Chapter 12

Pointers and Arrays

Introduction

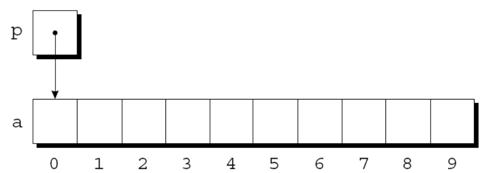
- C allows us to perform arithmetic—*addition* and *subtraction*—on pointers to *array* ***elements***
- This leads to an alternative way of processing arrays in which *pointers* take the place of *array subscripts*
- The relationship between pointers and arrays in C is a close one
- Understanding this relationship is critical for mastering C

Pointer Arithmetic

- Chapter 11 showed that pointers can point to *array* ***elements***:

```
int a[10], *p;
p = &a[0];
```

- A graphical representation:

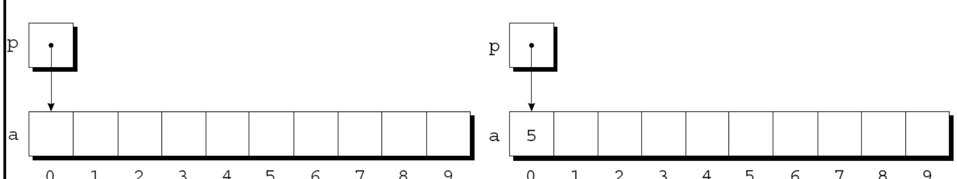


Pointer Arithmetic

- We can now access `a[0]` through `p`; for example, we can store the value `5` in `a[0]` by writing

```
*p = 5;
```

- An updated picture:



Before ***After***

Pointer Arithmetic

- If `p` points to an ***element*** *of an array* `a`, the other elements of `a` can be accessed by performing ***pointer arithmetic*** (or ***address arithmetic***) on `p`
- `C` supports three (***and only three***) forms of pointer arithmetic:
  - Adding an integer to a pointer
  - Subtracting an integer from a pointer
  - Subtracting one pointer from another

Adding an Integer to a Pointer

- Adding an integer `j` to a pointer `p` yields a pointer to the *element* `j` places after the one that `p` points to
- More precisely, if `p` points to the *array* ***element*** `a[i]`, then `p + j` points to `a[i+j]`
- Assume that the following declarations are in effect:

```
int a[10], *p, *q, i;
```

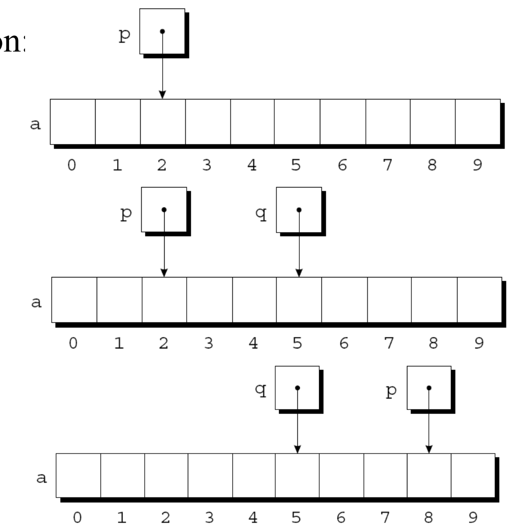
Adding an Integer to a Pointer

- Example of pointer addition:

```
p = &a[2];
```

```
q = p + 3;
```

```
p += 6;
```



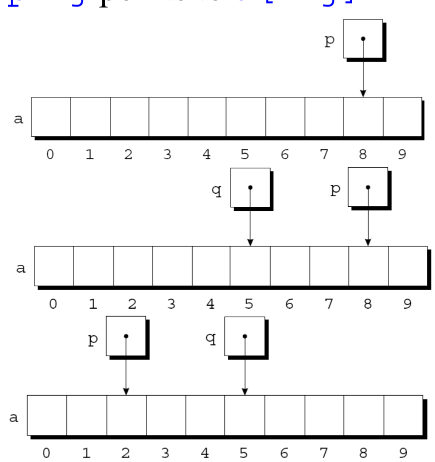
Subtracting an Integer from a Pointer

- If `p` points to `a[i]`, then `p - j` points to `a[i-j]`
- Example:

```
p = &a[8];
```

```
q = p - 3;
```

