Pointers and Arrays

Based on slides from K. N. King and Dianna Xu
Bryn Mawr College
CS246 Programming Paradigm

The NULL Pointer

- C guarantees that zero is never a valid address for data
- A pointer that contains the address zero known as the NULL pointer
- It is often used as a signal for abnormal or terminal event
- It is also used as an initialization value for pointers

Pass by Value

- All functions are pass-by-value in C
  - A copy is made of each parameter’s value and then the copy is passed
  - Variables supplied as parameters to a function call are protected against change
    - i.e. impossible to write a swap(x, y) function
- Only way to modify a variable through a function is to assign the return value to that variable

Pass by Value and Pointers

- All functions are pass-by-value in C
- Pass-by-value still holds even if the parameter is a pointer
  - A copy of the pointer’s value is made – the address stored in the pointer variable
  - The copy is then a pointer pointing to the same object as the original parameter
  - Thus modifications via de-referencing the copy STAYS.

Function Arguments

- x and y are copies of the original, and thus a and b can not be altered.

    ```c
    void swap(int x, int y) {
        int tmp;
        tmp = x; x = y; y = tmp;
    }
    int main() {
        int a = 1, b = 2;
        swap(a, b);
        return 0;
    }
    ```

Pointers as Function Arguments

- Passing pointers – a and b are passed by reference (the pointers themselves px and py are still passed by value)

    ```c
    void swap(int *px, int *py) {
        int tmp;
        tmp = *px; *px = *py; *py = tmp;
    }
    int main() {
        int a = 1, b = 2;
        swap(&a, &b);
        return 0;
    }
    ```
Pointers as Function Arguments

• Write a function that will decompose a double value into an integer part and a fractional part.
• As a result of the call, int_part points to i and frac_part points to d:

```
    x  3.14159
   int_part  3
    frac_part .14159 d
```

```
void decompose(double d, int *i, double *frac) {
    *i = (int) d;
    *frac = d - *i;
}
```

```
int main() {
    int int_part;
    double frac_part, input;
    scanf("%lf", &input);
    decompose(input, &int_part, &frac_part);
    printf("%f decomposes to %d and %f\n", *frac_part);
    return 0;
}
```

Pass by Reference

• The pointer variables themselves are still passed by value
• In a function, if a pointer argument is dereferenced, then the modification indirectly through the pointer will stay

```
void f(int *px, int *py) {
    px = py;
}
```

```
int main() {
    int x = 1, y = 2, *px;
    px = &x;
    f(px, &y);
    printf("%d", *px);
}
```

Modification of a Pointer

```
void g(int **ppx, int *py) {
    *ppx = py;
}
```

```
int main() {
    int x = 1, y = 2, *px;
    px = &x;
    g(&px, &y);
    printf("%d", *px);
}
```

Pointers are Passed by Value

```
void f(int *px, int *py) {
    px = py;
}
```

```
int main() {
    int x = 1, y = 2, *px;
    px = &x;
    f(px, &y);
    printf("%d", *px);
}
```

Pointer as Return Value

• We can also write functions that return a pointer
• Thus, the function is returning the memory address of where the value is stored instead of the value itself
• Be very careful not to return an address to a temporary variable in a function!!!
Example

- \( x \) and \( y \) are copies of the original, and thus what is \( &x \) and \( &y \)?

```c
int max(int *x, int *y) {
    if (*x > *y)
        return x;
    return y;
}

int main() {
    int a = 1, b = 2, *p;
    p = max(&a, &b);
    return 0;
}
```

Arrays

- Declaration – `int a[5];` a[0][1][2][3][4]
- Assignment – `a[0] = 1;`
- Reference – `y = a[0];` a[1][2][3][4]
- Schematic representation

Name of an Array

- The variable name of an array is also a **pointer** to its first element.

```c
int a[5];
```

```c
a[0] == *a
a[] == &a[0]
```

Pointer Arithmetic

- One can add/subtract an integer to/from a pointer
- The pointer advances/retracts by that number of elements (of the type being pointed to)
  - \( a[i] \) == \&a[i]
  - \( a[i] \) == *(a+i)
- Subtracting two pointers yields the number of elements between them

Adding an Integer to a Pointer

- Example of pointer addition:
  ```c
  p = &a[2];
  ```
  ```c
  q = p + 3;
  ```
  ```c
  p += 6;
  ```
  If \( p \) points to the array element \( a[1] \), then \( p + j \) points to \( a[j] \).
Subtracting an Integer from a Pointer

• If \( p \) points to \( a[i] \), then \( p - j \) points to \( a[i-j] \).

