even though they have nonzero initial values, these arrays are in RAM, and thus can be modified at run time.

```
short Data[5]={1,-1,2,-2,6};
```

```
long scores[20]={-2,-1,0,1,1,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2}
int Width[6]={-10,-20,10,100,2000,40};
short Image[5][10]={
  {0,0,1,0,1,1,1,0,1,0},
  {0,0,1,0,0,1,0,0,1,0},
  {0,0,1,0,0,1,0,0,1,0},
  {0,0,1,0,0,0,0,0,1,0},
  {0,0,0,1,1,1,1,1,0,0}};
```

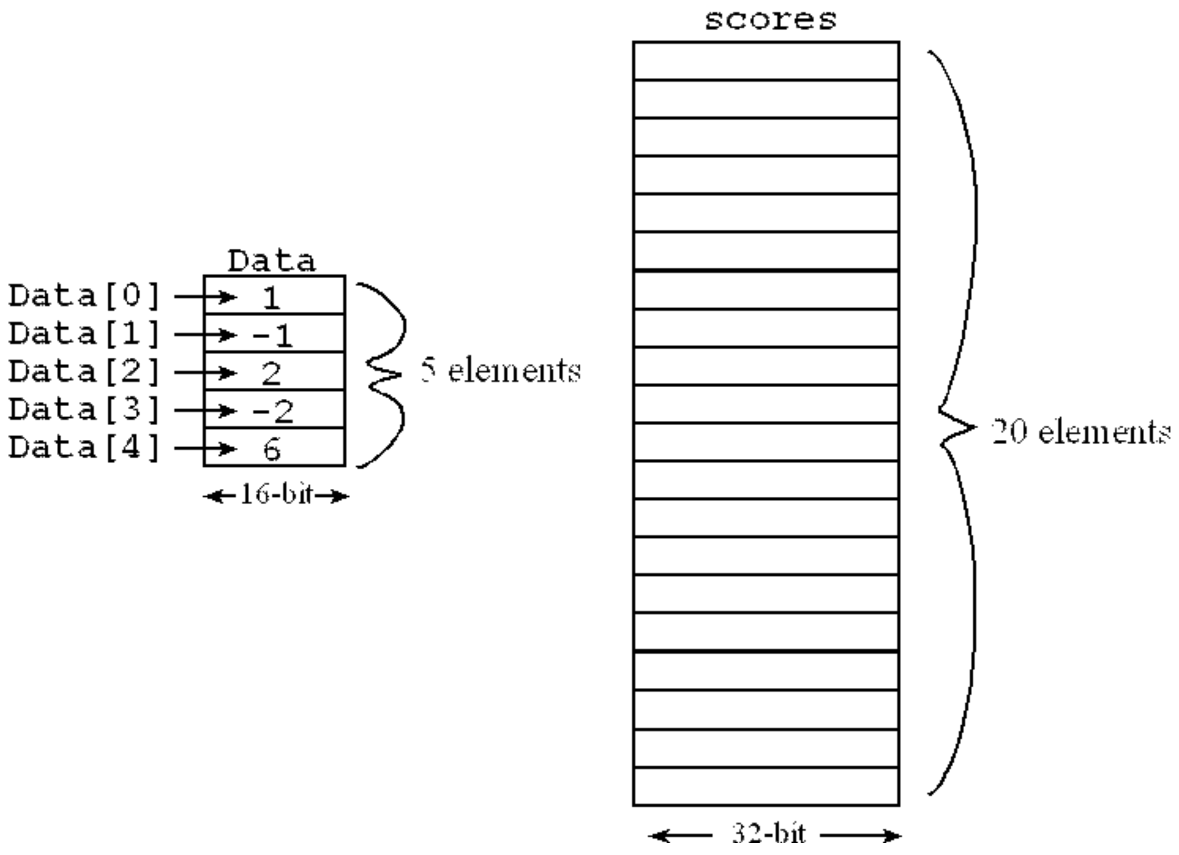
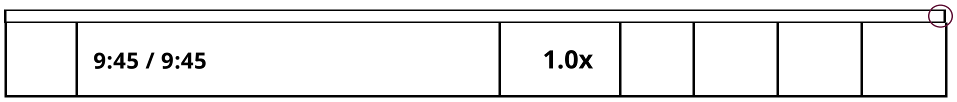


Figure 9.1. The ***`Data`*** *array has 5 16-bit elements, and the* ***`scores`*** *array has 20 32-bit elements.*

EXAMPLE OF AN ARRAY

DR. RAMESH YERRABALLI: So we looked at arrays.



Help

And we saw how we define an array, we access elements of an array.

And now, we will look at manipulating an array.

So now to look at the manipulation of an array, we will look at two examples.

So, manipulation involves two things.

One, where we sequentially access-- and that is first example

will be sequentially access an array.

And the second example will involve accessing until success.

And I'll explain more detail.

What we'll also learn in this module is how to pass an array to a subroutine.

So the first problem we're going to look at

is to find the max score in an array of student scores.

And the idea is that we will write a subroutine.

So the subroutine's going to be called Max.

And it will take as input an array.

This is an array of students in a class.

I'm going to call that class.

And second parameter, which will be the size.

So the idea is that the array when called, the subroutine when called,

will be passed an array of some length.

But since we don't know what the length is by looking at this parameter,

we pass a second parameter, which is the size of the array.

