Introduction to C

Pointers and Arrays

Instructor: Yin Lou

01/31/2011

Pointers

- A pointer is a variable that contains the address of a variable
- Pointers are powerful but dangerous as well

Pointers

- A pointer is a variable that contains the address of a variable
- Pointers are powerful but dangerous as well
  - Sometimes pointers are the only way to express the computation
  - Points usually lead to more compact and efficient code
  - But the programmer must be extremely careful

Memory

- Variables are stored in memory
- Think of memory as a very large array
  - Every location in memory has an address
  - An address is an integer, just like an array index
- In C, a memory address is called a *pointer*
  - C lets you access memory locations directly

Two Operators

- & (“address of”) operator
  - Returns the address of its argument
  - Said another way: returns a *pointer* to its argument
  - The argument must be a variable name.
- * (“dereference”) operator
  - Returns the value stored at a given memory address
  - The argument must be a pointer

Declaration

```
int i;                 // Integer i
int *p;                // Pointer to integer
int **m;               // Pointer to int pointer

p = &i;                // p now points to i
printf("%p", p);       // Prints the address of i (in p)

m = &p;                // m now points to p
printf("%p", m);       // Prints the address of p (in m)
```

Example

```
int a = 0;
int b = 0;
int *p;

a = 10;
p = &a;
*p = 20; // a = ? b = ?

p = &b;
*p = 10; // a = ? b = ?
a = *p;  // a = ? b = ?
```

Passing Pointers to Functions

```
void swap(int *a, int *b)
{
    int t = *a;
    *a = *b;
    *b = t;
}

void main()
{
    int a = 5, b = 3;
    printf("Before swap: a = %d b = %d\n", a, b);
    swap(&a, &b);
    printf("After swap: a = %d b = %d\n", a, b);
}
```

Multiple Return Values

```
void initialize(int *a, char *b)
{
    *a = 10;
    *b = ’x’;
}

void main()
{
    int a, b;
    initialize(&a, &b);
}
```

Pointers Are Dangerous

What does this code do?

```
void main()
{
    char *x;
    *x = ’a’;
}
```

Pointers Are Dangerous

What does this code do?

```
void main()
{
    char *x;
    *x = ’a’;
}
```

What about this code?

```
void main()
{
    char x = ’a’;
    char *p = &x;
    p++;
    printf("%c\n", *p);
}
```

Arrays

- To declare an array, use [], e.g:

Arrays

- To declare an array, use [], e.g:
  - `int a[5]; // Creates an array with 5 integer elements`

Arrays

- To declare an array, use [], e.g:
  - `int a[5]; // Creates an array with 5 integer elements`
- The size of an array can't be changed
- The number between the brackets must be a constant

Arrays

- To declare an array, use [], e.g:
  - `int a[5]; // Creates an array with 5 integer elements`
- The size of an array can't be changed
- The number between the brackets must be a constant
- You can give initial values for array elements, e.g:

Arrays

- To declare an array, use [], e.g:
  - `int a[5]; // Creates an array with 5 integer elements`
- The size of an array can't be changed
- The number between the brackets must be a constant
- You can give initial values for array elements, e.g:
  - int a[5] = {3, 7, -1, 4, 6};

Arrays

- To declare an array, use [], e.g:
  - `int a[5]; // Creates an array with 5 integer elements`
- The size of an array can't be changed
- The number between the brackets must be a constant
- You can give initial values for array elements, e.g:
  - int a[5] = {3, 7, -1, 4, 6};
  - A better way: int a[] = {3, 7, -1, 4, 6}; // Let the compiler calculate the size

Arrays

- Array indices in C are zero-based, e.g. a[0], a[1], ..., a[4]

Arrays

- Array indices in C are zero-based, e.g. a[0], a[1], ..., a[4]

Example

```
void main()
{
    int a[] = {3, 7, -1, 4, 6};
    int i;
    double mean = 0;

    // compute mean of values in a
    for (i = 0; i < 5; ++i)
    {
        mean += a[0];
    }
    mean /= 5;
    printf("Mean = %.2f\n", mean);
}
```