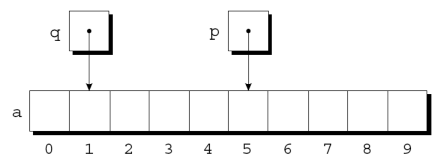
```
p -= 6;
```



Subtracting One Pointer from Another

- When one pointer is subtracted from another, the result is the *distance* (measured in array elements) between the pointers
- If `p` points to `a[i]` and `q` points to `a[j]`, then `p - q` is equal to `i - j`
- Example:

```
p = &a[5];
q = &a[1];
```




```
i = p - q;   /* i is 4 */
i = q - p;   /* i is -4 */
```


Subtracting One Pointer from Another

- Operations that cause undefined behavior:
  - Performing arithmetic on a pointer that does not point to an *array* ***element***
  - Subtracting pointers that do not point to elements of the same array

Comparing Pointers

- Pointers can be compared using the relational operators (`<`, `<=`, `>`, `>=`) and the equality operators (`==` and `!=`)
  - Using relational operators is meaningful *only* for pointers to ***elements*** of the same *array*
- The outcome of the comparison depends on the relative positions of the two ***elements*** in the *array*
- After the assignments

```
p = &a[5];
q = &a[1];
```

the value of `p >= q` is `true` and `p <= q` is `false`

Using Pointers for Array Processing

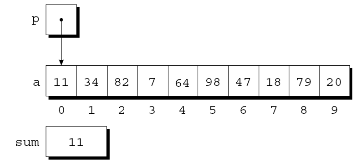
- Pointer arithmetic allows us to visit the ***elements** of an array* by repeatedly incrementing a pointer variable
- A loop that sums the ***elements** of an array* `a`:

```
#define N 10
…
int a[N], sum, *p;
…
sum = 0;
for (p = &a[0]; p < &a[N]; p++)
  sum += *p;
```

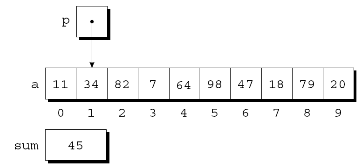
Using Pointers for Array Processing

```
for (p = &a[0]; p < &a[N]; p++)
  sum += *p;
```

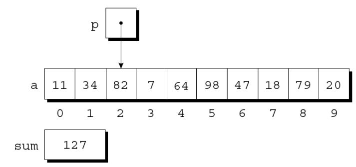
At the end of the first iteration:



At the end of the second iteration:



At the end of the third iteration:



Using Pointers for Array Processing

- The condition `p < &a[N]` in the `for` statement deserves special mention
- It is *legal* to apply the address operator to `a[N]`, even though this *element does not exist*

- Pointer arithmetic may save execution time
- However, some **C** compilers produce better code for loops that rely on subscripting

Combining the `*` and `++` Operators

- `C` programmers often combine the `*` (indirection) and `++` operators
- A statement that modifies an array element and then advances to the next element:
  `a[i++] = j;`
- The corresponding pointer version, where `p = &a[i]`:
  `*p++ = j;`
- Because the postfix version of `++` takes precedence over `*`, the compiler sees this as
  `*(p++) = j;`
- *How about* `(*p)++ = j;`*?*

Combining the `*` and `++` Operators

- Possible combinations of `*` and `++`, where `p = &a[i]`:

Expression	*Meaning*
`*p++` or `*(p++)`	Value of expression is `*p` before increment; increment `p` later, i.e., `a[i++]`
`(*p)++`	Value of expression is `*p` before increment; increment `*p` later, i.e., `a[i]++`
`*++p` or `*(++p)`	Increment `p` first; value of expression is `*p` after increment, i.e., `a[++i]`
`++*p` or `++(*p)`	Increment `*p` first; value of expression is `*p` after increment, i.e., `++a[i]`

Combining the * and ++ Operators

- The most common combination of * and ++ is *p++ (i.e., a[i++]), which is handy in loops
- Instead of writing

```
for (p = &a[0]; p < &a[N]; p++)
  sum += *p;
```

to sum the elements of the array a, we could write

```
p = &a[0];
while (p < &a[N])
  sum += *p++;
```


Combining the * and ++ Operators

- The * and -- operators mix in the same way as * and ++
- For an application that combines * and --, let us return to the stack example of Chapter 10
- The original version of the stack relied on an integer variable named top to keep track of the "top-of-stack" position in the contents array
- Let us replace top by a pointer variable that points initially to ***element*** 0 of the contents array:

```
int *top_ptr = &contents[0];
```

Combining the * and ++ Operators

- The new push and pop functions:

```
void push(int i)
{
  if (is_full())
    stack_overflow();
  else
    *top_ptr++ = i;
}
```