• Example:

\[
\begin{align*}
p &= &a[8]; \\
q &= p - 3; \\
p &=& 6;
\end{align*}
\]

Subtracting One Pointer from Another

• When one pointer is subtracted from another, the result is the distance (measured in array elements) between the pointers.

• If \( p \) points to \( a[i] \) and \( q \) points to \( a[j] \), then \( p - q \) is equal to \( i - j \).

• Example:

\[
\begin{align*}
p &= &a[5]; \\
q &= a[1]; \\
i &= p - q; \\
\text{/* } i \text{ is 4 */} \\
i &= q - p; \\
\text{/* } i \text{ is -4 */}
\end{align*}
\]

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 \( a \):

\[
\begin{align*}
\text{#define } N &\text{ 10} \\
\text{int } a[N], &\text{ sum, } p; \\
\text{sum} &= 0; \\
\text{for } &\text{ (} p = a[0]; p < &a[N]; p++) \\
\text{sum} &= &p; \\
\end{align*}
\]

• \&\( a[N] \) is legal since the loop doesn’t attempt to examine its value.

Combining * and ++/--

• The most common combination of * and ++ is \( *p++ \), which is handy in loops.

• Instead of writing

\[
\text{for } \text{(} p = \&a[0]; p < \&a[N]; p++) \\
\text{sum} = &p;
\]

• to sum the elements of the array \( a \), we could write

\[
\begin{align*}
p &= &a[0]; \\
\text{while } &\text{(} p < \&a[N]) \\
\text{sum} &= &p++;
\end{align*}
\]

Using an Array Name as a Pointer

• The name of an array can be used as a pointer to the first element in the array.

• Suppose that \( a \) is declared as follows:

\[
\text{int } a[10];
\]

• Examples of using \( a \) as a pointer:

\[
\begin{align*}
\text{\( a = 7; \)} &\text{ /* stores } 7 \text{ in } a[0] */ \\
\text{\( (a+1) = 12; \)} &\text{ /* stores 12 in } a[1] */ \\
\end{align*}
\]

• 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:
  ```c
  for (p = &a[0]; p < &a[N]; p++)
  sum += *p;
  ```

• Simplified version:
  ```c
  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:
  ```c
  while (*a != 0)
  a++;           /*** WRONG ***/
  ```

• This is no great loss; we can always copy `a` into a pointer variable, then change the pointer variable:
  ```c
  p = a;
  while (*p != 0)
  p++;
  ```

Arrays as Arguments

• Arrays are passed by reference

• Modifications stay

```c
#define SIZE 10
void init(int a[]) {
  int i;
  for(i = 0;i<SIZE;i++)
  *(a+i) = 0;
}

int main() {
  int a[SIZE];
  init(a);
  return 0;
}
```

Consequence of Array Arguments

• 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. An array used as an argument is NOT protected against change.

```c
void store_zeros(int a[], int n) {
  int i;
  for (i = 0; i < n; i++)
  a[i] = 0;
}
```

Consequence of Array Arguments

• To indicate that an array parameter won’t be changed, we can include the word `const` in its declaration:

```c
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`. 
Consequence of Array Arguments

• Consequence 2: The time required to pass an array to a function does not depend on the size of the array.
• 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) {
      ...
  }

Consequence of Array Arguments

• Declaring a parameter to be an array is the same as declaring it to be a pointer.
• However, it is NOT same for a variable.
  int a[10];
  The compiler to set aside space for 10 integers
  int *a;
  The compiler to allocate space for a pointer variable.
  a is not an array; attempting to use it as an array can have disastrous results.

Consequence of 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);

Using a Pointer as an Array Name

• C allows us to subscript a pointer as though it were an array name:
  #define N 10
  ...
  int a[N], i, sum = 0, *p = a;
  for (i = 0; i < N; i++)
      sum += p[i];
  The compiler treats p[i] as *(p+i).

