Lecture 6

Arrays and Pointers

CPSC 275
Introduction to Computer Systems

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;
```


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:

```
*p++ = j;
```

- Because the postfix version of `++` takes precedence over `*`, the compiler sees this as

```
*(p++) = j;
```


Combining the * and ++ Operators

- Possible combinations of `*` and `++`:

Expression	*Meaning*
`*p++` or `*(p++)`	Value of expression is `*p` before increment; increment `p` later
`(*p)++`	Value of expression is `*p` before increment; increment `*p` later
`*++p` or `*(++p)`	Increment `p` first; value of expression is `*p` after increment
`++*p` or `++(*p)`	Increment `*p` first; value of expression is `*p` after increment

Combining the * and ++ Operators

- The most common combination of `*` and `++` is `*p++`, 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++;
```


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.
- This relationship simplifies pointer arithmetic and makes both arrays and pointers more versatile.

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` of `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's not possible to assign it a new value.
- Attempting to make it point elsewhere is an error:

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

- Copy `a` into a pointer variable, then change the pointer variable:

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


Array Arguments

- 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;
}
```


Array Arguments

- 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 isn't copied.

Array Arguments

- 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 don't affect the variable.
- In contrast, an array used as an argument isn't protected against change.

Array Arguments

- 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;
}
```

13

Array Arguments

- To indicate that an array parameter won't 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 check that no assignment to an element of `a` appears in the body of `find_largest`.

14

Array Arguments

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

15

Array Arguments

- *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)
{
  …
}
```

16

Array Arguments

- Although declaring a *parameter* to be an array is the same as declaring it to be a pointer, the same isn't 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;
```

17

Array Arguments

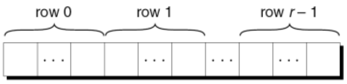
- *Consequence 4:* A function with an array parameter can be passed an array "slice"—a sequence of consecutive elements.
- An example that applies `find_largest` to elements 5 through 14 of an array `b`:

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

18

Pointers and Multidimensional Arrays

- C stores two-dimensional arrays in row-major 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];
```

- Using nested `for` loops:

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


Processing the Elements of a Multidimensional Array, cont'd

- 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 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'd initialize `p` to point to element 0 in row `i` in the array `a`:

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

or we could simply write

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


Dynamic Storage Allocation

- C's data structures, including arrays, are normally fixed in size.
- Fixed-size data structures can be a problem, since we're forced to choose their sizes when writing a program.
- Fortunately, C supports ***dynamic storage allocation:*** the ability to allocate storage during program execution.
- Dynamic storage allocation is done by calling a memory allocation function.

Memory Allocation Functions

- The `<stdlib.h>` header declares three memory allocation functions:

`malloc`—Allocates a block of memory but doesn't initialize it.
`calloc`—Allocates a block of memory and clears it.
`realloc`—Resizes a previously allocated block of memory.

- These functions return a value of type `void *` (a "generic" pointer).

`malloc()`

- Prototype for the `malloc` function:

```
void *malloc(size_t size);
```

- `malloc` allocates a block of `size` bytes and returns a pointer to it.
- `size_t` is an unsigned integer type defined in the library.
- Example:

```
char *p;
p = (char *) malloc(10);
```


Null Pointers

- If a memory allocation function can't locate a memory block of the requested size, it returns a ***null pointer.***
- A null pointer is a special value that can be distinguished from all valid pointers.
- After we've stored the function's return value in a pointer variable, we must test to see if it's a null pointer.

Null Pointers

- An example of testing `malloc`'s return value:

```
p = malloc(10000);
if (p == NULL) {
  /* allocation failed; take appropriate action */
}
```

- `NULL` is a macro (defined in various library headers) that represents the null pointer.
- Some programmers combine the call of `malloc` with the `NULL` test:

```
if ((p = malloc(10000)) == NULL) {
  /* allocation failed; take appropriate action */
}
```


Null Pointers

- Pointers test true or false in the same way as numbers.
- All non-null pointers test true; only null pointers are false.
- Instead of writing
  `if (p == NULL) …`
  we could write
  `if (!p) …`
- Instead of writing
  `if (p != NULL) …`
  we could write
  `if (p) …`

Dynamically Allocated Arrays

- Suppose a program needs an array of `n` integers, where `n` is computed during program execution.
- We'll first declare a pointer variable:

```
int *a;
```

- Once the value of `n` is known, the program can call `malloc` to allocate space for the array:

```
a = (int *) malloc(n * sizeof(int));
```

- Always use the `sizeof` operator to calculate the amount of space required for each element.

Dynamically Allocated Arrays, cont'd

- We can now ignore the fact that `a` is a pointer and use it instead as an array name, thanks to the relationship between arrays and pointers in C.
- For example, we could use the following loop to initialize the array that `a` points to:

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

- We also have the option of using pointer arithmetic instead of subscripting to access the elements of the array.

Deallocating Storage

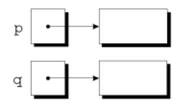
- `malloc` and the other memory allocation functions obtain memory blocks from a storage pool known as the ***heap.***
- Calling these functions too often—or asking them for large blocks of memory—can exhaust the heap, causing the functions to return a null pointer.
- To make matters worse, a program may allocate blocks of memory and then lose track of them, thereby wasting space.

Deallocating Storage

- Example:

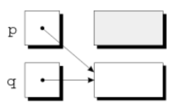
```
p = malloc(…);
q = malloc(…);
p = q;
```

- A snapshot after the first two statements have been executed:



Deallocating Storage

- After `q` is assigned to `p`, both variables now point to the second memory block:



- There are no pointers to the first block, so we'll never be able to use it again.

Deallocating Storage

- A block of memory that's no longer accessible to a program is said to be ***garbage.***
- A program that leaves garbage behind has a ***memory leak.***
- Some languages provide a ***garbage collector*** that automatically locates and recycles garbage, but C doesn't.
- Instead, each C program is responsible for recycling its own garbage by calling the `free` function to release unneeded memory.

The `free` Function

- Prototype for `free`:

```
void free(void *ptr);
```

- `free` will be passed a pointer to an unneeded memory block:

```
p = malloc(…);
q = malloc(…);
free(p);
p = q;
```

- Calling `free` releases the block of memory that `p` points to.

The "Dangling Pointer" Problem

- Using `free` leads to a new problem: ***dangling pointers.***
- `free(p)` deallocates the memory block that `p` points to, but doesn't change `p` itself.
- If we forget that `p` no longer points to a valid memory block, chaos may ensue:

```
char *p = malloc(4);
…
free(p);
…
*p = 'a';    /*** WRONG ***/
```

The “Dangling Pointer” Problem

- Dangling pointers can be hard to spot, since several pointers may point to the same block of memory.
- When the block is freed, all the pointers are left dangling.

dotprod3.c

- Rewrite **dotprod1.c** so that arrays are dynamically allocated using malloc() function.