Pointers and Arrays

Marcus Birkenkrahe

October 26, 2024

README

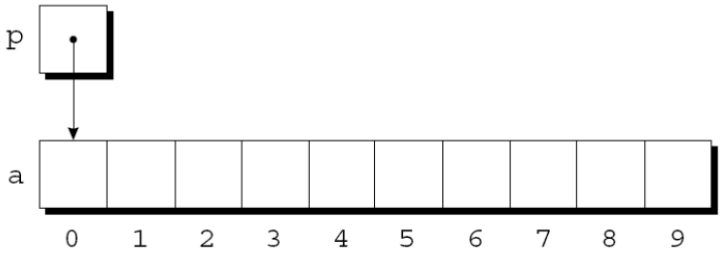
- Pointers and arrays are closely related in the C programming language. Understanding pointer arithmetic and how arrays are accessed through pointers is fundamental for low-level memory management and efficient program execution.
- In class, we'll be working with a wiki-type summary and interactive exercises. This section is accompanied by one bonus and by one mandatory programming assignment, as well as an online test.
- Much of this is based on King, C Programming (2008), chapter 12.

  "Premature optimization is the root of all evil." –Donald E Knuth

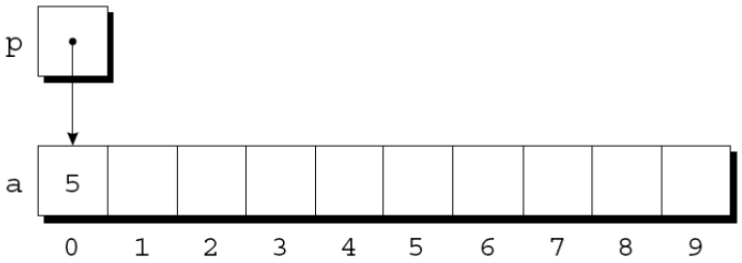
Pointer Arithmetic

- Pointers can point to array elements:

```
int a[10] = {0,1,2,3,4,5,6,7,8,9};
int *p;

p = &a[0]; // p points to a[0]
printf("p points at a[%d]\n", *p);


p points at a[0]
```



- You can now store a value in `a[0]` by writing `*p = 5`:



- You can use the pointer to access other array elements: this is called *pointer* or *address arithmetic* on `p` by

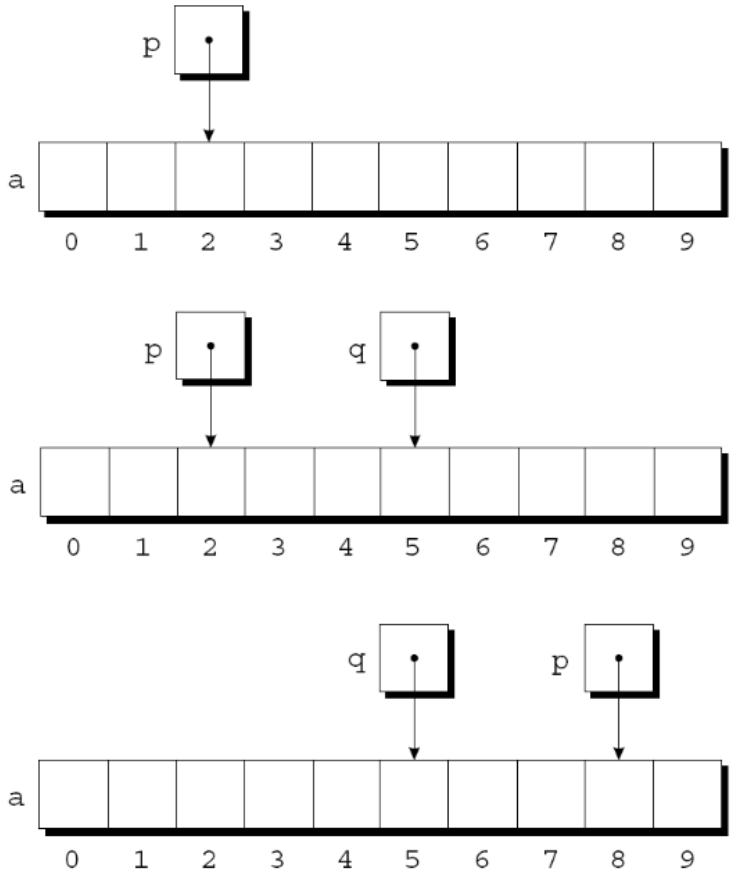
  1. Adding an integer to a pointer
  2. Subtracting an integer from a pointer
  3. Subtracting one pointer from another

  Pointers cannot be multiplied with or divided by one 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 element `p` points to: if `p` points to `a[i]`, then `p+j` points to `a[i+j]`.
- Code example (illustrated below)

```
int a[10] = {0,1,2,3,4,5,6,7,8,9};
int *p, *q, i;

p = &a[2]; // p points at a[2]
printf("p points at a[%d]\n", *p);

q = p + 3; // q points at a[2+3] = a[5]
printf("q points at a[%d]\n", *q);

p += 6;  // p points at a[2+6] = a[8]
printf("p points at a[%d]\n", *p);


p points at a[2]
q points at a[5]
p points at a[8]
```



