# 2.21 Array pointers

Pointer and arrays (2 sides of the same coin)

Pointer arithmetic

*EE108 – Computing for Engineers*

1

## Overview



Aims

- Learn how to use pointers to arrays as function input/output parameters and for iterating over an array without an explicit counter

Learning outcomes – you should be able to…

- Declare and initialize a pointer to an array (using either the address of element zero or the array name)
- Use basic pointer arithmetic to point at arbitrary elements in an array
- Iterate over an array using either pointer arithmetic or subscripting with an array pointer



2

## Pointers and array elements

3

Pointers can point to array elements in addition to ordinary variables

In C, an **array pointer** is nothing more than a pointer to the first element (index zero)

The array variable name (without square brackets) is actually just a pointer to the array (i.e. a pointer to the first element)

An array pointer is identical to an array element pointer which is identical to an ordinary variable pointer. You can do the same things with all of them.



3

## Pointers and array elements

4

Consider the following code…

```
int arr[] = { 100, 101, 102, 103 }; // an array of 4 elements
int *p1;                            // a pointer to an int
int *p2;                            // a pointer to an int
int *p3;                            // a pointer to an int
int tmp;

// array variable name without [] is a pointer to start of array
p1 = arr;     // p1 points at the array, arr, which also means it
              // points at element 0 of arr

// point at individual array elements by getting their address
p2 = &arr[0]; // p2 points to element 0 of arr
p3 = &arr[2]; // p3 points to element 2 of arr

tmp = *p1;  // tmp is assigned value of p1's pointee (arr[0]), i.e. 100
tmp = *p3;  // tmp is assigned value of p3's pointee (arr[2]), i.e. 102

*p2 = 1000; // p2's pointee (arr[0]) is set to 1000, i.e. arr[0] is now 1000
```

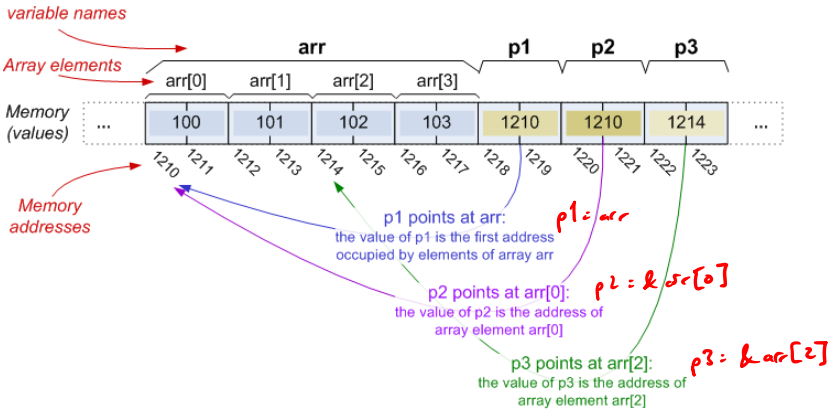


4

# Pointers and array elements



Suppose arr is an array of 4 int values (each 2 bytes wide). Then suppose p1, p2, and p3 are all pointers (of type pointer to int). Finally suppose that p1 points at the array, p2 points at element 0 of the array, and p3 points at element 2 of the array.








# Array element pointer vs. array pointer



There is no difference in the way that you declare or use each of the following:

- an array pointer (which is a pointer to element zero of the array)
- a pointer to an arbitrary array element

…we make use of this fact when iterating over the array using a pointer

You can change the value of an array/element pointer variable so that it points at a different element

However, you cannot change the value of the array name [pointer] to point at anything else

```
int arr[] = { 100, 101, 102, 103 };
int *p1;

// changing a pointer to point at one element and then another is OK
p1 = &arr[0]; // p1 points to element 0
p1 = &arr[1]; // p1 now points to element 1

// You cannot change an array name to point at some other location
arr = p1; // ERROR: won't compile – cannot change what arr points to
```

# Pointer arithmetic

7



7

## Pointer arithmetic

8

**Pointer arithmetic** is used to change what a pointer points to by specifying the change relative to where it currently points

Except for some advanced uses, pointer arithmetic is **used almost exclusively with arrays**

It turns out that pointer arithmetic and array subscripting have essentially identical meanings



