COMsW 1003-1

Introduction to Computer Programming in C

Lecture 12

Spring 2011

Instructor: Michele Merler

http://www1.cs.columbia.edu/~mmerler/comsw1003-1.html
Announcements

Homework 3 is out
• Due on Monday, 03/21/11 at the beginning of class, no exceptions

Midterm
• In class on Wednesday, 03/09/11
• Will cover everything up to Lecture 13 (included)
• Open books, open notes
• Closed electronic devices
Today

• Passing arguments to function by value vs. by reference (from Lec 11)

• Functions returning pointers

• Pointers of pointers
Functions Returning Pointers

• Naturally, a function can return a pointer
• This is a way to return an array, but must be careful about what has been allocated in memory

```c
returnType * functionName( parameters )
```

**NOTE**

**NULL** is the equivalent of zero for pointers
Functions Returning Pointers

Example: using pointers to return a string

Given a string of the type “firstNAme/lastName”
We want to split it into two separate entities to print
Functions Returning Pointers

POINT 1
firstName

POINT 2
firstName
lastName

POINT 3
firstName
lastName

POINT 4
firstName
lastName
When we try to declare a pointer to be a constant like this, it means that the value at the address in memory it points to cannot be modified.

This does NOT mean that the pointer is constant, it can be changed!

```c
int x = 7, y = 3;
const int *ptr = &x;
*ptr = 11; // ✗

x = 8;
ptr = &y;
*ptr = 9;

printf("x = %d, y = %d\n", x, *ptr);
```
Const pointers

When we try to declare a pointer to be a constant like this, it means that the value at the address in memory it points to cannot be modified.

This does NOT mean that the pointer is constant, it can be changed!

```c
int x = 7, y = 3;

const int *ptr = &x;

*ptr = 11;  // ✗
x = 8;  // ✓
```

Main memory

```
 ptr  0000 0000 0000 1000  x
      0000 0000 0000 0011  y
```
Const pointers

When we try to declare a pointer to be a constant like this, it means that the value at the address in memory it points to cannot be modified.

This does NOT mean that the pointer is constant, it can be changed!

```c
int x = 7, y = 3;

const int *ptr = &x;

*ptr = 11;  // ✗

x = 8;     // ✓

ptr = &y;  // ✓
```

![Diagram showing memory layout with pointers and variables]
Const pointers

When we try to declare a pointer to be a constant like this, it means that the value at the address in memory it points to cannot be modified:

This does NOT mean that the pointer is constant, it can be changed!

```
int x = 7, y = 3;
const int *ptr = &x;
*ptr = 11;  // ✗
x = 8;  // ✓
ptr = &y;  // ✓
*ptr = 9;  // ✗
printf("x = %d, y = %d\n", x, *ptr);
```
### Const pointers

This is the declaration of a constant pointer. In this case, the pointer is fixed, but the value at the address it points to can be modified.

```c
int x = 7, y = 3;
int * const ptr2 = &x;
*ptr2 = 9;  // V
ptr2++;   // X
ptr2 = &y; // X

printf("x = %d, x = %d\n", x, *ptr2);
```
Arrays of strings

• An array Arr of 3 strings of variable length

```c
```

```c
```

• Arr is an array of 3 elements. Each element in Arr is of type `pointer to char`.
Arrays of strings

• An array Arr of 3 strings of variable length

```c

```

• An array Arr of 3 strings of maximum length = 15

```c

Arr2[0]      Arr2 // "Hello2"
Arr2[1]      Arr2+1 // "World2"
```
Pointers of pointers

<table>
<thead>
<tr>
<th>Arr</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>'H'</td>
<td>'e'</td>
<td>'l'</td>
<td>'l'</td>
<td>'o'</td>
<td>'\0'</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>'W'</td>
<td>'o'</td>
<td>'r'</td>
<td>'l'</td>
<td>'d'</td>
<td>'\0'</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>'W'</td>
<td>'o'</td>
<td>'n'</td>
<td>'d'</td>
<td>'e'</td>
<td>'r'</td>
<td>'f'</td>
<td>'u'</td>
<td>'l'</td>
<td>'\0'</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Arr2</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>'H'</td>
<td>'e'</td>
<td>'l'</td>
<td>'l'</td>
<td>'o'</td>
<td>'2'</td>
<td>'\0'</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>'W'</td>
<td>'o'</td>
<td>'r'</td>
<td>'l'</td>
<td>'d'</td>
<td>'2'</td>
<td>'\0'</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>'W'</td>
<td>'o'</td>
<td>'n'</td>
<td>'d'</td>
<td>'e'</td>
<td>'r'</td>
<td>'f'</td>
<td>'u'</td>
<td>'l'</td>
<td>'2'</td>
</tr>
</tbody>
</table>