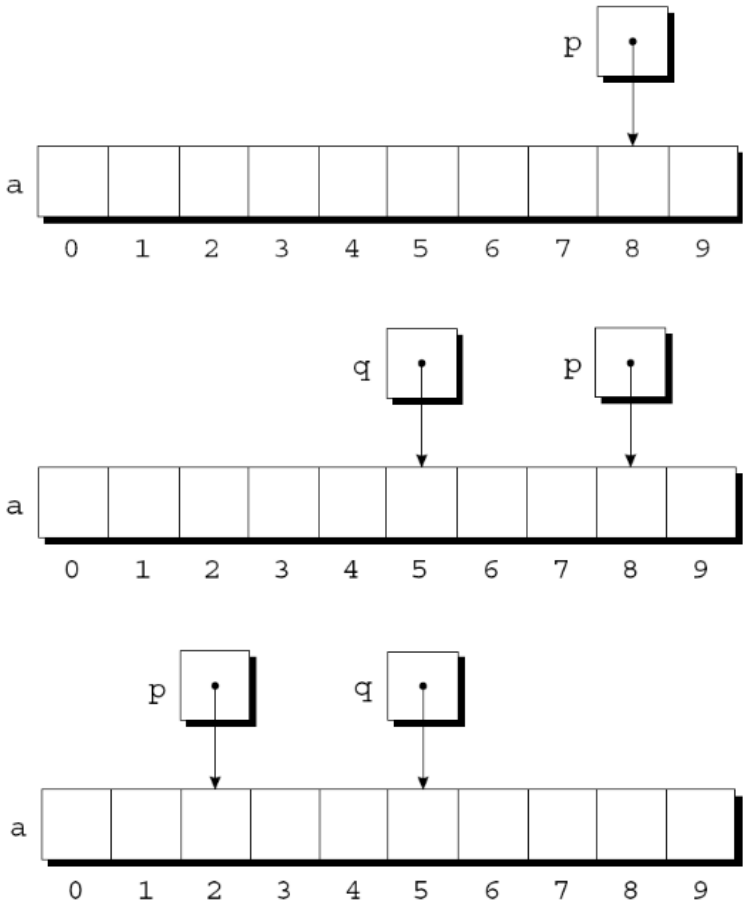
Subtracting an Integer from a Pointer

- If `p` points to `a[i]` then `p - j` points to `a[i-j]`.

```
int a[10] = {0,1,2,3,4,5,6,7,8,9};
int *p, *q, i;

p = &a[8]; // p points at a[8]
printf("p points at a[%d]\n", *p);

q = p - 3; // q points at a[8-3] = a[5]
printf("q points at a[%d]\n", *q);
```

```
p -= 6;  // p points at a[8-6] = a[2]
printf("p points at a[%d]\n", *p);
```

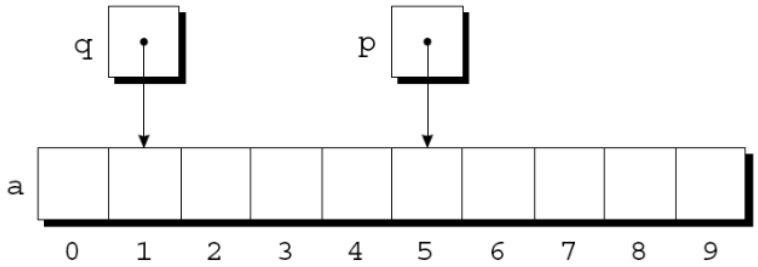
```
p points at a[8]
q points at a[5]
p points at a[2]
```



Subtracting one Pointer from Another

- When one pointer is subtracted from one another, the result is the distance (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`.

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

p = &a[5]; // p points at a[5]
q = &a[1]; // q points at a[1]

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

printf("p - q = %d\n", (int)(p - q));
printf("q - p = %d\n", (int)(q - p));


p - q = 4
q - p = -4
```



Comparing pointers

- You can compare pointers if they point to the same array. The outcome is determined by the relative positions of the elements in the array:

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

p = &a[5]; // p points at a[5]
q = &a[1]; // q points at a[1]

printf("p <= q: %d\n", p <= q); // p<=q is false b/c 5 > 1
printf("p >= q: %d\n", p >= q); // p>=q is true b/c 5 > 1


p <= q: 0
p >= q: 1
```

Using Pointers for Array Processing

- Using pointer arithmetic, you can visit array elements by increasing a pointer variable.
- Example: Sum elements of an array `a`

```
#define N 10

int a[N] = {11,34,82,7,64,98,47,18,79,20};
int sum, *p;

sum = 0;
for (p = &a[0]; p < &a[N]; p++) {
  sum += *p;
  printf("iter: %d sum: %d\n", *p, sum);
 }
printf("Sum of array elements: %d\n", sum);
```

```
iter: 11 sum: 11
iter: 34 sum: 45
iter: 82 sum: 127
iter: 7 sum: 134
iter: 64 sum: 198
iter: 98 sum: 296
iter: 47 sum: 343
iter: 18 sum: 361
iter: 79 sum: 440
iter: 20 sum: 460
Sum of array elements: 460
```

- Noteworthy: `p < &a[N]` is allowed for the loop condition even though `a[10]` does not exist (the maximum element of `a` is `a[9]`).
- The same result can be achieved with subscripting, of course (avoiding pointers) but using pointers can improve performance.