Instead of
`contents[top++] = i;`

```
int pop(void)
{
  if (is_empty())
    stack_underflow();
  else
    return *--top_ptr;
}
```

Instead of
`return contents[--top];`

Using an Array Name as a Pointer

- Pointer arithmetic is one way in which arrays and pointers are related
- Another key relationship:

  The name of an array can be used as a pointer to the ***first element*** *in the array*

Using an Array Name as a Pointer

- Suppose that `a` is declared as follows:

```
int a[10];
```

- Examples of using `a` as a pointer:

```
*a = 7;    /* stores 7 in a[0] */
*(a+1) = 12;    /* stores 12 in a[1] */
```

- In general, `a + i` is the same as `&a[i]`
  - Both represent ***a pointer*** to ***element*** `i` in array `a`
- Also, `*(a+i)` is equivalent to `a[i]`
  - Both represent element `i` itself

Using an Array Name as a Pointer

- The fact that an array name can serve as a pointer makes it easier to write loops that step through an array
- Original loop:

```
for (p = &a[0]; p < &a[N]; p++)
  sum += *p;
```

- Simplified version:

```
for (p = a; p < a + N; p++)
  sum += *p;
```

Using an Array Name as a Pointer

- Although an array name can be used as a pointer, it is *not possible* to assign it a new value
- Attempting to make it point elsewhere is an error:

```
while (*a != 0)
  a++;            /*** WRONG ***/
```

- This is no great loss; we can always copy `a` into a pointer variable, then change the pointer variable:

```
p = a;
while (*p != 0)
  p++;
```


Program: Reversing a Series of Numbers (Revisited)

- The **`reverse.c`** program of Chapter 8 reads 10 numbers, then writes the numbers in reverse order
- The original program stores the numbers in an array, with subscripting used to access elements of the array
- **`reverse3.c`** is a new version of the program in which subscripting has been replaced with pointer arithmetic

reverse3.c

```
/* Reverses a series of numbers (pointer version) */

#include <stdio.h>

#define N 10

int main(void)
{
  int a[N], *p;

  printf("Enter %d numbers: ", N);
  for (p = a; p < a + N; p++)
    scanf("%d", p);

  printf("In reverse order:");
  for (p = a + N - 1; p >= a; p--)
    printf(" %d", *p);
  printf("\n");

  return 0;
}
```

Was: `for (i = 0; i < N; i++)`
`scanf("%d", &a[i]);`

Was: `for (i = N - 1; i >= 0; i--)`
`printf(" %d", a[i]);`

Array Arguments (Revisited)

- When passed to a function, an array name is treated as a pointer
- Example:

```
int find_largest(int a[], int n)
{
  int i, max;

  max = a[0];
  for (i = 1; i < n; i++)
    if (a[i] > max)
      max = a[i];
  return max;
}
```

- A call of `find_largest`:

```
largest = find_largest(b, N);
```

This call causes a pointer to the ***first element*** of `b` to be assigned to `a`; the array itself is not copied

Array Arguments (Revisited)

- The fact that an array argument is treated as a pointer has some important consequences
- ***Consequence 1***:
  When an ordinary variable is passed to a function,
  - its value is copied
  - any changes to the corresponding parameter do not affect the variable
- In contrast, when an array used as an argument:
  - its address is copied
  - Array elements are not protected against change

Array Arguments (Revisited)

- For example, the following function modifies an array by storing zero into each of its elements:

```
void store_zeros(int a[], int n)
{
  int i;

  for (i = 0; i < n; i++)
    a[i] = 0;
}
```

Array Arguments (Revisited)

- To indicate that the elements of an array parameter will not be changed, we can include the word `const` in its declaration:

```
int find_largest(const int a[], int n)
{
  …
}
```

- If `const` is present, the compiler will make sure that no assignment to an element of `a` appears in the body of `find_largest`.