stringArrays.c
Pointers of pointers

- A pointer can point to another pointer
- In a sense, it’s the equivalent of matrices!

```c
int x = 3;

int *p = &x;

int **p2 = &p;

x = 2;  *p = 2;  **p2 = 2;


char **ptr;

ptr = Arr;
```
Pointers of pointers

```
0 1 2 3 4 5 6 7 8 9
Arr
ptr
0 ‘H’ ‘e’ ‘l’ ‘l’ ‘o’ ‘\0’
ptr + 1
1 ‘W’ ‘o’ ‘r’ ‘l’ ‘d’ ‘\0’
2 ‘W’ ‘o’ ‘n’ ‘d’ ‘e’ ‘r’ ‘f’ ‘u’ ‘l’ ‘\0’
```

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
Arr2
0 ‘H’ ‘e’ ‘l’ ‘l’ ‘o’ ‘2’ ‘\0’
1 ‘W’ ‘o’ ‘r’ ‘l’ ‘d’ ‘2’ ‘\0’
2 ‘W’ ‘o’ ‘n’ ‘d’ ‘e’ ‘r’ ‘f’ ‘u’ ‘l’ ‘2’ ‘\0’
```

C stringArrays.c
Pointers of pointers

```c
char **ptr;
ptr = Arr;
```

```c
*((*(ptr+1)+2)
```
Pointers of pointers

```c
char **ptr;
ptr = Arr;

*(*(ptr+1)+2)
```

1. `ptr+1`
   - `ptr+1` points to the whole line

2. `*(ptr+1)`
   - `*(ptr+1)` points to the first element of the line
Pointers of pointers

\[
\text{char } *\text{Arr}[3]=\{ "Hello", "World", "Wonderful" \};
\]
\[
\text{char } **\text{ptr};
\]
\[
\text{ptr }= \text{Arr};
\]

\[
\ast(\ast(\text{ptr}+1)+2)
\]

3. \(\ast(\text{ptr}+1)+2\)

\[
\ast(\text{ptr}+1)+2 \text{ points to the third element of the line}
\]

\[
\ast(\ast(\text{ptr}+1)+2)
\]

2. \(\ast(\ast(\text{ptr}+1)+2)\)

Now we get the value stored at the address we point
Pointers of pointers

```c
char **ptr;
ptr = Arr;
```

Avoid this notation!
`ptr[1][2]` is much better!
**Pointers vs. Arrays**

<table>
<thead>
<tr>
<th>Arrays</th>
<th>Pointers</th>
</tr>
</thead>
<tbody>
<tr>
<td>1D array of 5 int</td>
<td>int *xPtr;</td>
</tr>
<tr>
<td>2D array of 6 int</td>
<td>int **yPtr;</td>
</tr>
<tr>
<td>2x3 matrix</td>
<td>int**zPtr;</td>
</tr>
<tr>
<td>2D array of 4 int</td>
<td>int**zPtr;</td>
</tr>
<tr>
<td>2x2 matrix</td>
<td>int**zPtr;</td>
</tr>
<tr>
<td>1D array of 5 char</td>
<td>char *cPtr;</td>
</tr>
<tr>
<td>string</td>
<td>char *cPtr;</td>
</tr>
</tbody>
</table>

- **Arrays**
  - 1D array of 5 int: `int x[5];`
  - 2D array of 6 int: `int y[2][3];`
  - 2D array of 4 int: `int *z[2] = {{1, 2}, {2, 1}};`
  - 1D array of 5 char: `char c[] = “mike”;`

- **Pointers**
  - Pointer to an array of 5 int: `int *xPtr;`
  - Pointer to a 2D array of 6 int: `int **yPtr;`
  - Pointer to a 2D array of 4 int: `int **zPtr;`
  - Pointer to a 1D array of 5 char: `char *cPtr;`

---

**Space has been allocated in memory for the arrays**

**Space has been allocated in memory only for the pointers variables, NOT for the arrays they will point to.**

*The DIMENSIONS of the arrays are UNKNOWN*
Multidimensional Arrays

2x3 matrix of double

```c
double M0[2][3];
double *M1[2] = M0;
double **M = M0;
```

![Diagram of multidimensional array access]

double ** double * double
Multidimensional Arrays

2x3 matrix of double

double M0[2][3];
double *M1[2] = M0;
double **M = M0;

The difference between M0, M1 and M is that M1 and M can have ANY SIZE!