C Arrays

Kurt Schmidt
Skipjack Solutions
May 8, 2023
Intended audience: Student who has working knowledge of Python

Target compiler: I’ll try to center the discussion on C99 using gcc 7.4

Objectives:

- To gain some experience with a statically-typed language
- To gain some experience with a compiled language
- To get practice in using pointers
  - To get practice with dynamically-allocated memory
  - Play around with linked lists
Intro
Arrays

- A linear, indexed, homogeneous container
- Handy collection of related values
- A single variable (symbol), indexed
- With an index, we can access any element in constant time
• Code examples might have an accompanying link
  • Follow link to step through example at pythontutor.com
  • Does a nice job of graphically showing variables in memory, the heap, and the stack
• I’m using gcc 9 and tcc 0.9
  • We will avoid GNU extensions
Statically-Allocated Arrays
Statically-Allocated Arrays

- Size of array must be known at compile time
  - Number of elements (capacity)
  - Type (size) of element stored
- These arrays can *not* be resized
- There is a discussion about heap arrays on slide 26
  - These *can* be resized
- There is a discussion about variable-length arrays on slide 28
Declaration of Static Arrays

\[ T \text{ name}[\text{size}] \]

- Arrays are typed
- Size of \textit{statically-allocated} arrays (global or local) must be known at compile time
  - Integer literal, or
  - Integer constant
- Can be global, or declared inside a function
Example of Declaring Arrays

```c
int a[1024] ;
enum { MAX_ROWS=4096 } ;
float b[MAX_ROWS] ;
#define MAX_STUDENTS 10000
long c[MAX_STUDENTS] ;
```

- *Static* (global) arrays are zeroed out
- *Automatic* arrays (declared inside a function) are not
The `const` Qualifier

**const Isn’t All That Const**

- The `const` keyword does, sorta, make a variable immutable:

```c
const short MAX_ASSETS = 312;
MAX_ASSETS = 472;
```

- Compiling yields a warming:

```
$ gcc c2.c
c2.c:24:13: error: assignment of read-only variable ‘MAX_ASSETS’
```

- However, a `const` int is not const enough to declare a statically-allocated array
  - If local, it’ll make a Variable Length Array

- Use a literal, an `enum` constant, or a preprocessor macro
Using `enum` Constants

- Before `const` was added to C, we used this idiom
- Only for integers
- This example defines 8 bit-flags:

```c
enum { MAIN=1, MIZZEN=2, TRY=4, STAY=8, GENNY=16, STORM=32, ANCHOR=64, KEDGE=128 } ;
```

- These symbols can be used to initialise arrays
Using Preprocessor Macros – `#define`

**#define**

```c
#define symbol value
```

- symbol replaced everywhere with value
  - Compiler can’t do type checking on symbol (never sees it)
  - symbol not available to debugger
- Another common idiom for constants in C

**Preprocessor doesn’t know C**

```c
#define X 4+3
#define Y 5+7
...
int r = X * Y ;
```

In this example `r` has value 26. Why?
**size_t Type**

- Not a built-in type
  - Defined in `<stddef.h>`
    - Included by `<stdlib.h>` and others
- Created to be a type sufficient to store an index into an array
- Integer type
- Unsigned
  - Indices can’t be negative

```c
for( size_t i=0; i<size; ++i )
a[i] += 12 ;
```
Accessing Arrays

- Consider:

  ```c
  enum { CAP=20 } ;
  double arr[CAP] ;
  ```

- If \( i \) evaluates to an integer, and is a valid index into the array, then `arr[i]` is the `double` stored at that location

  ```c
  d = arr[i] - 3 ;
  ```

- If `arr` is not `const`, then `arr[i]` serves as an *L-value*
  - It can be assigned into

  ```c
  arr[i] += 12 ;
  ```
Consider, again:

<table>
<thead>
<tr>
<th>arr</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>...</th>
<th>19</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td></td>
<td>σ</td>
<td>γ</td>
<td>α</td>
<td>τ</td>
<td>δ</td>
<td>??</td>
<td>??</td>
<td>...</td>
</tr>
<tr>
<td>CAP</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>??</td>
</tr>
</tbody>
</table>

- The vector itself, the data, is of length 5
  - Valid indices (holding data) are 0-4
  - More generally, \([0, \text{size} - 1]\)

- The memory set aside for this array can store 20 elements
- It is on the programmer to remember these values
This is a valid statement:

```c
printf( "%f\n", arr[7] );
```

If array is unitialised, behavior is undefined
- Some float will be printed
- Logic error
```c
int main( void )
{
    int i ;
    int a[5] ;
    int *p = NULL ;

    for( i=0; i<5; ++i )
        a[i] = (i+2)*3 ;

    for( i=0; i<5; ++i )
        printf( "*(a+%d) = %d\n", i, *(a+i) ) ;

    for( p=a, i=0; i<5; ++i, ++p )
        *p += 2 ; /* add 2 to each element */

    return( 0 ) ;
}
```