Combining * and ++ Operators

- Can you guess what the pointer equivalent of `a[i++] = j` is?

  - The non-pointer statement assigns `j` to the array element `a[i]` and then advances one element.
  - If `p` points at `a[i]`, then `*p++` points at `a[i++]`, so the statement is equivalent to `*p++ = j`.
  - Because `++` takes precedence over `*`, the compiler sees `*p++` as `*(p++)`, and the value of `p++` is `p` (because of the postfix operator).

```
int a[10]={0}, i, j, val, *p;

i = 1;
val = -1;
a[i++] = val;

i = 6;
val = -1;
p = &a[6];
(*p++) = val;

printf("0   1  2  3  4  5   6  7  8  9\n");
for (j=0; j<10; j++)
  printf("%d  ", a[j]);
```

```
0   1  2  3  4  5   6  7  8  9
0  -1  0  0  0  0  -1  0  0  0
```

- Note: Org-mode code blocks need the extra parentheses around `*p++` because the leading asterisk is otherwise interpreted
- `*p++` is handy in loops: to sum the elements of an array `a` as before, with `for (p = &a[0]; p < &a[N]; p++)`, you can now write

```
#define N 10

int a[N] = {11,34,82,7,64,98,47,18,79,20};
int sum, *p;

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

```
while (p < &a[N])
  sum = sum + (*p++);
printf("Sum of array elements: %d\n", sum);


Sum of array elements: 460
```

- Another application is the stack implementation: you can replace the variable `top` used to keep track of the top-of-stack position in the `contents` array by a pointer variable that points initially to `contents[0]`:

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

- How would the `push` and `pop` functions look like? We can now replace `contents[top++]` by `*top_ptr++`, and `contents[--top]` by `*--top_ptr`.

- Testing this and updating the other stack functions is an exercise.

Using an Array Name as a Pointer

- The name of an array can be used as a pointer to the first element in the array - this simplifies pointer arithmetic:

  ```
  int a[10];

  (*a) = 7; // stores 7 in a[0]

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

- Since `a + i` is the same as `&a[i]` and `*(a+i)` is the same as `a[i]`, array subscripting can be viewed as a form of pointer arithmetic.

- It is now easier to step through an array using pointers (last line):

  ```
  #define N 10
  int a[N] = {[0 ... N-1]=1}, *p; // initialize with a designated initializer
  for (int i=0;i<N;i++) printf("%d ",a[i]); puts("");// regular subscripting
  for (p = &a[0];p<&a[N];p++) printf("%d ",*p); puts("");// pointer subscripting
  for (p = a; p < a + N; p++) printf("%d ",*p); puts("");// pointer subscripting
  ```

```
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
```

- You cannot assign the array name a new value but have to copy it to a pointer first:

```
#define N 10
int a[N] = {[0 ... N-1]=1}, *p; // initialize with a designated initializer

p = a;
while (*p != 0)
  p--;

printf("%p %d\n", p, *p);


0x7ffdaf35b374 0
```

- As an application of array names as pointers, we revisit `reverse.c`:

```
/*************************************************************/
// reverse.c: reverse a series of numbers entered by the user
// Store numbers as int array with scanf in a for loop
// Print numbers in reverse order with printf in a for loop
/*************************************************************/
#include <stdio.h> // include input/output header file
#define N 10  // define constant array length with directive

int main() // main program
{
  int i;           // declare loop variable
  int a[N] = {0};  // macro definition (uses constant N)
  printf("Enter 10 numbers: "); // ask for input
  for ( i = 0; i < N; i++) { // loop over array (counting up)
    scanf("%d", &a[i]); // get the i-th array element from stdin
    printf("%d ", a[i]); // print the i-th array element to stdout
  }
  puts("");  // add new line
  printf("In reverse order: ");
```

```
  for ( i = N-1; i >= 0; i--) { // loop over array (counting down)
    printf("%d ", a[i]);
  }
  puts("");
  return 0; // confirm program ran successfully
} // end of main program
```

- After tangling the file (`C-u C-c C-v t`):

```
gcc reverse.c -o rev
echo 34 82 49 102 2 94 23 11 50 31 | ./rev
```

- The program reads 10 numbers, stores them in an array `a` and then steps backwards through the same array as it prints the numbers.

