ECE 2400 Computer Systems Programming
Fall 2021

Topic 5: C Arrays

School of Electrical and Computer Engineering
Cornell University

revision: 2021-08-28-13-32

1 Array Basics 3

2 Mapping Conceptual Storage to Machine Memory 6

3 Iterating Over Arrays 8

4 Arrays as Function Parameters 10

5 Strings 11

zyBooks The zyBooks logo is used to indicate additional material included in the course zyBook which will not be discussed in detail in lecture. Students are responsible for all material covered in lecture and in the course zyBook.

Copyright © 2021 Christopher Batten. All rights reserved. This handout was prepared by Prof. Christopher Batten at Cornell University for ECE 2400 / ENGRD 2140 Computer Systems Programming. Download and use of this handout is permitted for individual educational non-commercial purposes only. Redistribution either in part or in whole via both commercial or non-commercial means requires written permission.
In C, we would like to be able to store a sequence of values all of the same type and then perform operations on this sequence.

We already saw how to implement a sequence of values using a chain of nodes; each node is a struct with a value and a next pointer.

Arrays are an alternative approach where the sequence of values is directly mapped into a linear sequence of variables.
1. **Array Basics**

- Arrays require introducing **new types and new operators**
- Every type T has a corresponding array type
- T name[size] declares an array of size elements each of type T

```c
int    a[4];     // array of four ints
char   b[4];     // array of four chars
float  c[4];     // array of four floats
```

- size should be a constant expression (e.g., literal)
- Technically a const variable is not a constant expression
- Can initialize an array with {} initialization syntax

```c
int a[] = { 10, 11, 12, 13 };
```

- Cannot assign to an array

```c
int a[] = { 10, 11, 12, 13 };    // array of four ints
int b[4];                        // array of four ints
b = a;                            // illegal!
```
1. Array Basics

Relationship between arrays and pointers

- Assume we declare an array `int a[4]`
- Type of the expression `a` is an “array of four ints”
- Expression `a` can act like a pointer to first element in the array
- Can use **pointer arithmetic** to access elements in an array
- The following expressions evaluate to pointers to each element
  - `a` pointer to element 0
  - `a+1` pointer to element 1
  - `a+2` pointer to element 2
  - `a+3` pointer to element 3

Example declaring, initializing, accessing an array

```c
int a[] = { 10, 11, 12, 13 };
int* a_ptr0 = a;
int* a_ptr1 = a+1;
int b = *a_ptr0 + *a_ptr1;
int c = *(a+2) + *(a+3);
*a = 20;
*(a+1) = 21;
*(a+2) = 22;
*(a+3) = 23;
```

1. Array Basics

Subscript syntactic sugar

- The subscript operator \(a[i]\) is syntactic sugar for \(*\(a+i)\)
- A pointer can act like an array
- Can use subscript operator to access elements via pointer

Example declaring, initializing, accessing an array

```c
int a[] = { 10, 11, 12, 13 };

int b = a[0] + a[1];
int c = a[2] + a[3];

a[0] = 20;
a[1] = 21;
a[2] = 22;
a[3] = 23;

int* a_ptr0 = &(a[0]);
int* a_ptr1 = &(a[1]);
int d = a_ptr0[1] + a_ptr1[1];

int* a_ptr4 = &(a[4]);
int e = ( a_ptr4 == &(a[4]) );

int f = *a_ptr4;
int* a_ptr5 = &(a[5]);
```
2. Mapping Conceptual Storage to Machine Memory

- Recall that our current use of state diagrams is conceptual.
- Real machine uses memory to store variables.
- Real machine does not use “arrows”, uses memory addresses.
- Arrays are stored with index 0 at the lowest address.

![Diagram showing mapping of conceptual storage to machine memory.](image)
2. Mapping Conceptual Storage to Machine Memory

Draw both a conceptual storage and machine memory state diagram corresponding to the execution of this program:

```c
int a[] = { 10, 11 }; int b[] = { 20, 21 }; int* a_ptr = a; int* b_ptr = b; a_ptr = a_ptr + 1; int c = *a_ptr; int d = *b_ptr; int e = b[1];
```

Memory
(4B word addr)

```
<table>
<thead>
<tr>
<th>124</th>
<th>120</th>
<th>116</th>
<th>112</th>
<th>108</th>
<th>104</th>
<th>100</th>
<th>96</th>
<th>92</th>
<th>88</th>
</tr>
</thead>
</table>
```

stack

```
<table>
<thead>
<tr>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>
```

Topic 5: C Arrays
3. Iterating Over Arrays

- We primarily work with arrays by iterating over their elements
- Example of calculating average of an array of ints

```c
int a[] = { 10, 20, 30, 40 }; int sum = 0;
for ( int i = 0; i < 4; i++ )
    sum += a[i];
int avg = sum / 4;
```

- Similar code except using pointer arithmetic

```c
int a[] = { 10, 20, 30, 40 }; int sum = 0;
for ( int i = 0; i < 4; i++ )
    sum += *(a+i);
int avg = sum / 4;
```

```c
int a[] = { 10, 20, 30, 40 }; int* curr = &(a[0]);
int* end = &(a[4]);
int sum = 0;
while ( curr != end ) {
    sum += *curr;
    curr++;
}
int avg = sum / 4;
```
Draw a state diagram corresponding to the execution of this program

```c
int a[] = { 0, 13, 0, 15 }; int b[4];
int j = 0;
for ( int i=0; i<4; i++ ) {
    if ( a[i] != 0 ) {
        b[j] = a[i];
        j++;
    }
}
```

Should we use `int` or `size_t`?

- `size_t` is a typedef for a type suitable for subscripting
- `size_t` is defined in `stddef.h`
- Originally, we advocated preferring `size_t` over `int` since `size_t` cannot be negative
- However, over the past several years we have found it causes more bugs than it prevents
- Growing consensus in the C++ community that usage of `size_t` (except in very specific situations) was a mistake
4. Arrays as Function Parameters

- Arrays are *always* passed by pointer
- Must pass the size along with the actual array

```c
int avg( int* x, int n )
{
    int sum = 0;
    for ( int i=0; i<n; i++ )
        sum += x[i];
    return sum / n;
}

int main( void )
{
    int a[] = { 10, 20, 30, 40 };
    int b = avg( a, 4 );
    return 0;
}
```

- Arrays are *always* passed by pointer
- ... even with the following syntax

```c
int avg( int x[], int n )
{
    int sum = 0;
    for ( int i=0; i<n; i++ )
        sum += x[i];
    return sum / n;
}
```

- Prefer using `int* x` for parameters
- It makes it obvious arrays are *always* passed by pointer
5. Strings

- Strings are just arrays of chars
- The length of a string is indicated in a special way
- The null terminator character (\0) indicates the end of string
- New syntax using double quotes for string literals (""")

```c
char a[] = { 'e', 'c', 'e', '\0' };
char b[] = "2400";
char c[8];
c[0] = 'f';
c[1] = 'o';
c[2] = 'o';
c[3] = '\0';
```

- C standard library provides many string manipulation functions
- These functions are declared in the string.h header
  - strlen: calculate length of a string
  - strcmp: compare two strings
  - strcpy: copy one string to another string
  - atoi: convert a string into an integer
Draw a state diagram corresponding to the execution of this program

```c
int strlen( char* str )
{
    int i = 0;
    while ( str[i] != '\0' )
        i++;
    return i;
}

int main( void )
{
    char a[] = "ece2400";
    int b = strlen( a );
    return 0;
}
```