Array Arguments (Revisited)

- ***Consequence 2***:
  The time required to pass an array to a function does not depend on the size of the array
- There is no penalty for passing a large array, since no copy of the array is made

Array Arguments (Revisited)

- ***Consequence 3***:
  An array parameter can be declared as a pointer, if desired
- `find_largest` could be defined as follows:

```
int find_largest(int *a, int n)
{
  …
}
```

- Declaring `a` to be a pointer is equivalent to declaring it to be an array; the compiler treats the declarations as though they were identical

Array Arguments (Revisited)

- Although declaring a *parameter* to be an array is the same as declaring it to be a pointer, the same is not true for a *variable*
  - The following declaration causes the compiler to set aside space for 10 integers:

```
int a[10];
```

  - The following declaration causes the compiler to allocate space for a pointer variable:

```
int *a;
```

Array Arguments (Revisited)

- In the latter case, `a` is not an array;
  attempting to use it without initialization can have disastrous results
- For example, the assignment

```
*a = 0;    /*** WRONG ***/
```

  will store `0` where `a` is pointing
- Since we do not know where `a` is pointing, the effect on the program is undefined

Array Arguments (Revisited)

- ***Consequence 4***:
  A function with an array parameter can be passed an array "slice"—a sequence of consecutive elements
- An example that applies `find_largest` to elements 5 through 14 of an array `b`:

```
largest = find_largest(&b[5], 10);
```

Using a Pointer as an Array Name

- C allows us to subscript a pointer as though it were an array name:

```
#define N 10
…
int a[N], i, sum = 0, *p = a;
…
for (i = 0; i < 5; i++)
  sum += p[i];
```

What if we replaced `*p=a` by `*p=a+5`?

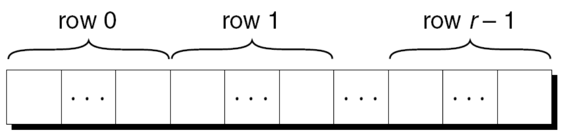
The compiler treats `p[i]` as `a[i]` and as `*(p+i)`

Pointers and Multidimensional Arrays

- Just as pointers can point to ***elements*** of one-dimensional arrays, they can also point to ***elements*** of multidimensional arrays
- This section explores common techniques for using pointers to process the elements of multidimensional arrays

Processing the Elements of a Multidimensional Array

- Chapter 8 showed that C stores two-dimensional arrays in row-wise order
- Layout of an array with *r* rows:



- If `p` initially points to the ***element*** in row 0, column 0, we can visit every element in the array by incrementing `p` repeatedly

Processing the Elements of a Multidimensional Array

- Consider the problem of initializing all elements of the following array to zero:

```
int a[NUM_ROWS][NUM_COLS];
```

- The obvious technique would be to use nested `for` loops:

```
int row, col;
…
for (row = 0; row < NUM_ROWS; row++)
  for (col = 0; col < NUM_COLS; col++)
    a[row][col] = 0;
```

- If we view `a` as a one-dimensional array of integers, a single loop is sufficient:

```
int *p;
…
for (p = &a[0][0]; p <= &a[NUM_ROWS-1][NUM_COLS-1]; p++)
  *p = 0;
```

Processing the Elements of a Multidimensional Array

- Although treating a two-dimensional array as one-dimensional may seem like *cheating*, it works with most **C** compilers

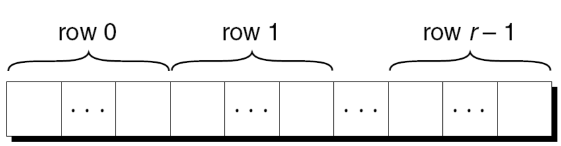
Processing the Rows of a Multidimensional Array

- A pointer variable `p` can also be used for processing the ***elements*** in just one *row* of a two-dimensional array
- To visit the ***elements*** of row `i`, we would initialize `p` to point to ***element*** 0 in row `i` in the array `a`:

  ```
  p = &a[i][0];
  ```

  As `&b[0]` means just `b` or we could simply write

  ```
  p = a[i];
  ```



Processing the Rows of a Multidimensional Array

- For any two-dimensional array `a`, the expression `a[i]` is a pointer to the first ***element*** in row `i`
- To see why this works, recall that `h[k]` is equivalent to `*(h + k)`
- Thus, `&a[i][0]` is the same as `&(*(a[i] + 0))`, which is equivalent to `&*a[i]`
- This is the same as `a[i]`, since the `&` and `*` operators cancel each other

Processing the Rows of a Multidimensional Array

- A loop that clears row `i` of the array `a`:

```
int a[NUM_ROWS][NUM_COLS], *p, i;
…
for (p = a[i]; p < a[i] + NUM_COLS; p++)
  *p = 0;
```

- Since `a[i]` is ***a pointer to row*** `i` of the array `a`, we can pass `a[i]` to a function that is expecting a one-dimensional array as its argument
- In other words, a function that is designed to work with one-dimensional arrays will also work with a row belonging to a two-dimensional array

