Pointers and Structures in C
Chapter 4

Overview

• **Pointers**
  • Variables that hold memory addresses
  • Using pointers to do “call-by-reference” in C

• **Struct**
  • Multiple values grouped together
  • Dereferencing to access individual elements
Exercise: Write a swap function

```c
void swap(int x, int y) {
    __________
    __________
    __________
    __________
}

int main(void) {
    int a = 11, b = 22;
    swap(a, b);
    printf("a=%d, b=%d\n");
}
```

Output should be
a=22, b=11

Function Parameters

- C always passes arguments BY-VALUE
  - Memory allocated for parameters when function is called
  - Argument VALUES copied into newly allocated memory
  - Memory automatically de-allocated when function ends

```c
void swap(int x, int y) {
    __________
    __________
    __________
    __________
}

int main(void) {
    int a = 11, b = 22;
    swap(a, b);
    printf("a=%d, b=%d\n");
}
```
What are Pointers?

• A pointer is a variable that holds the ADDRESS of another variable

• Suppose that we have an integer variable

  ```
  int i;
  ```

  and wish to have a pointer point to this variable.

• How do we know where \( i \) is located?

  ```
  &i
  ```

  is the address of \( i \). The operator \( & \) is called the

  ADDRESS-OF operator.
Pointers and Addresses

- We can declare that a pointer `iPtr` points to an `int` by saying

  ```
  int * iPtr;
  ```

- Suppose that we have:

  ```
  int i = 5;
  int j = 7;
  ```

- We can make `iPtr` point to `i` by assigning to `iPtr` the memory location where `i` is stored. Thus

  ```
  iPtr = &i;
  ```

  sets `iPtr` to point to `i`.

Declaring Pointers

- When declaring several pointer variables in one statement - the asterisk does not distribute throughout the statement:

  ```
  int *p, q;
  ```

  equivalent to

  ```
  int * p;
  int q;
  ```

  ```
  int * p, * q;
  ```

  equivalent to

  ```
  int * p;
  int * q;
  ```
Initializing Pointers

• We can also initialize \( iPtr \) at the point of declaration:

\[
\begin{align*}
\text{int } i; \\
\text{int } * iPtr = &i;
\end{align*}
\]

• Here is a common error:

\[
\begin{align*}
\text{int } i; \\
\text{int } * iPtr = i; & // \text{ERROR: } i \text{ is not an address}
\end{align*}
\]

Dereference *

• The value of the data being pointed at is obtained by using the operator

• If \( p \) is a pointer value, then

\[
*p
\]

refers to the variable pointed to by \( p \). Since reference is another name for address, the operator * is called dereference operator.
Dereference Example

- A dereferenced pointer behaves exactly like the variable it points to.

Example Program

```c
#include <stdio.h>
int main(void) {
    char x = 'M';
    char* p = &x;
    printf("Value of x is %c\n", x);
    printf("Address of x is %u\n", p);
    printf("Address of p is %u\n," &p);
    return 0;
}
```

- Output
  - Value of x is M
  - Address of x is 4290770463
  - Address of p is 4290770456

What is this? *p
swap Function Revisited

```c
void swap(int *x, int *y)
{
    ______________
    ______________
    ______________
    ______________
}

int main(void) {
    int a = 11, b = 22;
    swap(__, __);
    printf("a=%d, b=%d\n");
}
```

Output should be: \texttt{a=22, b=11}

Note the Difference ...
Uninitialized Pointers

• Suppose that we have the following declarations:

\[
\begin{align*}
\text{int } i; \\
\text{int } * \text{iPtr; } \\
* \text{iPtr } = 100;
\end{align*}
\]

• What is the value of \text{iPtr? Undefined. What could happen?}
  • \text{iPtr} could hold an address that does not make sense at all, causing your program to crash if dereferenced.

  • \text{iPtr} could point to an address which is accessible. Then the assignment \text{*iPtr = 100;} would accidentally change some other data, which could result in a crash at a later point. This is a tough error to detect since the cause and symptom may be widely separated in time.

Putting it all Together...

\[
\begin{align*}
\text{int } i, \text{value;} \\
\text{int } * \text{iPtr;} // \text{declares } \text{iPtr to be a pointer to an integer} \\
i = 510; & /* \text{Step 1 */} \\
i\text{Ptr } = & \text{i; } & /* \text{Step 2 */} \\
\text{value } = \text{*iPtr; } & /* \text{Step 3 */}
\end{align*}
\]

\[
\begin{array}{ccc}
\text{iPtr} & \text{i} & \text{value} \\
\text{After Step1:} & \boxed{} & \boxed{} & \boxed{} \\
\text{After Step2:} & \boxed{} & \boxed{} & \boxed{} \\
\text{After Step3:} & \boxed{} & \boxed{} & \boxed{}
\end{array}
\]
The null Pointer

- The value of a pointer can be:
  - some garbage (pointer unassigned)
  - the address of some variable (e.g., `int * p = &i;`)
  - the constant 0 (the null pointer, points to absolutely nothing)

  ```
  somePointer = 0;
  ```

  *This statement does not cause somePointer to point to memory location zero; it guarantees that somePointer does not point to anything.*

- The null pointer is a special pointer value that a program can test for:

  ```
  if (somePointer == 0) ...
  ```

Arrays vs. Pointers
Arrays and Pointers

• An array name is basically a *constant* pointer

• Consider the declaration:

  ```
  int a[3];
  ```

  The compiler allocates three integers for the array object. These are referenced as `a[0]`, `a[1]`, `a[2]` and occupy a *contiguous* block of memory.