As a result of running this subroutine, it will find the maximum element

and return the maximum element back to the caller.

So the way we pass an array to a subroutine, then,

is by specifying the name of the subroutine.

But we do not specify the length of the array.

This way, we can make the subroutine very genertic

so that we could have an array-- for example,

this could be a class of students with, let's say, this is an array or 50 students indexed from zero to, let's say, 49.

So there's 50 students here.

Now I can find the maximum of all 50 students,

or I could only find the maximum of the first 20 students.

I can make this subroutine general so that I

can pass the size as that term by which I

want to limit the search for the maximum to.

So let's write the subroutine then.

So I'm going to declare a local variable.

I call it the largest, which is going to hold at any time the largest value

so far.

And I'm going to use an index i

So this is a case of using for loop.

And the idea is that any time we know we have to reach all of the elements,

so in order to find the maximum element out

of n elements, whatever this n is here in this size,

I have to touch every element to find out which one is the max.

So the natural fit is a for loop.

So I say, for.

I start from the beginning-- i is equal to 0.

I go all the way until the end, which means i is less than size.

And remember that if the size is, let's say, five, then

I have to look at element zero, one, two, three, four.

So it's always less than the size that I have to go out to.

And then I have i++.

So if you saw this in the previous video, we have the initialization,

we have the condition to stop, and the update step.

And now I'm going to look at each item.

So I say, if class of i, the score that I'm currently looking at,

the i-th score,

if this is larger-- greater than the largest, then I have to set this item to be the new largest.

So I said largest equals class of i.

Now, if this is not the new largest-- if it's not the new largest, then

I don't worry about it.

So I go off and pick the next element.

So this way, as I iterate through this, I

attach the first element-- let's say, my array that has being passed to me.

So this is class.

And let's say size was equal to 50.

So size is 50.

And I start from i.

So i is initially equal to zero.

That's what that says.

I compare against the first element.

Let's say the first score was 15, the second score was 30,

the third score was 21, and so on all the way up to the last score which is,

let's say, another 17.

So I'm going to look at each item.

And any time I find an item that's larger than the largest so far,

I will swap it.

But the only problem in this code is that the largest which

was declared here has not been initialized.

If you don't initialize a variable, you can't

assume that it's going to be equal to zero.

So I'm going to do an explicit initialization.

I say largest equal to zero.

So when I haven't touched any item yet, the largest item has a value of zero.

So once I'm out of this for loop, I have the largest in the variable largest.

So I simply return it.

So that is the end of my subroutine.

So this is an example where we see sequential access, which

means that we access the first item, the second item, the third item, and so

on all the way up to the last item.

And we also saw that we are passing an array to a subroutine.

We pass it by not specifying-- no size-- in the array that we pass.

But we pass a second parameter.

Second parameter has the size.

So this, for sequential access, the for loop is a natural fit.

So, in the next video, we will take a look

at arrays that are manipulated using this second paradigm, which is access

until success.

Help

If the array contains constants, and we know the values at compile time, we can place the data in ROM using the **`const`** modifier. For ROM-based data, we must define the value explicitly in the software. These arrays are in ROM, and thus cannot be modified at run time.

```
const short Data2[5]={1,2,3,4,5};
const long scores2[20]={-2,-1,0,1,1,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2}
const int Width2[6]={-10,-20,10,100,2000,40};
const short Image2[5][10]={
  {0,0,1,0,1,1,1,0,1,0},
  {0,0,1,0,0,1,0,0,1,0},
  {0,0,1,0,0,1,0,0,1,0},
  {0,0,1,0,0,0,0,0,1,0},
  {0,0,0,1,1,1,1,1,0,0}};
```

In a formal way, **`const`** actually means “constant”. More specifically it means the executing software cannot change the value at run time. This means in C programs not implemented on an embedded system, **`const`** data could be stored in RAM, initialized at startup. In this case, the **`const`** modifier simply means the running program cannot change the values. However, in the class we are writing C code that will be deployed into a microcontroller as part of an embedded system, so **`const`** data will be stored in ROM.

When an array element is referenced, the subscript expression designates the desired element by its position in the data. The first element occupies position zero, the second position one, and so on. It follows that the last element is subscripted by [N-1] where N is the number of elements in the array. The statement

```
scores[9] = 0;
```

for instance, sets the tenth element of the array to zero. The array subscript can be any expression that results in an integer. The following for-loop clears 20 elements of the array data to zero.

```
for(j=0;j<20;j++) {scores[j] = 0;}
```

If the array has two dimensions, then two subscripts are specified when referencing. As programmers we may any assign logical meaning to the first and second subscripts. For example we could consider the first subscript as the row and the second as the column. Then, the statement

```
TheValue = Image[3][5];
```

copies the information from the 4th row 6th column into the variable **`TheValue`**. If the array has three dimensions, then three subscripts are specified when referencing. Again we may any assign logical meaning to the various subscripts. For example we could consider the first subscript as the x coordinate, the second subscript as the y coordinate and the third subscript as the z coordinate. Then, the statement

```
Points[2][3][4]=100;
```

sets the values at position (2,3,4) to 100. Array subscripts are treated as signed integers. It is the programmer's responsibility to see that only positive values are produced, since a negative subscript would refer to some point in memory preceding the array. One must be particularly careful about assuming what exists either in front of or behind our arrays in memory.

Help