Processing the Rows of a Multidimensional Array

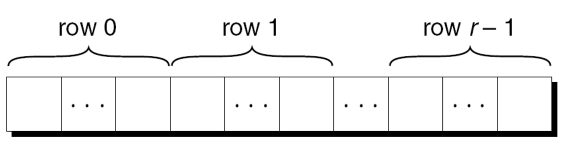
- Consider `find_largest`, which was originally designed to find the largest element of a one-dimensional array
- We can just as easily use `find_largest` to determine the largest element in row `i` of the two-dimensional array `a`:

```
largest = find_largest(a[i], NUM_COLS);
```


Processing the Columns of a Multidimensional Array

- Processing the elements in a *column* of a two-dimensional array is not as easy, because arrays are stored by row, not by column
- A loop that clears column `i` of the array `a`:

```
int a[NUM_ROWS][NUM_COLS], (*p)[NUM_COLS], i;
…
for (p = &a[0]; p < &a[NUM_ROWS]; p++)
  (*p)[i] = 0;
```



Chapter 12: Pointers and Arrays

Declaration Syntax

- Declarators include:
  - Identifiers (names of simple variables)
    For example ➔ `int x;`
  - Identifiers followed by `[]` (array names)
    For example ➔ `int x[10];`
  - Identifiers preceded by `*` (pointer names)
    For example ➔ `int *x;`
  - Identifiers followed by `()` (function names)
    We did not cover this yet
- Declarators in actual programs often combine these three notations together ,i.e., `*`, `[]`, and `()`

C PROGRAMMING A Modern Approach SECOND EDITION

45

Chapter 12: Pointers and Arrays

Deciphering Complex Declarations

- Rules for understanding declarations:
  - ***Always read declarators from the inside out***
    - locate the identifier that's being declared, and
    - start deciphering the declaration from there
  - ***When there's a choice***
    - always favor [] over *
  - ***Parentheses can be used to override the normal priority of `[]` over `*`***
- Examples:
  `int *b[10];`
  `b` is a ***10-element array*** of *pointers to integer*
  `int (*a)[10];`
  `a` is a single ***pointer*** to *an array* of 10 integers

C PROGRAMMING A Modern Approach SECOND EDITION

46

Deciphering Complex Declarations

- Examples:

  `int *b[10];`

  `b` is a ***10-element array*** of *pointers to integer*

  - Assuming that each pointer is 4 bytes
    - This object is 40 bytes (10 elements)
    - Each element is pointing to an integer, i.e., 4 bytes

  `int (*a)[10];`

  `a` is a single ***pointer*** to *an array* of 10 integers

  - Assuming that each pointer is 4 bytes
    - This object is 4 bytes (a single element)
    - This element is pointing to 10 integers, i.e., 40 bytes

  `int a[10][5];`

  `a` is a ***10-element array*** of *5-element array of integers*

Using the Name of a Multidimensional Array as a Pointer

- The name of *any* array can be used as a pointer, regardless of how many dimensions it has, but some care is required
- Example:

  `int a[NUM_ROWS][NUM_COLS];`

  `a` is *not* a pointer to `a[0][0]`; instead, it is a pointer to `a[0]`
- **C** regards `a` as a ***one-dimensional array whose elements are one-dimensional arrays***
- When used as a pointer, `a` has type `int (*)[NUM_COLS]` (pointer to an integer array of length `NUM_COLS`)

`a` is a ***NUM_ROWS-element array*** of ***NUM_COLS****-element array of integers*

Using the Name of a Multidimensional Array as a Pointer

- Knowing that `a` points to `a[0]` is useful for simplifying loops that process the elements of a two-dimensional array
- Instead of writing

```
for (p = &a[0]; p < &a[NUM_ROWS]; p++)
  (*p)[i] = 0;
```

to clear column `i` of the array `a`, we can write

```
for (p = a; p < a + NUM_ROWS; p++)
  (*p)[i] = 0;
```


Using the Name of a Multidimensional Array as a Pointer

- We can “trick” a function into thinking that a multidimensional array is really one-dimensional
- A first attempt using `find_largest` to find the largest element in `a`:

```
largest = find_largest(a, NUM_ROWS * NUM_COLS); /*WRONG*/
```

This is an error, because the type of `a` is `int (*)[NUM_COLS]` but `find_largest` is expecting an argument of type `int *`

- The correct call:

```
largest = find_largest(a[0], NUM_ROWS * NUM_COLS);
```

`a[0]` points to element 0 in row 0, and it has type `int *`