Pointers and Strings

CSE 130: Introduction to Programming in C

Stony Brook University

Pointer Basics

- Remember that every variable occupies some location in the computer's memory
- A ***pointer*** is a special type of variable that stores the memory address of another variable
- Pointers are used to access memory and manipulate addresses

Pointer Operators

- If `v` is a variable, then `&v` is the location (address) in memory of its stored value
- If `p` is a pointer, then `*p` is the value of the variable at memory location `p`
- The `*` operator is also used to declare a pointer:

```
int *p; /* p is a pointer to an int */
```

Pointer Assignment Examples

```
int *p;

p = 0;

p = NULL; /* equivalent to p = 0; */

p = &i;   /* "p points to i" */

p = (int *) 1776; /* 1776 is an absolute address in
                     memory */
```

A Simple Pointer Example

- Start with:

```
int a = 1, b = 2, *p;
```

- `p = & a; /* p is assigned the address of a */`
- `b = *p; /* b gets the value pointed to by p */`

  which is equivalent to `b = a;`

```
#include <stdio.h>

int main(void)
{
  int i = 7, *p = &i;

  printf("Value of i: %d\nLocation of i: %p\n",
         *p, p);

  return 0;
}
```

Sample program output:

```
Value of i: 7
Location of i: effffb24
```

`%p` prints an address in a system-specific format (usually hexadecimal)
`%u` prints an address as an unsigned decimal integer

Pointer Conversions

- Even though pointers all store memory addresses, they are not interchangeable!
- Conversions are only allowed if the right side is 0, or if one of the types is a pointer to `void`
  - `void *` is a generic pointer type

```
int *p;
float *q;
void *v;
```

Legal Assignments	Illegal Assignments
p = 0;	p = 1;
p = (int *) 1;	v = 1;
p = v = q;	p = q;
p = (int *) q;	

Pointer Prohibitions

- Not every value is stored in an accessible memory location
- Do not point at constants (e.g., &3)
- Do not point at ordinary expressions (e.g., &(k + 99))
- Do not point at register variables

Call-by-Reference

- Ordinarily, when a variable is passed as an argument to a function, its value is copied and not modified
  - This is known as "call-by-value"
- Pointers can be used to implement "call-by-reference" instead, where the original variables *are* modified
  - Pass the addresses of variables as the arguments

```
#include <stdio.h>

void swap (int *p, int *q) /* parameters are pointers */
{
  int tmp = *p;
  *p = *q;
  *q = tmp;
}

int main(void)
{
  int i = 3, j = 5;
  swap(&i, &j);           /* arguments are addresses */
  printf("%d %d\n", i, j);
  return 0;
}
```

Arrays and Pointers

- An array name is an address or pointer value
- Pointers, like arrays, can be subscripted
- An array name, however, is a pointer value that is fixed; a regular pointer variable can take different addresses as values

Pointer Arithmetic

- Suppose `a` is an array and `i` is an integer
- Then `a[i]` is equivalent to `*(a + i)`
- The expression `a + i` is the *i*th offset from the base address of array `a`
  - The actual address depends on the type of values stored in `a`

Pointer Arithmetic and Arrays

- Assume that an int occupies 4 bytes of memory
- Given: `int a[100], *p;`

  `p = a;` is equivalent to `p = &a[0];`

  and

  `p = a + 1;` is equivalent to `p = &a[1];`

Two Ways to Sum an Array

```
for (p = a; p < &a[100]; p++)
{
  sum += *p;
}
```

```
for (i = 0; i < 100; i++)
{
  sum += *(a + i);
}
```

Arrays as Function Arguments

- When an array is passed as an argument to a function, its base address is passed "call-by-value"
  - The array elements are not copied
- Equivalent function headers:

```
double sum (double a[], int n) { ... }
double sum (double *a, int n)  { ... }
```