8

## Pointer arithmetic – adding/subtracting integers



**Adding** an integer to an array element pointer, p, advances the pointer that number of places after p

**Subtracting** an integer from an array element pointer, p, rewinds the pointer that number of places before p

Also, array subscript notation is equivalent to using pointer arithmetic to advance a pointer and then dereferencing it, i.e. `arr[i]` is equivalent to `*(arr + i)`

```
int arr[] = { 100, 101, 102, 103 };
int *p1;
int *p2;

p1 = &arr[2]; // p1 points to element 2
p2 = arr + 2; // p2 also points at element 2 (2 places after arr which
              // points at start of array which is element 0)

p2 = p1 + 1; // p2 now points to element 3 (1 place after p1)
p2 = p1 - 1; // p2 now points to element 1 (1 place before p1)

arr[0] = 50;     // element 0 of arr is now 50
*(arr + 1) = 51; // element 1 of arr is now 51
```

***Note:** when adding/subtracting an integer and a pointer, **the pointer address changes by the integer times the sizeof the pointee** (as defined in the pointer declaration)*





## Pointer arithmetic – increment/decrement



We can also use the ++ and -- operators to move a pointer by exactly one place

- This is frequently used when iterating over an array

```
int arr[] = { 100, 101, 102, 103 };
int *p1 = arr; // p1 points to (element 0 of) arr

p1++; // equivalent to p1 = p1 + 1; so p1 now points to element 1
p1--; // equivalent to p1 = p1 - 1; so p1 now points to element 0 again
```

A commonly used idiom (when iterating over arrays) is to combine the dereference operator, *, with ++ or --

```
int arr[] = { 100, 101, 102, 103 };
int *p1 = arr; // p1 points to (element 0 of) arr
int tmp;

tmp = *p1++; // equivalent to tmp = *(p1++)
             // which becomes tmp = *p1; p1 = p1 + 1;
             // so tmp is 100 (value of element 0),
             // and p1 ends up pointing to element 1
```

## Adding/subtracting two pointers



Just now we've looked at adding/subtracting an integer to/from a pointer. What happens if we add/subtract two pointers?

**Subtracting one pointer from another** yields the number of places/elements between the two pointers

*The two pointers must refer to elements of a single array for this to be meaningful*

```
int arr[] = { 100, 101, 102, 103 };
int *p1 = &arr[1]; // p1 points to element 1
int *p2 = &arr[3]; // p2 points to element 3
int result;

result = p2 - p1; // result is 2 => p2 is 2 places after p1
result = p1 - p2; // result is -2 => p1 is 2 places before p2
```

*Adding one pointer to another does not yield a meaningful result and for that reason trying to do this will result in a compile time error.*





## Pointer comparisons



It is possible and frequently useful to compare pointers to each other

`p1 == p2` is true if both pointers point to the same variable or array element

`p1 > p2` is true if p1 points to an element in the array which comes after that pointed to by p2

`p1 < p2` is true if p1 points to an element in the array which comes before that pointed to by p2

```
int arr[] = { 100, 100, 100, 100 };

int *p1 = &arr[1]; // p1 points to element 1
int *p2 = &arr[3]; // p2 points to element 3
boolean result;


result = (p1 == p2); // result is FALSE – p1 doesn't point same place as p2
result = (p1 < p2); // result is TRUE – p1 points before p2 in array
result = (p1 > p2); // result is FALSE – p1 does not point after p2 in array

if (p1 > p2) // if p1 points to an element after where p2 points
  …
```

## Functions that operate on arrays/array pointers

13





## Functions and arrays (choices)

14

To pass an array as a parameter to a function we can use the array[] notation (which we saw in section 2.10 of the notes), but the standard C library and the author of the C language prefer to use array pointers instead.

For this reason you must be able to read and understand both approaches, even if you only choose to use one of them.

As mentioned previously, when passing an array to a function (whether using an array or array pointer) we usually need to pass array length as a separate parameter to functions which handle arrays, unless the array length is somehow known to the function (e.g. it is a global constant) or can be determined in some other way (e.g. using a special end of array marker like C strings which are essentially just char arrays)

```
// parameter using array notation
void zeroArrayAS(int a[], int len) {
  …
}

// parameter using pointer notation
void zeroArrayPP(int *a, int len) {
  …
}
```

```
void loop() {
  int arr[] = { 100, 101, 102, 103 };

  // Note: both functions called
  // identically despite the difference
  // in parameters
  zeroArrayAS(arr, 4);
  zeroArrayPP(arr, 4);
}
```