- In `reverse2.c`, array subscripting has been replaced with pointer arithmetic:

```
/*************************************************************/
// reverse2.c: reverse a series of numbers entered by the user
// Store numbers as int array with scanf in a for loop
// Print numbers in reverse order with printf in a for loop
// *** This version of the program uses pointer arithmetic ***
/*************************************************************/
#include <stdio.h> // include input/output header file
#define N 10  // define constant array length with directive

int main() // main program
{
  int *p;       // declare loop pointer
  int a[N] = {0};  // macro definition (uses constant N)
  p = a;
  printf("Enter 10 numbers: "); // ask for input
  for ( p = a; p < a + N; p++) { // loop over array (counting up)
    scanf("%d", p); // get the i-th array element *(p+i) from stdin
    printf("%d ", *p); // print the i-th array element to stdout
  };
  puts("");  // add new line
  printf("In reverse order: ");
  for ( p = a + N -1; p >= a; p--) { // loop over array (counting down)
```

```
    printf("%d ", *p);
  }
  puts("");

  return 0; // confirm program ran successfully
} // end of main program


Enter 10 numbers: 0 0 0 0 0 0 0 0 0 0
```

- After tangling the file (`C-u C-c C-v t`):

```
gcc reverse2.c -o rev2
echo 34 82 49 102 2 94 23 11 50 31 | ./rev2
```

- Note:

  1. `p < N` is wrong (comparing pointer and integer), but `p < a + N` is correct because the compiler sees `p < &a[0+N]`.
  2. `scanf` now takes `p` as second argument instead of `&a[i]`.
  3. When printing in reverse order, `i=N-1` becomes `p=a+N-1`. What the compiler sees now, is `p = &a[0+N-1]`.
  4. The end condition when counting down is `p>=a`. What the compiler sees is `p>=&a[0]`.

- It might be instructive to step through this program using a debugger.

Array Arguments (Revisited)

- When passed to a function, an array name is always treated as a pointer (to its first element) - the array "decays to a pointer".

- Example: `find_largest` returns the largest element in an array of integers.

```
#include <stdio.h>
#define N 5

int find_largest(int a[], int n);
```

```
int main() {
  // call function and pass array as a compound literal
  printf("%d\n", find_largest( (int []) {5, 6, 100, -3, 0}, 5));

  return 0;
}

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

```
100
```

```
100
```

- The array was passed as a *compound literal*, and only the pointer to the first element was passed and assigned to `a` in `find_largest`.
- When an ordinary variable is passed to a function, its value is copied and therefore no changes in the function affect the original.
- When an array is used as an argument, it is not protected against change since no copy is made of the array itself.

```
#define N 5
// function modifies array
void store_zeros(int a[], int n)
{
  int i;
  for (i = 0; i < n; i++)
    a[i] = 0;
}
int main(void)
{
  int b[N]={1,1,1,1,1}, i;
```

```
  for (i = 0; i < N; i++) printf("%d ", b[i]);
  store_zeros(b,5);
  puts("");
  for (i = 0; i < N; i++) printf("%d ", b[i]);
  return 0;
}
```

```
1 1 1 1 1
0 0 0 0 0
```

- To indicate that an array parameter won't be changed, include `const`:

```
#include <stdio.h>
#define N 5

int find_largest(const int a[], int n);

int main() {
  // call function and pass array as a compound literal
  printf("%d\n", find_largest( (int []) {5, 6, 100, -3, 0}, 5));

  return 0;
}

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

```
100
```

- There is no time penalty for passing a large array to a function, since no copy is made!
- An array parameter can be declared as a pointer - the compiler treats them identical.

```
#include <stdio.h>
#define N 5

int find_largest(int *a, int n);

int main() {
  // call function and pass array as a compound literal
  printf("%d\n", find_largest( (int []) {5, 6, 100, -3, 0}, 5));

  return 0;
}

int find_largest(int a[], int n)
{
  int i, max = a[0];
  for (i = 1; i < n; i++)
    if (a[i] > max) max = a[i];
  return max;
}


100
```

- This is not true for a variable! `int a[10]` reserves space for 10 integers, but `int *a` only reserves space for a pointer variable (not an array): `*a = 0;` will store 0 where `a` is pointing, and since we don't know where that is, the effect is undefined.
- A function with an array parameter can be passed an array slice, a sequence of consecutive elements. For example to find `find_largest` in the elements `b[5]...b[14]`, we can pass `b[5]` and the number 10.

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

Using a Pointer as an Array Name

- Since you can use an array name as a pointer, you can also subscript a pointer as though it were an array name:

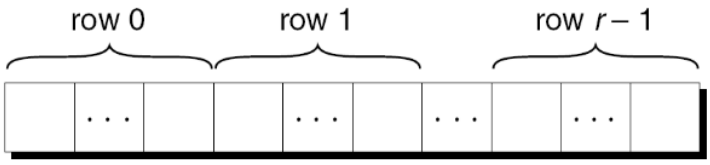
```
#define N 10
```

```
int a[N]={[0 ... 9] = 2}, i, sum = 0, *p = a;

for (i=0;i<N;i++)
  sum+=p[i];

printf("%d\n",sum);


20
```

- The compiler treats `p[i]` as `*(p+i)`. This will be useful when dynamically allocating memory (ch 17.3).
- I was surprised when I first saw the declaration + initialization `int *p = a` but the statement `p = a`. What's going on?
  - The declaration uses `*` as a type indicator and initializes the pointer with the array - now `p` points at `a[0]`.
  - The statement `p = a` with the already declared pointer `p` achieves the same but here, `*p = a` would be an error because `*` is the indirection operator, and the statement would try to set the value that `p` points at equal to the array. The error message: *Assignment to 'int' from 'int *' makes integer from pointer without a cast.*

Processing the Elements of a Multidimensional Array

- C stores two-dimensional arrays in row-major order:



- If the pointer `p` points at element `[0][0]` of the array, then we can visit every array element by incrementing `p`.

