Arrays and Pointers

(K&R, chapter 5)

Data Types and Memory Usage

- Variables occupy space in computer memory
- Formally, sizes are "implementation-dependent"...
- ...but some sizes are very common
  - char: 1 byte (8 bits)
  - short: 2 bytes (16 bits)
  - int: 4 bytes (32 bits)
  - long: 8 bytes (64 bits)
  - pointer: 4 bytes or 8 bytes

- Defined standard for real numbers:
  - float: 4 bytes
  - double: 8 bytes
  - not implementation-dependent

Data in Memory

- Declare an integer:
  - int myInt;
  - double myDouble;
- Reserves 4 bytes of memory for an integer, 8 bytes for a double

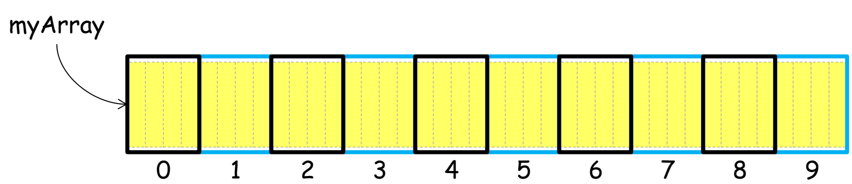
- "myInt" and "myDouble" refer to the reserved memory locations
  - Variable *addresses* are the first byte's address

Arrays

Read K&R, sections §1.6, §1.9, and §5.3

Arrays

- Array: collection of *elements* of the same type
  - Array elements can be used as individual variables
- Elements are *contiguous* in memory
- Elements are numbered starting at 0
  - The index is the *offset* from the beginning address



Array sizing

- Array size is the size of an element, times the number of elements
- In ANSI C (C89/C90), array size is *static*
  - must be specified by a constant expression, at compile time
- In C99, array size can be *variable*
  - a function can specify an array size with a variable
- Arrays cannot be *resized* after creation
  - "Resizing" (adding elements) requires creating a new, bigger array and then copying values from the old version to the new one

Static, Variable, and Dynamic Arrays

- Static: size determined at compile time
  - e.g. `int xarray[100];`
- Variable: *local storage*, size determined at beginning of run time

```
int nElts = atoi( argv[1] );
int xarray[ nElts ];
```

- Dynamic: *heap* memory explicitly allocated, released by function calls

```
int *xp = malloc(nElts * sizeof(int));
free(xp);
```

Initializing arrays

- When declaring an array, its elements can be given initial values
- List initial values in curly braces
- Initial values must be expressions that evaluate to constants
- If not enough initial values are supplied, the remaining elements are uninitialized
  - What happens if too many initial values?

Array declaration examples

- Fixed-size array, with initialization (unused)
- Fixed-size array, with string initialization (unused)
- Variable-size array

```
/* variable-array example
   Compile with  "gcc -Wall -std=c99  -o example  example.c"
*/
#include <stdio.h>

int main(int argc, char **argv)
{
   // Unused examples:
   char digits[10] = { '0', '1', '2', '3', '4', '5', '6', '7', '8', '9' };
   char vowels[] = "aeiouAEIOU";      // extra NULL at the end

   printf("How many elements? ");
   int size;
   scanf(" %i", &size);
   float elements[ size ];     // for user's values
   for ( int i = 0; i < size; i++ ) {
      printf("element %i? ", i);
      scanf( " %f", &elements[i] );
   }
   for (int j = size - 1; j >= 0; j--)
      printf( "%2i %10.2f\n" , j, elements[j] );

   return 0;
}
```

Array assignments

- You *can* assign new values to individual elements of an array
- You *cannot* assign one array to another
  - do it element-by-element in a loop, instead

- This doesn't work:

```
int i, a1[10], a2[10];

// fill a1 first (ok)
for (i = 0; i < 10; i++)
    a1[i] = i*i/2;

// try to copy a1
a2 = a1;      may not compile

// change original a1
a1[0] = -9;
```

- What happens to a2?