Example: Bubble Sort

- Bubble sort is a simple algorithm for sorting lists of values
- It uses multiple passes (rounds) to sort the data
  - Adjacent pairs of values are compared, and reordered as necessary
  - Each pass guarantees that the largest unsorted value will be moved to its proper sorted place

```
void bubble sort (int a[], int n)
{
  int i, j;

  for (i = 0; i < n - 1; i++)
  {
    for (j = n - 1; j > i; j--)
    {
      if (a[j-1] > a[j])
      {
        swap(&a[j-1], &a[j]);
      }
    }
  }
}
```

Dynamic Memory Allocation

- Sometimes we need to allocate memory for a data structure at run-time
  - We don't know ahead of time how much space will be needed
- We can do this using the `calloc()` ("continuous allocation") and `malloc()` ("memory allocation") functions from *stdlib.h*

Using `calloc()`

- `calloc()` takes two arguments, both of type `size_t` (an unsigned integral type)
- `calloc(n, el_size)` allocates contiguous space for an array of `n` elements, where each element occupies `el_size` bytes
- The space is initialized with all bits set to 0
- If the call is successful, a pointer of type `void *` is returned; otherwise, `NULL` is returned

```c
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
  int *a;
  int n;
  ...
  a = calloc(n, sizeof(int));
  ...
}
```

Using `malloc()`

- `malloc()` works in a similar fashion to `calloc()`, but it only takes one argument: the total number of bytes to allocate.

```c
a = malloc(n * sizeof(int));
```

- `malloc()` doesn't initialize the memory space to 0

Set Me Free!

- Memory allocated by `malloc()` and `calloc()` is ***NOT*** automatically released to the system when a function exits
- The programmer must explicitly release the memory using the `free()` command (and passing in a pointer to the memory to be deallocated):

```c
free(a);
```

Example: Merge Sort

- The mergesort algorithm works by repeatedly dividing an array in half, sorting the halves, and then merging the sorted halves back together.
- This is actually much more efficient than bubble sort: $O(n \log n)$ time rather than $O(n^2)$ time for n elements

```
void merge(int a[], int b[], int c[], int m, int n)
{
  /* c is the destination array, m and n are the sizes
     of a and b */
  int i = 0, j = 0; k = 0;
  while (i < m && j < n) /* both arrays have data */
  {
    if (a[i] < b[j])
      c[k++] = a[i++];
    else
      c[k++] = b[j++];
  }
  while (i < m)          /* collect leftover data */
    c[k++] = a[i++];
  while (j < n)
    c[k++] = b[j++];
}
```

```
void mergesort(int key[], int n)
{
  int j, k, *w;

  w = calloc(n, sizeof(int));
  assert(w != NULL); /* check that calloc() worked */

  for (k = 1; k < n; k *= 2)
  {
    for (j = 0; j < n - k; j += 2 * k)
    {
      merge(key + j, key + j + k, w + j, k, k);
    }
    for (j = 0; j < n; j++)
    {
      key[j] = w[j]; /* copy w back into key */
    }
  }
  free(w); /* release memory allocated by calloc() */
}
```

Strings

- A ***string*** is a one-dimensional array of type `char`
- Strings are delimited by the end-of-string sentinel \0, or null character
- Strings effectively have a variable length (determined by the position of \0) inside a maximum length (the size of the `char` array)
- The array containing a string ***MUST*** include storage for the end-of-string sentinel

String Constants

- String constants are written between double quotes
  - “abc” is a string constant of size 4 (don’t forget the end-of-string sentinel)
- String constants are treated as pointers
  - ```
    char *p = “abc”;
    printf(“%s %s\n”, p, p+1); /* prints abc bc */
    ```
    ```

String-Handling Functions

- `char *strcat(char *s1, const char *s2)` — concatenates s1 and s2, puts the result in s1, and returns s1
- `int strcmp(const char *s1, const char *s2)` — returns an integer less than, equal to, or greater than 0, based on whether s1 is less than, equal to, or greater than s2

More String-Handling Functions

- `char *strcpy(char *s1, const char *s2)` — Copies characters from s2 into s1 until \0 is encountered, then returns s1
- `size_t strlen(const char *s)` — Returns the number of characters before \0 in s