- Example: initializing all elements of a two-dimensional array - you can do it with nested `for` loops or with a single pointer loop:

```
#define NROWS 4
#define NCOLS 4

int a[NROWS][NCOLS], i, j, *p;

// initialize with nested loops
for (i = 0; i < NROWS; i++) {
  for (j = 0; j < NCOLS; j++) {
    a[i][j] = 0;
    printf("%d ", a[i][j]);
  }
 }
printf("\n");
// initialize with pointer arithmetic
for (p = &a[0][0];  // point to a[0][0]
     p <= &a[NROWS-1][NCOLS-1]; // a[0][1],...a[NROWS-1][NCOLS-1]
     p++) {
  (*p) = 1;
  printf("%d ", *p);
 }


0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

- Here is the same code but printing in matrix format:

```
#define NROWS 4
#define NCOLS 4

int a[NROWS][NCOLS], i, j, *p, k;

// initialize with nested loops
for (i = 0; i < NROWS; i++) {
  printf("\n");
  for (j = 0; j < NCOLS; j++) {
    a[i][j] = 0;
```

```
    printf("%d ", a[i][j]);
  }
 }
printf("\n");
// initialize with pointer arithmetic
k = NROWS;
for (p = &a[0][0];  // point to a[0][0]
     p <= &a[NROWS-1][NCOLS-1]; // a[0][1], a[0][2],...
     p++) {
  (*p) = 1;
  printf("%d ", *p);
  k--; // printout in matrix format
  if (!k) {
    printf("\n");
    k= NROWS;
  } // end printout in matrix format
 }
```

```
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
1 1 1 1
1 1 1 1
1 1 1 1
1 1 1 1
```

- Treating n-dimensional arrays as one-dimensional gives an efficiency advantage, especially with older compilers.

Processing the Rows of a Multidimensional Array

- To process elements just in one row `i` of a 2D array:

```
p = &a[i][0] // pointer p points at first column of i-th row
p = a[i]; // same! &a[i][0] = &(*(a[i] + 0)) = &*a[i] = a[i]
```

- Here, `*(a[i][0]) = *(a[i] + 0) because a[i] = *(a + i)`

- To to clear row `i` of an array `a`:

```
#define NROWS 4
#define NCOLS 4

int a[NROWS][NCOLS], *p, i = 0; // not setting i: segmentation fault!

for (p = a[i]; p < a[i] + NCOLS; p++) {
  (*p) = 0;
  printf("%d ", *p);
 }


0 0 0 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: any function that is designed to work with 1D arrays will also work with a row beloning to a 2D array!
- Example: we can use `find_largest` to determine the largest element in row `i` of the two-dimensional array `a`:

```
#include <stdio.h>

#define NROWS 4
#define NCOLS 4

int find_largest(int a[], int n);
int find_largest_ptr(int a[], int n);

int main()
{
  int a[NROWS][NCOLS] = {
    {1,2,3,4},
    {5,6,7,8},
    {9,10,11,12},
    {13,14,15,16}
  };
```

```
  int i = 3; // row index

  printf("%d\n", find_largest(a[i], NCOLS));
  printf("%d\n", find_largest_ptr(a[i], NCOLS));

  return 0;
}

int find_largest(int a[], int n)
{
  int i, max = a[0];
  for (i = 1; i < n; i++)
    if (a[i] > max) max = a[i];
  return max;
}

int find_largest_ptr(int a[], int n)
{
  int i, *p = a, max = *p;
  for (p = a; p < a + n; p++)
    if (*p > max) max = *p;
  return max;
}
```

```
16
16
```

- Note:
  1. do not replace `n` in the prototype and the function definition by `NCOLS`. That will lead to an error because you're trying to type a constant - `int NCOL` will become `int 4`.
  2. The code shows the function both with a regular and with a pointer loop. Notice that the loop only works on one row.

Processing the Columns of a Multidimensional Array

- This is not as easy since 2D arrays are stored by row, not column. We need to tell the computer to point at `[NCOLS]` one-dimensional arrays

- This code clears column `j` of an array `a`:

```
#define NROWS 4
#define NCOLS 4

int a[NROWS][NCOLS] = {
  {1,2,3,4},
  {5,6,7,8},
  {9,10,11,12},
  {13,14,15,16}
};
int (*p)[NCOLS], j=1;

for (p = a; p < a + NROWS; p++) {
  (*p)[j] = 0;
  printf("%d\n", (*p)[j]);
 }


0
0
0
0
```

- Here, `p` is declared to be a pointer to an array of length `NCOLS` whose elements are integers. Without the parentheses around `*p` in `(*p)[NCOLS]`, the compiler would treat p as an array of `NCOLS` pointers instead of a pointer to an array.