0 https://goo.gl/5guXKm
Array Initialisers

- Arrays can be initialised at declaration:

  ```c
  double arr[4] = { 1.4142, 2.2361, 3.1415, 4.1231 } ;
  ```

- Extra initialisers are ignored

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

  yields

  ![Array Initialisers Table]

- If length specifier omitted, initialiser list is used:

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

  yields

  ![Array Initialisers Table]
Initialising Entire Array

• If not enough initialisers are provided, the last is repeated:

```c
int arr[5] = { 1, 2, 3 } ;
```

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

• So, to initialise a large array to the same value:

```c
double arr[1000] = { 0 } ;
sNode* table[1000] = { NULL } ;
```
Initialising Strings

- Strings are just arrays of characters:

  ```c
  char f[10] = { 'K', 'u', 'r', 't', '\0' } ;
  ```

  Array must have space for the null-terminator

- '\0' is the end-of-string marker (sentinel)
- Array must have space for the null-terminator

- C understands string literals:

  ```c
  char f[10] = "Kurt" ;
  ```

  Semantically the same statement
Initialising Strings – Unspecified Length

• Given an initialiser, C makes array just big enough for string plus 1 for the null-terminator

```
char f[] = "ketch" ;
```

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>'k'</td>
<td>'e'</td>
<td>'t'</td>
<td>'c'</td>
<td>'h'</td>
<td>\0</td>
</tr>
</tbody>
</table>

Pointers to String Literals

You’ll see initialisations like this:

```
char *t = "yaw1" ;
```

• t behaves much as an array
• Treat it as **readonly storage**
Passing Arrays to Functions

- Arrays are passed by reference
  - Remember, name of an array is a pointer
    - A pointer is a reference to a memory location,
    - The pointer is copied into the function
    - The function can access the caller’s array
  - Array can be modified by the function
- Size (number of elements) must also be passed to function
  - Unless a suitable sentinel value exists (e.g., ‘\0’)
- The following 2 prototypes are equivalent

```c
void foo( float a[], int n );
void foo( float* a, int n );
```
#include <stdio.h>

```c
void print( float a[], int n, FILE* ofile )
{
    for( int i=0; i<n; ++i )
        fprintf( ofile, "%f ", a[i] ); /* access by index */
}

void addPhi( float* a, int n )
{
    for( int i=0; i<n; ++i, ++a )
        *a += 1.61803398; /* access through ptr */
}
```

- First parameter same in both functions\(^1\)
- FILE* is how we pass pointers to files

\(^1\)Not quite, in C++
```c
int main( int argc, char **argv )
{
    float fs[] = { 3.1415926535, 2.718281828, 22.4, 1.414213562 } ;

    addPhi( fs, 4 ) ;

    printf( "The array now: " ) ;
    print( fs, 4, stdout ) ;
    printf( "\n" ) ;

    return( 0 ) ;
}
```

- Name of array is pointer to first element
- We’ll show you pointers soon

¹https://goo.gl/g3AoeD
Dynamically-Allocated Arrays
Arrays From the Deep Heap

- Allocated from the *heap* (*free store*)
  1. Size (and frequency) can be unknown until runtime

- You manage your own memory

- If a larger array is needed:
  1. You get a larger array
  2. Copy contents from old array
  3. Return old array to free store

- We’ll show you how to do this soon
  - After you learn some pointers
Variable-Length Arrays
Variable-Length Arrays

- These are much the same as statically-allocated arrays
- Size can be unknown until runtime
- Only for local (automatic) arrays
- Can **not** be resized

```c
int main() {
    int n;
    fputs("How many entries? => ");
    fscanf("%d", &n);
    double arr[n]; // n unknown at compile time
    ...
}
```

- Of questionable utility
- I’d avoid them
Avoiding VLAs

Compile with `-Wvla`

To avoid inadvertently creating VLAs, get the compiler to help you:

```
$ gcc -Wvla ...
```

Given

```c
const size_t CAP = 72 ;

int main( int argc, char *argv[] )
{
    double arr[CAP] ;
    ...
}
```

```
$ gcc -Wvla vla2.c
vla2.c: In function ‘main’: 
nla2.c:13:2: warning: ISO C90 forbids variable length array ‘arr’ [-Wvla]
  13 | double arr[CAP] ;
  | ^~~~~~~
```
Array Assignment
Assigning Entire Array

- C doesn’t support it
- None of the arrays can be assigned as a unit
- Initialisation is different than assignment
- Don’t do it

Assigning Arrays

None of these work:

```c
int a[] ;
int b[] ;
char l[] ;
...
b = a ;  // NO
b = { 3, 2, 1 } ;  // NO
l = "Schmidt" ;  // NO
```
Assigning Into Arrays

- Only assign into individual elements

```c
int a[LEN] = {0};
int b[LEN] = {0};
size_t n = 0;
...

// modify first n<=LEN elements of a
for( size_t i=0; i<n; ++i )
    b[i] = a[i];
```

- Preview of string operations (coming soon)

```c
char f[LEN] = "Waldo";
char l[LEN] = "Wearizee";
char n[2*LEN];

strcpy( n, f );
strcat( n, l );
```