Copying Arrays

- Use a loop:
  - ```
    for (i = 0; i < ARRAY_SIZE; i++)
       new_array[i] = old_array[i];
    ```
- For text strings – arrays of characters terminated by NULL – use **`strncpy()`**:
  - ```
    strncpy(new_str, old_str, ARRAY_SIZE);
    ```
  - **`ARRAY_SIZE`** specifies the *maximum* number of characters to copy – avoids "buffer overflow"
  - **`strcpy()`** is simpler, but permits buffer overflows

Exercises

- Exercise: Initialize array of 100 Fibonacci numbers with first 2
- Exercise: write a function of an array and an index, that calculates the *next* Fibonacci number
- Exercise: write a program that calculates a requested number of Fibonacci numbers

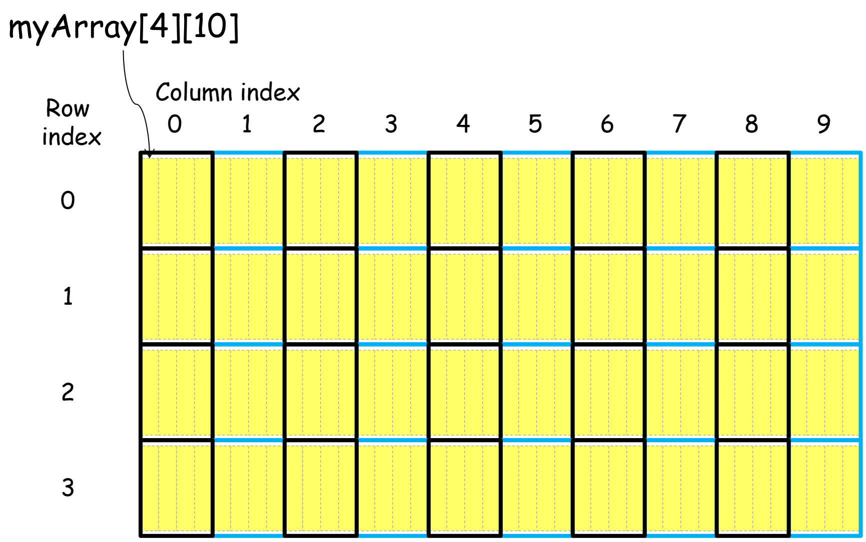
Multidimensional Arrays

Added in c99 standard

Two-Dimensional Arrays

- A "rectangle" of *elements* *(see next slide)*
  - Rows and columns
  - C uses a **row-major** organization
    - Vertical "y" axis comes first
- Rows are *contiguous* in memory
  - Rows follow each other immediately
- Example definitions:
  - `int myarray[4][10];`
  - ```
    int ysize = 4, xsize = 10;
    double grid[ysize][xsize];
    ```
    ```

A Two-Dimensional Array in Memory



Accessing Array Elements

- Straightforward use of nested "for loops"
  - // row-wise access, looks like previous diagram:

```
for (y = 0; y < ysize; y++) {
    for (x = 0; x < xsize; x++)
        printf("%d ", grid[y][x]);
    printf("\n");
}
```

  - // column-wise access, looks "rotated":

```
for (x = 0; x < xsize; x++) {
    for (y = 0; y < ysize; x++)
        printf("%d ", grid[y][x]);
    printf("\n");
}
```

Correspondence of 1-D and 2-D Arrays

- Row-major 2-D array, with "ysize" rows and "xsize" columns of elements, can be treated as a 1-D array with ("ysize" × "xsize") elements

```
-
 int array1D [ ysize * xsize ];
 int array2D [ysize][xsize];

        ⋮

 for (y = 0; y < ysize; y++)
     for (x = 0; x < xsize; x++)

         array2D[y][x] =
                     array1D[y*xsize + x] ;
```

An Example of A Multidimensional Array

A 24-bit RGB image *(e.g*. "gif" format picture):

```
unsigned char picture [rows][columns][3];
unsigned char red_layer [rows][columns];
unsigned char green_layer [rows][columns];
unsigned char blue_layer [rows][columns];