# Functions and arrays (choices) contd.



Internally within the function

- Pointer arithmetic and array subscripting are seen as equivalent by the compiler, i.e. `a[i]` (element i of a) is equivalent to `*(a+i)` (what's pointed at by i places after element 0 of a)
- We can use either pointer arithmetic or array subscripting regardless of the parameter notation (but most commonly we would use subscripting with array notation and pointer arithmetic with pointer notation)

Convenient

```
// array notation, subscripting
void zeroArrayAS(int a[], int len) {
  int i;
  for (i=0; i < len; i++)
    a[i] = 0;
}
```

Sometimes

```
// array notation, pointer arithmetic
void zeroArrayAP(int a[], int len) {
  int *p;
  for (p=a; p < a+len; p++)
    *p = 0;
}
```

```
// pointer notation, subscripting
void zeroArrayPS(int *pA, int len) {
  int i;
  for (i=0; i < len; i++)
    pA[i] = 0;
}
```

Sometimes

```
// pointer notation and arithmetic
void zeroArrayPP(int *pA, int len) {
  int *p;
  for (p=pA; p < pA+len; p++)
    *p = 0;
}
```

Most common





# Outputting vs. Returning an array from a function



Returning an array from a function – not recommended!

- It's tricky and not used much except for strings (see later)
- To make it work you must return an array pointer and the caller must somehow know or be able to determine the array length

It is more common to use an array as an output or in-out parameter

- The function simply modifies the contents of the array as necessary
- Because it is (effectively) using a pointer, any changes are made directly the caller's array elements

```
// example of a function which modifies the array it receives, i.e. a is
// an output parameter
void zeroArray(int *a, int len) {
  int *p;

  for (p=a; p < a+len; p++)
    *p = 0;
}

void loop() {
  int arr[] = { 100, 101, 102, 103 };

  zeroArray(arr, 4); // value of arr is now { 0,0,0,0 }
}
```

## Array slices

17

An **array slice** is a contiguous part of an array

- e.g. from element 2 to element 4 inclusive is a slice comprising 3 elements (at indexes 2, 3, and 4 of the original array)

Any function that can take an array can also take an array slice

```
// example of a function which modifies the array it receives, i.e. a is
// an output parameter
void zeroArray(int *a, int len) {
  int *p;

  for (p=a; p < a+len; p++)
    *p = 0;
}

void loop() {
  int arr[] = { 100, 101, 102, 103, 104, 105 };

  // pass in slice of arr consisting of elements 2, 3 and 4
  zeroArray(arr+2, 3); // value of arr is now { 100,101,0,0,0,105 }
}
```



## Worked example

19

Create a program to do the following:

- **Use pointer arithmetic rather than array subscripting in all helper functions**
- Initialize two arrays, array1 and array2, with 8 elements each and set the elements to random values between 0 and 10 that you choose at design/coding time
- Define a function called addToElements that can add a scalar to all elements of an array slice. Use this function to add 2 to the first 4 elements of the array1 and to subtract 2 from the last 3 elements of array2.
- Define a function called multiplyElements and use it to calculate the element-wise product of array1 and array2, storing the result in array1 (i.e. effectively array1Elements = array1Elements × array2Elements)

## contd



```
// worked example solution top level functionality…
#define ARRAY_SIZE 8

void loop() {
  int array1[ARRAY_SIZE] = {1,2,3,4,5,6,7,8};
  int array2[ARRAY_SIZE] = {0,2,7,5,4,3,9,1};

  // add 2 to the first 4 elements of array1
  addToElements(array1, 4, 2);
  // sub 2 from the last 3 elements of array2
  addToElements(array2 + (ARRAY_SIZE-3), 3, -2);

// calculate the element-wise product
  multiplyElements(array1, array2, ARRAY_SIZE);
}
```





## contd



```
// add value to each of the len elements of the array slice
// starting at pSlice

void addToElements(int *pSlice, int len, int value) {
  int *p;

  for (p= pSlice; p < pSlice + len; p++)
    *p += value;
}
```

```
// element-wise product
void multiplyElements(int *pSlice1, int *pSlice2, int len) {
  int *p1;
  int *p2;

  // Note: there is no update expression in the following for loop
  // as p1 and p2 are already incremented inside the loop body
  for (p1=pSlice1, p2=pSlice2; p1 < pSlice1 + len; /* no updateExpr */ )
    *p1++ *= *p2++;

  // you could also write it the following long hand way to be more explicit:
  // for (p1=pSlice1, p2=pSlice2; p1 < pSlice1 + len; p1++, p2++)
  //   *p1 = (*p1) * (*p2);
}
```