- The expression `(*p)[j]`, `(*p)` represents an entire row of `a`, so `(*p)[j]` selects the element in column `j` of that row. Again, the parentheses are essential because the compiler would interpret `*p[j]` as `*(p[j])`

- This code sums up the elements of column `j`:

```
#define NROWS 4
#define NCOLS 4

int a[NROWS][NCOLS] = {
  {1, 2, 3, 4},
  {5, 6, 7, 8},
```

```
  {9, 10, 11, 12},
  {13, 14, 15, 16}
};
int (*p)[NCOLS], j = 3, sum = 0;

for (p = a; p < a + NROWS; p++) {
  sum += (*p)[j];
 }
printf("%d\n", sum);

40
```

- This code computes the largest element by column.

```
#define NROWS 4
#define NCOLS 4

int a[NROWS][NCOLS] = {
  {1, 2, 3, 4},
  {5, 6, 7, 8},
  {9, 10, 11, 12},
  {13, 14, 15, 16}
};

int (*p)[NCOLS] = a, j = 2, largest = *(*p);
printf("%d\n",largest); // initial value

for (p = a; p < a + NROWS; p++)
  if ( (*p)[j] > largest ) largest = (*p)[j];

printf("%d\n",largest); // largest value in column j

1
15
```

Using the Name of a Multidimensional Array as a Pointer

- You can use the name of a one-dimensional array as a pointer (to its 0th element), and you can use the name of any array regardless of its

dimensions.

- The array `int a[NROWS][NCOLS]` is not a pointer to `a[0][0]` but to `a[0]`: when used as a pointer, `a` has type `(int *) [NCOLS]`, that is a pointer to an integer array of length `NCOLS`.
- That's why the following loops get the same results:

```
// using addresses
for (p = &a[0]; p < &a[NROWS]; p++)
  (*p)[i] = 0;
// using array name as pointer
for (p = a; p < a + NROWS; p++)
  (*p)[i] = 0;
```

- That's also why in the code below, the function `single` (that initializes the matrix `a` with `n = NROWS * NCOLS` elements) can be called in either of these three ways:

```
single(&a[0][0], n); // address
single( (int *)a, n); // explicit cast
single(a[0],n); // implicit cast a[0] = (int *)[NCOLS]
```

Review questions

What does "pointer arithmetic" mean?

It refers to using pointers to access array elements that they're pointing at. You can add an integer to a pointer, subtract an integer from a pointer, subtract pointers from one another, and compare pointers. All of these operations are translated to array index operations.

Give an example for adding an integer to a pointer.

```
int a[10]={100,101,102,103,104,105,105,800,900,1000}, *p, i;
p = &a[0]; // p points at a[0] = 100
i = 4;
printf("*p = %d, a[%d] = %d\n",*(p+i), i, a[i]);
printf("p = %p, &a[%d] = %p\n",(p+i), i, &a[i]);

*p = 104, a[4] = 104
p = 0x7ffe93b25590, &a[4] = 0x7ffe93b25590
```

Give an example for subtracting an integer from a pointer.

```
int a[10]={100,101,102,103,104,105,105,800,900,1000}, *p, i;
p = &a[9]; // p points at a[9] = 9
i = 3;
printf("*p = %d, a[%d] = %d\n",*(p-i), 9-i, a[9-i]);
printf("p = %p, &a[%d] = %p\n", (p-i), 9-i, &a[9-i]);
```

```
*p = 105, a[6] = 105
p = 0x7ffee55585f8, &a[6] = 0x7ffee55585f8
```

Give an example for subtracting one pointer from another.

```
int a[10]={100,101,102,103,104,105,105,800,900,1000}, *p, *q;
p = &a[9]; // p points at a[9]
q = &a[3];
printf("p - q = %ld\n", p-q); // distance between locations as indices
printf("*p - *q = %d\n", *p-*q); // difference of values
printf("&a[9] - &a[3] = %ld\n", &a[9] - &a[3]); // distance between addresses
```

```
p - q = 6
*p - *q = 897
&a[9] - &a[3] = 6
```

Give an example for comparing two pointers

```
int a[10]={100,101,102,103,104,105,105,800,900,1000}, *p, *q;
p = &a[9]; // p points at a[9]
q = &a[3]; // q points at a[3]
printf("p == q : %d\n", p==q);
```

```
p == q : 0
```

If `p = &a[3]`, what is `p+=3` ?

If `p` points at the array element `a[3]`, then `p+=3` (or `p = p + 3`) points at the array element `a[3+3] = a[6]`.

Difference between pointers

A pointer `p` points at `a[9] = 100`, and another pointer `q` points at `a[6] = 25`. What is the difference between `p - q` and `*p - *q` and what will be the output of these operations?