      ⋮

for (y = 0; y < rows; y++)
    for (x = 0; x < columns; x++) {
        red_layer[y][x] = picture[y][x][0];
        green_layer[y][x] = picture[y][x][1];
        blue_layer[y][x] = picture[y][x][2];
    }
```

Pointers

- Pointers are memory addresses
- The “address-of” operator (**&**) gives the *memory address* of a variable
  - **`printf( "%x  %p\n", a, &a );`**
- What happens if you try to print the address of a constant?
  - **`printf( "%x  %p\n", 77, &77 );`**
- Array names are pointers
- Pointer variables hold pointers

Pointers and Coding – Description

```
int main(int argc, char**argv)
{
   int a;
   int *pa;

   a = 7;
   pa = &a;

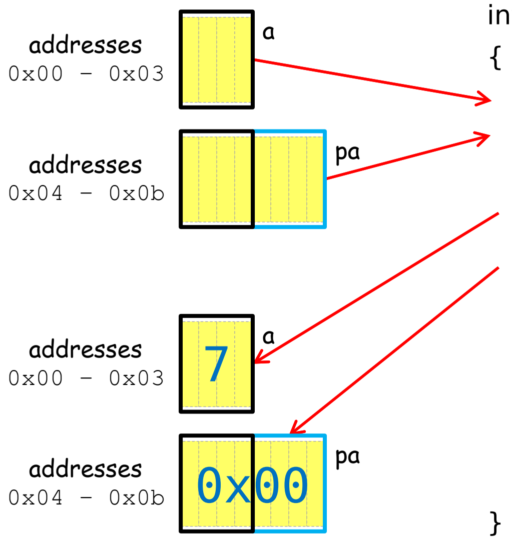
   printf("%d  %p\n", a, &a,

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

   return 0;
}
```

- "a" refers to someplace in memory
  - that place contains "7"
- "pa" refers to another place in memory
  - *that* place gets the address of the memory that holds the "7"
- Print:
  - contents of "a"
  - address of "a"
- Print:
  - contents of "pa"
  - what "pa" points to

Pointers and Coding – Picture




```
int main(int argc, char**argv)
{
   int a;
   int *pa;

   a = 7;
   pa = &a;

   printf("%d  %p\n", a, &a,

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

   return 0;
}
```

try:

```
#include<stdio.h>
#define ARRAYSIZE 10
int main(int argc, char**argv)
{
   char ary[ARRAYSIZE];
   int i;
   for (i = 0; i < ARRAYSIZE; i++)
      ary[i] = i*i;

   printf("ary: %p\n", ary);
   for(i = 0; i < ARRAYSIZE; i++)
      printf("%2d:  %p  %#02x  %d\n",
         i,  &ary[i],  ary[i],  ary[i]);

   return 0;
}
```

- Create an array of chars, and fill it with something
- Print array's address
- Loop over each element – print:
  - element's index
  - element's address
  - element's contents

remarks

- Use of the ARRAYSIZE macro
- **&** - "*address-of*" operator
- Compare array's value (address) to first element's address – they're the same
- Observe how the element addresses change – compare to the size of a char

modify:

```
#include<stdio.h>
#define ARRAYSIZE 10
int main(int argc, char**argv)
{
   unsigned ary[ARRAYSIZE];
   int i;
   for (i = 0; i < ARRAYSIZE; i++)
      ary[i] = i*i;

   printf("ary: %p\n", ary);
   for(i = 0; i < ARRAYSIZE; i++)
      printf("%2d:  %p  %#02x  %d\n",
         i,  &ary[i],  ary[i],  ary[i]);