Multi-Dimensional Array

int a[2][3];

• Layout of an array with r rows:
  row 0  row 1  row r - 1

• 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 Multi-Dimensional Array

int a[NUM_ROWS][NUM_COLS];

• Use nested for loops:
  
  ```c
  int row, col;
  for (row = 0; row < NUM_ROWS; row++)
    for (col = 0; col < NUM_COLS; col++)
      a[row][col] = 0;
  ```

• If we view `a` as a one-dimensional array of integers, a single loop is sufficient:
  
  ```c
  int *p;
  for (p = &a[0][0];
       p < &a[NUM_ROWS-1][NUM_COLS-1]; p++)
    *p = 0;
  ```

Processing the Rows of a Multi-Dimensional Array

• A loop that clears row `i` of the array `a`:
  
  ```c
  int a[NUM_ROWS][NUM_COLS], *p, i;
  for (p = a[i]; p < a[i] + NUM_COLS; p++)
    *p = 0;
  ```

• Use `find_largest` to determine the largest element in row `i` of the two-dimensional array `a`:
  
  ```c
  int largest = find_largest(a[i], NUM_COLS);
  ```

Processing the Columns of a Multi-Dimensional Array

• A loop that clears column `i` of the array `a`:
  
  ```c
  int a[NUM_ROWS][NUM_COLS], (*p)[NUM_COLS], i;
  for (p = &a[0]; p < &a[NUM_ROWS]; p++)
    (*p)[i] = 0;
  ```

Using the Name of a Multidimensional Array as a Pointer

• The name of any array can be used as a pointer, regardless of how many dimensions it has, but some care is required.

• Example:
  
  ```c
  int a[NUM_ROWS][NUM_COLS];
  a is not a pointer to a[0][0];
  instead, it's a pointer to a[0].
  ```

• C regards `a` as a one-dimensional array whose elements are one-dimensional arrays.

• When used as a pointer, `a` has type `int (*) [NUM_COLS]` (pointer to an integer array of length `NUM_COLS`).

Using the Name of a Multidimensional Array as a Pointer

• Since `a` points to `a[0]`, we can simplify loops that process the elements of a two-dimensional array.

• To clear column `i` of the array `a`:
  
  ```c
  for (p = a[i]; p < a[i][NUM_COLS]; p++)
    (*p)[i] = 0;
  ```

• Now we can write
  
  ```c
  for (p = a; p < a + NUM_ROWS; p++)
    (*p)[i] = 0;
  ```
Using the Name of a Multidimensional Array as a Pointer

- We can "trick" a function into thinking that a multidimensional array is really one-dimensional.
- A first attempt at using find_largest to find the largest element in a:
  
  ```
  largest = find_largest(a, NUM_ROWS * NUM_COLS);
  /* WRONG */
  ```
  This is an error, because the type of `a` is `int(*)[NUM_COLS]` but `find_largest` is expecting an argument of type `int*`.
- The correct call:
  
  ```
  largest = find_largest(a[0], NUM_ROWS * NUM_COLS);
  a[0] points to element 0 in row 0, and it has type `int*` (after conversion by the compiler).
  ```

Summary

- Understand the relationship between arrays and pointers
- Understand the relationship between two-dimensional arrays and pointer arrays
- Arrays are passed by reference to functions
- Pointer arithmetic is powerful but dangerous!

Exercise

- Suppose that the following declarations are in effect:
  
  ```
  int a[] = {5, 15, 34, 54, 14, 2, 52, 72};
  int *p = &a[1], *q=&a[5];
  ```
  (a) what is the value of `*(p+3)`?
  (b) what is the value of `*(q-3)`?
  (c) what is the value of `q-p`?
  (d) Is `p<q` true or false?
  (e) Is `*p<*q` true or false?

Exercise

- What will be the contents of the array after the following statements are executed?
  
  ```
  #define N 10
  int a[N] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
  int *p=&a[0], *q=&a[N-1], temp;
  while(p<q){
    temp = *p;
    *p++ = *q;
    *q-- = temp;
  }
  ```