Pointers and Arrays

- Pointers and arrays are closely related

Pointers and Arrays

- Pointers and arrays are closely related
  - An array variable is actually just a pointer to the first element in the array

Pointers and Arrays

- Pointers and arrays are closely related
  - An array variable is actually just a pointer to the first element in the array
- You can access array elements using array notation or pointers

Pointers and Arrays

- Pointers and arrays are closely related
  - An array variable is actually just a pointer to the first element in the array
- You can access array elements using array notation or pointers
  - a[0] is the same as *a
  - a[1] is the same as *(a + 1)
  - a[2] is the same as *(a + 2)

Pointers and Arrays

- Accessing array elements using pointers

Example

```
void main()
{
    int a[] = {3, 7, -1, 4, 6};
    int i;
    double mean = 0;

    // compute mean of values in a
    for (i = 0; i < 5; ++i)
    {
        mean += *(a + i)
    }
    mean /= 5;
    printf("Mean = %.2f\n", mean);
}
```

Pointers and Arrays

- If pa points to a particular element of an array, (pa + 1) always points to the next *element*, (pa + i) points i elements after pa and (pa - i) points i elements before.

Pointers and Arrays

- If pa points to a particular element of an array, (pa + 1) always points to the next *element*, (pa + i) points i elements after pa and (pa - i) points i elements before.
- The only difference between an array name and a pointer:
  - A pointer is a variable, so pa = a and pa++ is legal
  - An array name is not a variable, so a = pa and a++ is illegal

Strings

- There is no string type in C!

Strings

- There is no string type in C!
- Instead, strings are implemented as arrays of characters: char * or char []
- Enclosed in double-quotes
- Terminated by NULL character ('\0')

Strings

- There is no string type in C!
- Instead, strings are implemented as arrays of characters: char * or char []
- Enclosed in double-quotes
- Terminated by NULL character ('\0')
- "Hello"
- printf format: %s
- same as
  char str[] = {'H', 'e', 'l', 'l', 'o', '\0'}

Built-in String Functions

- string.h has functions for manipulating null-terminated strings, e.g.

Built-in String Functions

- string.h has functions for manipulating null-terminated strings, e.g.
  - strlen(char *s): returns length of s
  - strcat(char *s1, char *s2): appends s2 to s1 (s1 must have enough space!)
  - strcpy(char *s1, char *s2): copies s2 into s1(Again, s1 must have enough space!)
  - strcmp(char *s1, char *s2): compares s1 and s2

Pointers, Arrays and Functions

- It's possible to pass part of an array to a function, by pass a pointer to the beginning of the subarray.

Pointers, Arrays and Functions

- It's possible to pass part of an array to a function, by pass a pointer to the beginning of the subarray.
  - f(&a[2])
  - f(a + 2)

Pointers, Arrays and Functions

- It's possible to pass part of an array to a function, by pass a pointer to the beginning of the subarray.
  - f(&a[2])
  - f(a + 2)
- Within f, the parameter declaration can read
  - f(int arr[]) { ... }
  - f(int *arr) { ... }

Example

```
int strlen(char *s)
{
    int n = 0;
    while (*s != ’\0’)
    {
        s++;
        n++;
    }
    return n;
}

char *p = "hello, world";
strlen(p);
strlen(p + 7);
```

Dynamically Allocating Arrays

Dynamically Allocating Arrays

- malloc: Allocate contiguous memory dynamically

Dynamically Allocating Arrays

- malloc: Allocate contiguous memory dynamically
  - int *p = (int *) malloc(n * sizeof(int));
  - An array of size n

Dynamically Allocating Arrays

- malloc: Allocate contiguous memory dynamically
  - int *p = (int *) malloc(n * sizeof(int));
  - An array of size n
- free: Deallocate the memory

Dynamically Allocating Arrays

- malloc: Allocate contiguous memory dynamically
  - int *p = (int *) malloc(n * sizeof(int));
  - An array of size n
- free: Deallocate the memory
  - free(p);

Dynamically Allocating Arrays

- malloc: Allocate contiguous memory dynamically
  - int *p = (int *) malloc(n * sizeof(int));
  - An array of size n
- free: Deallocate the memory
  - free(p);
- Make sure malloc and free are paired!