   return 0;
}
```

- Change array type from "char" to "int"
- Leave the rest unchanged:
  - Print array's address
  - Loop over each element – print:
    - element's index
    - element's address
    - element's contents
- How do element addresses change now?

Arrays and Pointers

- "ary" and "&ary[0]" both refer to the same memory address
- The "*dereference*" operator (*) works on an address, returns the contents of memory at that address
- Dereference:
  - ary[0] contains 0*0, or 0
  - *ary contains *the same* 0*0, or 0

  - What about ary[1]?
  - *(ary+1) is the same

Arrays, Pointers, and Functions

- Arrays can’t be passed as function arguments
- Arrays can't be returned as function values
- Pointers *can* be passed and returned
- Functions can:
  - receive *pointers* as references to arrays
  - return pointers into *received* arrays
  - MUST NOT return pointers into *local* arrays!

Arrays, Pointers, and Functions

- Functions can receive array *names* – same as pointers to the arrays' first elements
  - Also needs to know the ***size*** of the array – usually passed in as another argument
- *Example:*

```
void fill_array(float ary[], unsigned size)
{
    unsigned i;
    for (i = 0; i < size; i++)
        ary[i] = 0.01;
}
```

can be written as: `(float *ary, …`

be careful returning pointers

- Don't return a pointer to one of the function's local variables or arrays!
  - These variables will be destroyed when the function finishes executing.
  - The pointer will therefore point to invalid locations

be careful returning pointers

- You *can* return a pointer that points into an array that was originally passed into the function as an argument
- Example:

```
// return a pointer to the middle of the array:
float *split_array( float *ary,
                    unsigned size )
{
    return &(ary[ (size+1)/2 ]);
}
```

Pointer Arithmetic

- You can assign to, add to, and subtract from, a pointer

```
int ary[10], *pary;
pary = ary;      // pary points to 1st array element
pary += 3;       // now points to 4th array element
pary--;          // backs up to 3rd array element
```

- Operations work in units equal to the element's size
  - int, unsigned: a unit is 4 bytes
  - double: a unit is 8 bytes
  - etc.

Fibonacci Using Pointers

- Variables are held in memory
  - Memory addresses called *pointers*
- *Dereferencing* a pointer accesses memory contents, using "*" operator
  - example:

```
// calculate and return requested Fibonacci element:
int fibonacci(pfibo, n)
{
    int *elt = pfibo + n;     // int pointer
    *elt = *(elt-1) + *(elt-2); // dereferences
    return *elt;     // dereference
}
```

Pointers example: print cmd-line args

```
/* command-line arguments */
/* Use pointer manipulation */
#include <stdio.h>

int main(int argc, char **argv)
{
    char **pa;

    printf("program: %s\n", *argv);

    for (pa = argv + 1; pa < argv + argc; pa++)
        printf("%p  %p  %s\n", pa, *pa, *pa);

    return 0;
}
```

More examples

```c
/* strlen() implementation */
unsigned long strlen(char *ps)
{
   unsigned long len = 0;
   while (*(ps++))
      len++;
   return len;
}
```

```c
/* fputs implementation */
unsigned long fputs(char *ps, FILE *pf)
{
   unsigned n = 0;
   while (*ps) {
      fputc(*(ps++), pf);
      n++;
   }
   return n;
}
```

```c
/* create a reversed copy of a string */

unsigned long strlen(char *pstr);

void reverse(char *destination, char *source)
{
   unsigned long length = strlen(source);
   char *pd, *ps;

   for (pd = destination, ps = source + length - 1; ps >= source; pd++, ps--)
      *pd = *ps;
   // the for loop left pd pointing past the last copied char
   *pd = '\0';
}
```

more examples 2

```c
/* strlen() implementation */
unsigned long mystrlen(char *ps)
{
     char *p2 = ps;
     while ( *(p2++) != '\0' )
         ;
     // undo the final post-increment...
     return ((--p2) - ps);
}
```

- In mystrncpy() below, note the use of the *pre-increment* operator, ++n.
  - This adds one *before* testing the value, in contrast to the post-increment operator n++.

```c
/* strncpy implementation */
unsigned long mystrncpy(char *dest, char *src, unsigned long nchars)
{
   unsigned long n = 0;
   while ((++n < nchars) && *src)
      *(dest++) = *(src++);
   *dest = '\0';
   return --n;
}
```