Pointers
 Lecture 2
 Section 2.4

Robb T. Koether

Hampden-Sydney College

Fri, Jan 16, 2009
Outline

1. Introduction

2. Pointer Arithmetic
   - Pointer + int
   - Pointer - Pointer

3. Pointers and Arrays

4. Pointers as Function Parameters

5. Assignment
Big Endian and Little Endian

- Suppose we make the assignment
  \[
  \text{int } i = 0x25116634;
  \]

- If the architecture is **Big Endian**, then it is stored as
  "Big" end first → 25 11 66 34

- If the architecture is **Little Endian**, then it is stored as
  "Little" end first → 34 66 11 25
Big Endian and Little Endian

Definition (Big Endian, Little Endian)

If a processor has **Big Endian** architecture, then it stores the highest-order byte of the value in the lowest byte address. If a processor has **Little Endian** architecture, then it stores the lowest-order byte of the value in the lowest byte address.

- Multi-byte objects are addressed by their lowest byte address.
It is permissible to add an integer to a pointer.

\[
\begin{align*}
\texttt{int} & \quad i = 123; \\
\texttt{int*} & \quad p = & i; \\
\texttt{int*} & \quad q = p + 1;
\end{align*}
\]

- The integer is multiplied by the size of the object pointed to.
- That value is added to the address.
- The result is a pointer of the same type.
The integer is interpreted as the number of objects of that type between the original address and the new address.

The resulting address is interpreted as a pointer to the object located the specified number of objects to the right, as in an array.
**Introduction**

**Pointer Arithmetic**

```c
int i = 123;       // Addr of i is 0x0100
int* p = &i;       // p = 0x0100
char* q = (char*)&i; // q = 0x0100
short* r = (short*)&i; // r = 0x0100
```
**Introduction**

**Pointer Arithmetic**

```c
int i = 123; // Addr of i is 0x0100
int* p = &i; // p = 0x0100
char* q = (char*)&i; // q = 0x0100
short* r = (short*)&i; // r = 0x0100
```

```
0x00fc (i)
0x0100 0x0104 0x0108
```

```
p - 1  p   p + 1  p + 2
```

- p - 1 points to 0x00fc
- p points to 0x0100
- p + 1 points to 0x0104
- p + 2 points to 0x0108
int i = 123; // Addr of i is 0x0100
int* p = &i; // p = 0x0100
char* q = (char*)&i; // q = 0x0100
short* r = (short*)&i; // r = 0x0100

q - 1
q
q + 1
q + 2

0x00fc 0x0100 0x0104 0x0108
int i = 123; // Addr of i is 0x0100
int* p = &i; // p = 0x0100
char* q = (char*)&i; // q = 0x0100
short* r = (short*)&i; // r = 0x0100

r - 1 r r + 1 r + 2

0x00fc 0x0100 0x0104 0x0108
Demo - Pointer Arithmetic

Demo - Demo - PtrPlusPtr.cpp

Run PtrPlusPtr.cpp.
Demo - Endianness.cpp
Run Endianness.cpp.
Pointer Subtraction

- Subtraction of one pointer from another pointer (of the same type) is permitted.
  - Take the difference of the addresses.
  - Divide it by the size of the object pointed to.
  - The result is an `int`.
- Interpret the result as the number of objects of that type between two addresses.
Example (Pointer Subtraction)

```c
int* p;
int* q;
int a = q - p;
```
**Example (Pointer Subtraction)**

```c
int* p;
int* q;
int a = q - p;
```

\[ p \quad q - p = 5 \quad q \]
Demo - `PtrMinusPtr.cpp`

Run `PtrMinusPtr.cpp`. 
**Pointer Arithmetic**

**Example**

```
int a = (p2 - p1) + 5;
int b = p2 - (p1 - 5);
int c = (p2 + 5) - p1;
```

- These are legal operations.
- Are they equivalent?
Example (Illegal Addition)

```c
int* mid = (p2 + p1)/2; // Midpt b/t p1, p2
```

- Pointer addition is illegal.
- How can we obtain a pointer to the object halfway between `p1` and `p2`?
A pointer may point to a constant - the object pointed to cannot be changed.

A pointer itself may be constant - the pointer cannot be changed.

In fact, a constant pointer may point to a constant object!
An array name represents a pointer that points to the first member of the array.

The array name is a constant pointer; its value cannot be changed.

An array name may be assigned to a pointer of the same type.
A pointer name may be indexed.

\[ p[i] \text{ is equivalent to } *(p + i) \].
Demo - `PointerLoop.cpp`

Run `PointerLoop.cpp`. 
Pointers as Function Parameters

- In CS I, we learn to pass objects to function *by reference*.
- The intention is to allow the object to be modified by the function.
- Before pass-by-reference was introduced, the same was accomplished by passing a pointer.
- To access the object form within the function, the pointer parameter must be dereferenced.
Example (Pointer Parameter)

```c
void swap(int* a, int* b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
    return;
}
```
Demo

**Demo - IntegerInput.cpp**

**Run IntegerInput.cpp.**
Arrays as Function Parameters

- When an array is “passed” as a parameter, the name of the array is passed, as a pointer.
- Thus, the function receives a pointer to the first element of the array.
- This is far more efficient than copying the entire array.
Example

Example (Array Parameter)

```c
void Sort(int list[], int size)
{
    for (int i = 0; i < size - 1; i++)
    {
        int* q = list;
        while (q < list + size - 1)
        {
            if (*q > *(q + 1))
                Swap(q, q + 1);
            q++;
        }
    }
    return;
}
```
Assignment

Homework

- Read Section 2.4, pages 63 - 71.
- Read Section 3.2, pages 85 - 97.