> With `p - q`, we subtract one pointer from one another, which is computed as their distance in array elements, so `9 - 6 = 3`.
>
> With `*p - *q`, we subtract the values of the variables that the pointers point at, or `100 - 25 = 75`.

```
int a[10]={[6]=25,[9]=100}, *p = &a[9], *q = &a[6];
printf("p - q: %d\n", (int)(p - q));
printf("*p - *q: %d", *p - *q);
```

```
p - q: 3
*p - *q: 75
```

Summing one-dimensional array using a pointer

Sum an array `a` with the elements `100, 200, 300, 400, 500` using a pointer: use a `for` loop first, and a `while` loop next.

```
int a[5] = {100, 200, 300, 400, 500}, sum, *p, *q;
sum=0;
for(p=&a[0];p<&a[5];p++)
  sum+=*p;
printf("%d\n",sum);

q = &a[0];
while (q < &a[5])
  sum+=*q++;
printf("%d\n",sum);
```

```
1500
3000
```

Step through array from beginning and from end using pointers

Assume you have an `int a[10]`, with `a[0] = -1` and `a[9] = 1`. A pointer `p` is initialized `p = a`. How would you step through the array from beginning and from end to print `a`?

```
#define N 10
int a[N] = {[0]=-1,[N-1]=1}, *p;
p = a; // p points at a[0]

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


-1 0 0 0 0 0 0 0 0 1
1 0 0 0 0 0 0 0 0 -1
```

Find largest

Pass the integers `5 6 100 1001 -3 0 1000 -33` as a compound literal to a function `find_largest`, and `return` the largest integer among the last four elements of the array only.

```
#include <stdio.h>

int find_largest(int *a, int n);

int main()
{
  printf("%d\n", find_largest((int[]){5,6,100,1001,-3,0,1000,-33},8));
  return 0;
}

int find_largest(int *a, int n)
{
  int i, *p, max;
  p = a; // p points at a[0]
  max = *p; // max has the value a[0]
```

```
  for (p = a + 5; p < a + n; p++)
    if (*p > max) max = *p;
  return max;
}
```

```
1000
```

What could you also do if you didn't want the loop to be dependent on 5 but if you wanted to control the subset from `main`, and if you did not use a compound literal but instead declared and initialized an array `a` in `main`?

```
#include <stdio.h>
#define N 8

int find_largest(int *a, int n);

int main()
{
  int b[N] = {5,6,100,1001,-3,0,1000,-33};
  int sub = 4;
  printf("%d\n", find_largest(&b[sub],N-sub));
  return 0;
}

int find_largest(int *a, int n)
{
  int i, *p, max;
  p = a;
  max = *p;
  for (p = a; p < a + n; p++)
    if (*p > max) max = *p;
  return max;
}
n
```

Initialize matrix and print with array subscripting & pointer arithmetic

Assume `a` is a 4x4 matrix (2D array). Initialize `a` with the values `[0..16]`. Pass `a` to a function `nested` that initializes its elements to 1 using nested

`for` loops. Then pass it to a function `single` that changes its elements to 0.
Print all result in `main`.

```
#include <stdio.h>

#define NROWS 4
#define NCOLS 4

void nested(int a[NROWS][NCOLS]); // must specify size of 2nd dimension
void single(int *a, int n);
void print(const int a[NROWS][NCOLS]);

int main(void)
{
  int a[NROWS][NCOLS] = {0}; // initialize matrix to 0
  int n = NROWS * NCOLS;

  print(a);
  nested(a); // change values to 1
  print(a);
  //single(&a[0][0], n); // change values to zero: pass
  //single( (int *)a, n); // flatten array with pointer to integer cast
  single(a[0],n); // a[0] = (int *)a
  // address of the first element and
  // the total number of elements
  print(a);

  return 0;
}

void nested(int a[NROWS][NCOLS]) // must specify size of 2nd dimension
{
  int i,j;
  for (i=0;i<NROWS;i++)
    for (j=0;j<NCOLS;j++)
      a[i][j]=1;
}

void single(int *a, int n)
{
```

```
  int *p = a;
  for (p = a; p < a + n; p++)
    (*p) = 0;
}

void print(const int a[NROWS][NCOLS])
{
  for (int i = 0; i < NROWS; i++) {
    for (int j = 0; j < NCOLS; j++) {
      printf("%d ", a[i][j]);
    }
    printf("\n");
  }
}
```

```
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
1 1 1 1
1 1 1 1
1 1 1 1
1 1 1 1
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
```

```
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
1 1 1 1
1 1 1 1
1 1 1 1
1 1 1 1
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
```