• The value of `a` is exactly `&a[0]`, the address of the first integer in the array

Arrays and Pointers - Examples

• Consider the following declarations:

  ```
  int a[5] = {1, 2, 3};
  int * p;
  ```

  ```
  p = &a[2];
  ```

  You can use the index `[]` operator with a pointer:

  ```
  p[0] = 17;
  p[1] = 23;
  ```

  Indexing can be used with *any* pointer, but it only makes sense when the pointer points to an array.
Arrays are NOT Pointers

• Declaring an array sets aside space for its elements

```c
char a[5];
```

• Declaring a pointer variable sets aside only space to hold the variable

```c
char * p;
```

• You can change a pointer variable, but not the address of an array

```c
char b[6];
p = b;    // OK
b = p;    // ERROR!
```

Indexing Pointers

```c
int a[5];
int *p, *q;
p = a;
p[1] = 44;
q = p + 2;
q[-1] = 43;
q[2] = 46;
```
Pointer Arithmetic

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

Subscript: \( a[i] \) “means” \( *(a+i) \)

```c
int *p;
p = a + 2;
```

Note: arithmetic scales by data size (e.g., int of 4 bytes)

Quaint usage of pointer arithmetic

Add up the elements of an array:

```c
... for (p=a; p<a+100; p++)
    sum += *p;
```

More straightforwardly:

```c
... for (i=0; i<100; i++)
    sum += a[i];
```
Array Parameters to Functions

```c
void printArray(int *p, int n) {
    int i;
    for (i=0; i<n; i++)
        printf("%d\n",p[i]);
}
int fib[5] = {1, 1, 2, 3, 5};
int main(void) {
    printArray(fib, 5);
}
```

Array Params $\equiv$ Pointer Params

```c
void printArray(int *p, int n) { ... }  
void printArray(int p[5], int n) { ... }  
void printArray(int p[], int n) { ... }  
void printArray(int p[1000], int n) { ... }  

All these declarations are equivalent!

int main(void) {
    printArray(fib, 5);
}
```
Exercise: Reverse Array

• Reverse the values in an array
  • Inputs: integer array \( a \), and number of elements \( n \)
  • Output: values of \( a \) stored in reverse order

• Algorithm
  • Swap the first and last elements in the array
  • Swap the second and second-to-last elements
  • ...

Review: Strings

A string is just an array of characters (pointer to character), terminated by a ‘\0’ char (a null, ASCII code 0).

```
char mystring[6] = {'H','e','l','l','o','\0'};
char mystring[] = “Hello”;
```

```
mystring Hello \0
```

```
char *yourstring = “Hello”;
```

```
yourstring Hello \\
```
String and Pointer Manipulation

char mystring[] = "Hello";
char *yourstring = "Hello";

mystring = J e l l y \0
yourstring = C e l l o \0

mystring[0] = 'J';
yourstring[0] = 'C';
yourstring = mystring;
yourstring[4] = 'y';
mystring = yourstring;

Computing the Length of a String

int strlen(char* s)
{
    char* p = s;
    while (*p)
        p++;
    return (p - s);
}
Pointers and Strings

```c
void f(char *s) {
    char *p = s;
    while (*s)
        s++;
    for (s--; s>p; s--,p++) {
        char c = *s;
        *s = *p;
        *p = c;
    }
}
```

• What does this function do?

Boxes and Arrows

DRAW diagrams! They really help.

Example: you want an array of strings:

```c
char *query[4] = {
    "zero", "one", "two", NULL};
```

how to parse it:  *(query[4])

postfix operators bind tighter than prefix; whenever you’re not sure, just put the parentheses in
Structures

A struct value is a bunch of values glued together

```c
struct pair {
    int number;
    char grade;
};
```

A struct variable is a box holding a struct value

```c
struct pair x;
x.number = 217;
x.grade = 'A';
```
Pointers to structs

```
struct pair {int number; char grade;};
struct pair x;  x.number=217;  x.grade='A';
```

```
struct pair *p;
p = &x;
```

```
int n = (*p).number;
char g = (*p).grade;
```

Dereferencing Fields

```
struct pair {int number; char grade;}  *p;
```

```
int n = (p->number);
char g = (p->grade);
```
Summary

• C variables
  • Pointers
  • Struct

• Required Readings
  • Chapter 4 of the textbook