The C Programming Language
Chapter 4

(material from Dr. Michael Lewis, William & Mary Computer Science)

Overview

- Pointers and Addresses
- Swap Function
- Pointer Arithmetic
- Pointers and Arrays and Strings
- Function Pointers
- Dynamic Memory Allocation

Python vs. C vs. C++ vs. Java

	Python	C	C++	Java
pointer		type *	type *	
address of `var`		`&var`	`&var`	
memory allocation		`malloc()`, `calloc()`	`new`	same as C++
memory deallocation		`free()`	`delete`	same as C++

Indirection

- ability to refer to an object indirectly through some other mechanism than the object itself
- suppose we wish to sort large, heavy rocks in order of ascending weight; the swaps of adjacent rocks would be onerous work, so instead we will use indirection
  - label the rocks (in no particular order) 1 to n
  - for each rock, use a piece of paper to record the rock's weight and label
  - sort the pieces of paper so they are sorted in order of increasing weight
  - after sorting, look up the rocks using the labels and assemble them in sorted order
- using labels rather than the rocks themselves for sorting is an example of indirection
  - it allows us to move the heavy rocks exactly one time each

Pointers and Addresses

- one of C's strengths is the ability manipulate objects using indirection via their addresses in memory
- pointer: variable that can hold a memory address
- declaration

```
type *var;
```

  where type is the type we're pointing to
- examples
- the * modifies the variable, not the type

```
int* p, q;
int *p, q;    // equivalent declaration
```

  - p is a pointer to an int; q is an int
- use & to get the address of a variable

Pointers and Addresses

- example

```
cat -n ch04/pt_intro.c
```

```c
#include <stdio.h>

int main(void)
{
  int n = 42;
  int *p;
  int *q;

  printf("n = %d\n", n);

  p = &n;  /* Set p to point at n. */
  printf("address of n: %p\n", (void*) p);

  printf("p dereferences to: %d\n", *p);  /* Display the integer value p points to. */

  *p = 54;  /* Since p points to n, this changes the value of n! */

  printf("n = %d\n", n);

  q = &n;   /* Both p and q point to n. */
  *q = 42;  /* Change n through q. */

  printf("n = %d\n", n);

  return 0;
}
```

```
./a.out
```

```
n = 42
address of n: 0x7fff3e2ddf18
p dereferences to: 42
n = 54
n = 42
```

Pointers and Addresses

- line 11 sets p to the address of n
  - p now points to n
- line 12 prints the value of the pointer using %p
  - address begins with 0x, indicating hexadecimal (base 16)
- access the value of n through p by dereferencing
  - *p means the value at the address p points to
    - go to the location in memory stored in p
    - read one int’s worth of bits (32 bits)
    - interpret those bits as int
- line 16 writes to location p, thereby changing n
- multiple pointers can point to the same location (e.g., lines 20-21)

Pointers and Addresses

- pointers can be confusing
- pointer = location in memory
- overloaded *
  - in declaration, * means the variable is a pointer
  - for dereferencing, * means the value stored in location p

```
double x = 54;   /* A double. */
double *p = &x  /* A pointer to x. */
double y;        /* Another double. */

y = *p;   /* Equivalent to y = x. */
*p = 42;  /* Equivalent to x = 42. */

p = 42;   /* Sets p to point to the address 42, which is probably invalid! */
```

Pointers and Addresses

- **recall references from Python**
  - **two variables that refer to the same object**

```
cat -n ch04/reference.py
```

```
a = [1, 2, 3, 4]
b = a

print("b:", b)

a[0] = 42

print("b:", b)
```

```
python ch04/reference.py
```

```
b: [1, 2, 3, 4]
b: [42, 2, 3, 4]
```

- **references are like pointers, but without the need to dereference a pointer**
- **in practice, references are implemented with pointers that point to the same object in memory**

Swap Function

- swap the values of two integers
- good example of using pointers

```
cat -n ch02/bad_swap.c
```

```c
#include <stdio.h>

void swap(int m, int n)
{
    int temp;

    temp = m;
    m = n;
    n = temp;
}

int main(void)
{
    int m = 42, n = 54;

    printf("before the call to swap(): m = %d, n = %d\n", m, n);

    swap(m, n);  /* Only copies of m, n are passed. */

    printf("after the call to swap():  m = %d, n = %d\n", m, n);

    return 0;
}
```

```
gcc ch02/bad_swap.c
```

```
./a.out
```

```
before the call to swap(): m = 42, n = 54
after the call to swap():  m = 42, n = 54
```