Initialize matrix with random numbers

- To initialize a 4x4 matrix with random integers, you can use the `stdlib::rand` function. To ensure different random numbers on each execution, seed the random number generator with `srand` and `time(NULL)`.
- The code initializes the matrix using a function `init_rand`. In the function, array subscripting is used. The `print_mat` function prints the resulting matrix:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define NROWS 4
#define NCOLS 4

void init_rand(int *a, int n); // initialize with random integers
void print_mat(const int a[NROWS][NCOLS]); // print matrix

int main (void)
{
  int a[NROWS][NCOLS]; // un-initialized matrix
  int n = NROWS * NCOLS;

  init_rand((int *)a, n);
  print_mat(a);

  return 0;
}

void init_rand(int *a, int n)
{
  srand(time(NULL)); // seed random number generator
  int *p = a;
  for (p = a; p < a + n; p++)
    (*p) = rand() % 100; // generate random integers in [0,99]
}

void print_mat(const int a[NROWS][NCOLS])
{
```

```
  for (int i = 0; i < NROWS; i++) {
    for (int j = 0; j < NCOLS; j++) {
printf("%d ", a[i][j]);
    }
    printf("\n");
  }
}
```

```
77 3 49 84
10 16 95 47
63 32 37 84
19 20 5 36
```

Questions & Answers

1. Does `j` in `p + j` add `j` to the address stored in the pointer `p`?

   No - what is added depends on the type of pointer. if `p` is of type `int *`, `p + j` adds `4 * j` to `p` because `int` values are stored using 4 bytes.

2. What is better, pointer arithmetics or array subscripting?

   Depends on the compiler.

3. Is `i[a]` the same as `a[i]`?

   Yes! `i[a]` is seen as `*(i + a) = *(a + i) = a[i]` because pointer addition is commutative.

4. If `*a` and `a[]` are equivalent as function parameters, which is better?

   Style issue: if you do pointer arithmetic, the first indicates that, and if you do array subscripting, the second, but `*a` is much more common.

5. Are arrays and pointers interchangeable?

   No. array parameters are interchangeable with pointer parameters but array variables are not the same as pointer variables. The name of an array is not technically a pointer but is cast to a pointer when necessary. Also, `sizeof(a)`

is the byte-size of an array `a`, but if `p` is a pointer variable, `sizeof(p)` is the byte-size required to store a pointer (not the variable).

6. If `a` is a two-dimensional array, why can we pass `a[0]` but not `a` itself to `find_largest` since they both point to the beginning of the array?

   Because `a` has the wrong type: as an argument, `a` decays to a pointer to the array, but `find_largest(int *a, int n)` expects a pointer to an integer, namely the first element of `a`. However, `a[0]` has type `int *`, so it is an acceptable argument for `find_largest`.

Exercises

1. Suppose that the following declarations are in effect:

```
int a[] = {5, 15, 34, 54, 1414, 2, 52, 72};
int* p = &a[1];
int* q = &a[5];
```

   (a) What is the value of `*(p+3)`?

   (b) What is the value of `*(q-3)`?

   (c) What is the value of `q-p`?

   (d) Is the condition `p < q` true or false?

   (e) Is the condition `*p < *q` true or false?

2. What will be the contents of the `a` array after the following statements are executed?

```
#define N 10

int a[N] = {1,2,3,4,5,6,7,8,9,10};
int* p = &a[0];
int* q = &a[N-1];
int temp;

while (p < q) {
  temp = *p;
```

```
  (*p++) = *q;
  (*q--) = temp;
 }
```

3. Rewrite the following function to use pointer arithmetic instead of array subscripting. In other words: eliminate the variable `i` and all uses of the `[]` operator. Make as few changes as possible.

```
int sum_array(const int a[], int n)
{
  int i, sum;

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

4. Rewrite the following function to use pointer arithmetic instead of array subscripting. In other words: eliminate the variable `i` and all uses of the `[]` operator. Make as few changes as possible.

```
void store_zeros(int a[], int n)
{
  int i;
  for (i = 0; i < n; i++)
    a[i] = 0;
}
```

5. Update the stack program with pointers to arrays.

6. Write and test the following function:

```
double inner_product(const double *a, const double *b, int n);
```

   - `a` and `b` are both pinters to arrays of length `n`.
   - The function should return a[0] * b[0] + a[1] * b[1] + ... + a[n-1] * b[n-1].
   - Use pointer arithmeic - not subacripting - to visit array elements.

Programming Projects

I will be sharing sample solutions to these projects after the submission deadline.

1. Write a program that reads a message, and then prints the reversal of the message. Sample input/output:

   ```
   Enter a message: Don't get mad, get even.
   Reversal is: .neve teg ,dam teg t'noD
   ```

   Hint: Read the message one character at a time using `getchar`, and store the characters in an array. Stop reading when the array is full or the character read is ".

2. Revise the message reversal program (1) to use a pointer instead of an integer to keep track of the current position in the array.

3. Write a program that reads a message, then checks whether it's a *palindrome* (the letters in the message are the same from left to right as from right to left):

   ```
   Enter a message: He lived as a devil, eh?
   Palindrome:

   Enter a message: Madam, I am Adam
   Not a palindrome
   ```

   Hint: Ignore all characters that aren't letters. Use integer variables to keep track of positions in the array.

4. Revise the palindrome finding program (3) to use pointers instead of integers to keep track of positions in the array.

5. Simplify the program (2) by taking advantage of the fact that an array name can be used as a pointer.

6. Simplify the program (4) by taking advantage of the fact that an array name can be used as a pointer.