Swap Function

- C uses call by value, so swap function doesn't work
- the solution is to pass addresses of x and y and swap the values found at those addresses
  - we are still passing by value, but this time it's the copy of a pointer
- using pointers allows functions to change values of variables
  - these changes persist after returning from the function

Swap Function

```
cat ch04/swap.c
```

```
#include <stdio.h>

void swap(int *m, int *n)
{
    int temp;

    temp = *m;  /* Set temp = the value that x points to. */
    *m = *n;    /* Set the int that x points to = the int that y points to. */
    *n = temp;  /* Set the location y points to = the value in temp. */
}

int main(void)
{
    int m = 42, n = 54;

    printf("before the call to swap(): m = %d, n = %d\n", m, n);

    swap(&m, &n);  /* Copies of the addresses are passed. */

    printf("after the call to swap():  m = %d, n = %d\n", m, n);

    return 0;
}
```

```
gcc ch04/swap.c
```

```
./a.out
```

```
before the call to swap(): m = 42, n = 54
after the call to swap():  m = 54, n = 42
```

Pointer Arithmetic

- if p is a pointer to an int, p + i refers to a memory location i integers past p
  - in terms of bytes, p + i = p + i * sizeof (int)   or p + i * 4
- pointer arithmetic depends on the type
  - if p is a pointer to short int's (size 2 bytes)
    - p + i = p + i * sizeof (short)   or p + i * 2
  - similar for char (1 byte), float (4 bytes), and double (8 bytes)
  - also works for struct's
- usually we don't care about the actual value of pointers, but we do care about the relative locations of pointers and how far apart they are

Pointer Arithmetic

```
cat -n ch04/pt_arith.c
```

```
#include <stdio.h>

int main(void)
{
  int n = 42;
  int *p;

  p = &n;

  printf("%p\n", p);
  printf("%p\n", p + 1);

  printf("Number of ints  between %p and %p: %ld\n", p, (p + 1), (p + 1) - p);
  printf("Number of bytes between %p and %p: %ld\n", p, (p + 1), ((p + 1) - p) * sizeof(int));

  return 0;
}
```

```
gcc ch04/pt_arith.c
```

```
./a.out
```

```
0x7ffe1096ad6c
0x7ffe1096ad70
Number of ints  between 0x7ffe1096ad6c and 0x7ffe1096ad70: 1
Number of bytes between 0x7ffe1096ad6c and 0x7ffe1096ad70: 4
```

Pointers and Arrays and Strings

- very close connection in C between pointers, arrays, and strings (which are just arrays of characters)
- an array without an index represents the base address of an array, so it is a pointer

Pointers and Arrays

```
cat -n ch04/arrays.c
```

```c
#include <stdio.h>

int main(void)
{
  int loop = 1;
  int numbers[10];   /* An array of 10 int. */
  int *p;

  for (int i = 0; i < 10; i++) {
    numbers[i] = i * i;
    printf("%d ", numbers[i]);
  }
  printf("\n");
  printf("Done with loop %d.\n", loop++);

  p = numbers;  /* p points to numbers. */
  for (int i = 0; i < 10; i++) {
   printf("%d ", *(p + i));
  }
  printf("\n");
  printf("Done with loop %d, p - numbers = %ld.\n", loop++, p - numbers);

  for (int i = 0; i < 10; i++) {
   printf("%d ", p[i]);
  }
  printf("\n");
  printf("Done with loop %d, p - numbers = %ld.\n", loop++, p - numbers);

  return 0;
}
```

```
gcc ch04/arrays.c
```

```
./a.out
```

```
0 1 4 9 16 25 36 49 64 81
Done with loop 1.
0 1 4 9 16 25 36 49 64 81
Done with loop 2, p - numbers = 0.
0 1 4 9 16 25 36 49 64 81
Done with loop 3, p - numbers = 0.
```

Pointers and Arrays

- line 11 uses array indexing to access the elements of the array int's
- line 16 sets the integer pointer p to numbers
  - since numbers is an array, p points to the beginning of numbers in memory
- in line 18, *(p + i) refers to the int that is i integers past the beginning of the array numbers
  - p + i points to the location that is i * sizeof(int) bytes past the beginning of numbers
  - *(p + i) dereferences the address p + i as an int and is equivalent to p[i]
- line 24 pointer p uses array indexing, as in line 11

Pointers and Arrays

- in C, arrays are really just pointers (and vice-versa)
- an array is really just a pointer to the beginning of the array
- the type of the pointer (e.g. int*, double*) determines how the bytes at that location and subsequent locations are interpreted
- the programmer is responsible for keeping track of the length of the array
- pointers allow programmers to write terse and cryptic (but powerful) code
  - C idioms

Pointers and Arrays

```
cat -n ch04/arrays2.c
```

```c
#include <stdio.h>

int main(void)
{
  int loop = 1;
  int numbers[10];  // An array of 10 int.
  int *p;

  /* The original loop. */
  for (int i = 0; i < 10; i++) {
    numbers[i] = i * i;
    printf("%d ", numbers[i]);
  }
  printf("\n");
  printf("Done with loop %d.\n", loop++);

  p = numbers;
  while (p < numbers + 10) {
    printf("%d ", *p++);
  }
  printf("\n");
  printf("Done with loop %d, p - numbers = %ld.\n", loop++, p - numbers);

  p = numbers;
  for (p = numbers; p < numbers + 10; p++) {
   printf("%d ", *p);
  }
  printf("\n");
  printf("Done with loop %d, p - numbers = %ld.\n", loop++, p - numbers);

  return 0;
}
```

```
gcc ch04/arrays.c
```

```
./a.out
```

```
0 1 4 9 16 25 36 49 64 81
Done with loop 1.
0 1 4 9 16 25 36 49 64 81
Done with loop 2, p - numbers = 0.
0 1 4 9 16 25 36 49 64 81
Done with loop 3, p - numbers = 0.
```

Pointers and Arrays and Strings

- in line 18, numbers + 10 means the address plus 10 int's past the start of numbers
- in line 19, we encounter *p++, which is evaluated as
  - use the value of the int that p points to (*p)
  - increment p (p++); i.e., advance p to point to the next int
- for loop that begins at line 25 is equivalent to the while loop at line 18

Pointers and Strings

- a string in C is an array of characters
  - terminated by an ASCII NULL character (`'\0'`)
  - strings are really just pointers of type char* that point to the first character in the string
  - to read the string, go to that starting address and read bytes until you encounter the terminating ASCII NULL

Pointers and Strings

```
cat -n ch04/strlen.c
```

```
#include <stdio.h>

int my_strlen(char *s)
{
    int count;
    for (count = 0; *s; s++, count++);
    return count;
}

int main(void)
{
  char s[] = "Hello, world!";
  char t[] = "You're a lying dog-faced pony soldier!";

  printf("The string '%s' contains %d characters + an ASCII null.\n", s, my_strlen(s));
  printf("The string '%s' contains %d characters + an ASCII null.\n", t, my_strlen(t));

  return 0;
}
```

```
gcc ch04/strlen.c
```

```
./a.out
```

```
The string 'Hello, world!' contains 13 characters + an ASCII null.
The string 'You're a lying dog-faced pony soldier!' contains 38 characters + an ASCII null.
```

Pointers and Strings

- on line 6

```
for (count = 0; *s; s++, count++);
```

is equivalent to

```
count = 0;
while (*s) {
    s = s + 1;      // advance one character
    count = count + 1;  // increment count
}
```

- we start with the pointer s at the beginning of the string
- the increment s++ moves us across the string
- *s will not be zero until we reach the ASCII NULL that terminates the string so the non-zero value *s is interpreted as "true" and the loop continues

Pointers and Strings

- can we declare count inside the for statement?

```
int strlen(char *s)
{
    for (int count = 0; *s; s++, count++);
    return count;
}
```

- count would be local to the for statement so it would not be available to return

Function Pointers

- pointers to functions are also available in C
  - pass functions as parameters
  - assign functions to variables
- e.g., pass a comparison function to a sorting algorithm to achieve polymorphism

Function Pointers

- function pointer declarations require careful use of parentheses

```
double (*fun_pt)(double, double);
```

  - defines a pointer to a function with two double parameters which returns a double

```
double *fun_pt(double, double);
```

  - defines a function prototype for a function with two double parameters which returns a pointer to a double

Function Pointers

- a function name can be implicitly converted to a function pointer when needed
- here we convert pow() from the C math library to a pointer

```
cat -n ch04/fun_pt.c
```

```
#include <math.h>  /* For pow(). */
#include <stdio.h>
int main(int argc, char **argv)
{
  double (*fun_pt)(double, double) = pow;
  printf("%e\n", (*fun_pt)(2, 0.5));
}
```

```
gcc ch04/fun_pt.c -lm  # -lm tells gcc to link in pow() from the math library.
```

```
./a.out
```

```
1.414214e+00
```

Dynamic Memory Allocation

- in C/C++/Java, you can dynamically allocate new objects in memory
- when running a program, there are two types of memory
  - stack memory for function calls, static variables, etc.
  - heap memory for dynamically allocated memory whose contents persist
- each computational process on the computer has its own stack memory
- a family of functions in C are used to dynamically allocate memory to create new arrays and other objects
  - the most commonly used memory allocation function is malloc()

Dynamic Memory Allocation

- malloc()
  - allocates a contiguous region of memory and returns a pointer to the beginning of the region
  - it does not initialize the allocated memory
  - prototype: void *malloc (size_t bytes);
- calloc()
  - does the same thing as malloc()
  - but also clears the allocated region by initializing all the bytes to be zero
  - prototype: void *calloc (size_t bytes, int num_of items);
- since both return void*, casting to a different pointer is recommended

Dynamic Memory Allocation

```
cat -n ch04/alloc.c
```

```
#include <stdio.h>
#include <stdlib.h>  /* You need this to use malloc() and calloc(). */

int main(void)
{
  unsigned int n = 42;
  double *x;
  int *a;

  x = (double*) malloc(n * sizeof(double));  /* Space for n doubles. */
  free(x);

  a = (int*) calloc(n, sizeof(int));  /* Space for n integers. */
  free(a);

  a = calloc(n, sizeof(int));  /* Implicit pointer type conversion! */
  free(a);

  return 0;
}
```

```
clang -Weverything -Wall -pedantic ch04/alloc.c
```

```
./a.out
```

Dynamic Memory Allocation

- if the memory cannot be allocated (e.g., no memory is left), NULL is returned
  - need to include stdlib.h
  - really just 0
- NULL also used for initializing a pointer variable to indicate it does not currently hold a valid address

Dynamic Memory Allocation

```
cat -n ch04/null.c
```

```
     1	#include <stdio.h>
     2	#include <stdlib.h>
     3	
     4	int main(int argc, char **argv)
     5	{
     6	  int *n = NULL;
     7	
     8	  *n = 42;
     9	  printf("n: %d\n", *n);
    10	
    11	  return 0;
    12	}
```

```
clang -Weverything ch04/null.c
```

```
ch04/null.c:4:14: warning: unused parameter 'argc' [-Wunused-parameter]
int main(int argc, char **argv)
             ^
ch04/null.c:4:27: warning: unused parameter 'argv' [-Wunused-parameter]
int main(int argc, char **argv)
                          ^

2 warnings generated.
```

```
./a.out
```

```
Segmentation fault (core dumped)
```

Dynamic Memory Allocation

- common C practice for checking error for malloc

```
cat -n ch04/alloc1.c
```

```
#include <stdio.h>
#include <stdlib.h>  /* You need this to use malloc() and calloc(). */

int main(void)
{
  int n = 42;
  double *x;

  if ((x = (double*) malloc(n * sizeof(double))) == NULL) {
    fprintf("Allocation of x failed!\n");
  }
  else {
    free(x);
  }

  return 0;
}
```

Dynamic Memory Allocation

- call free to return memory to the system

```
cat -n ch04/free.c
```

```
     1	#include <stdio.h>
     2	#include <stdlib.h>
     3	
     4	int main(int argc, char **argv)
     5	{
     6	  int *n = (int*) malloc(42 * sizeof(int));
     7	
     8	  free(n + 3);  /* n no longer points to the start of the buffer.
     9	
    10	  return 0;
    11	}
```

```
clang -Weverything ch04/free.c
```

```
ch04/free.c:4:14: warning: unused parameter 'argc' [-Wunused-parameter]
int main(int argc, char **argv)
             ^
ch04/free.c:4:27: warning: unused parameter 'argv' [-Wunused-parameter]
int main(int argc, char **argv)
                          ^
2 warnings generated.
```

```
./a.out
```

```
free(): invalid pointer
Aborted (core dumped)
```

Dynamic Memory Allocation

- C does not perform garbage collection
  - it is up to the programmer to clean up and release memory when it is no longer needed
  - otherwise, may run out of memory causing your program to crash
  - memory leak results from a failure to properly manage memory usage
- all allocated memory is typically released when a program ends