Introduction to Computer Science and Programming in C

Session 25: December 4, 2008
Columbia University
Announcements

- Final Exam: Tuesday, 12/16, 1:10 pm - 4:00 pm
  Mudd 233 (our normal room)
Variables Revisited

- What actually happens when we declare variables?
  ```c
  char a;
  ```

- C reserves a byte in memory to store `a`.

- Where is that memory? At an `address`.

- Under the hood, C has been keeping track of variables and their addresses.
Pointers

- We can work with memory addresses too. We can use variables called **pointers**.

- **pointer**: an address variable

- All pointers are the same size, regardless of what they point to
Declaring a pointer variable:

```c
int * x_ptr; /* declares a pointer to an int */
```

The & operator means “the address of this thing”

The * operator means “the thing this points to”
int * x_ptr; /* declares a pointer to an int */
int x, y;

x_ptr = &x; /* set x_ptr to the address of x */
y = *x_ptr; /* set y to whatever x_ptr points to */

/* is equivalent to */
y = x;
Some vocabulary

- * operator is also known as dereference
- a pointer references a variable in memory
C blurs the distinction between pointers and arrays

When we declare an array
\begin{verbatim}
char A[10];
\end{verbatim}
what is A?

A can be treated as a pointer to the first element of A
Pointers and Arrays

- In other words, the following two lines are equivalent:
  - `char * array_ptr = &A[0];`
  - `char * array_ptr = A;`

- This also means the following:
  - `A[0] == *array_ptr`
Pointers and Arrays

- When we want a function to be able to modify the value of a variable, we pass it by reference:
  \[
  \text{scanf}((\text{price}, \"\$%f\", \&\text{dollars);}
  \]

- Because arrays are basically pointers, this happens *automatically* when we pass arrays to functions.

- For example:
  \[
  \text{strcpy}((\text{stringA, stringB});
  \]
Pointer Arithmetic

- What if A was an array of ints?
  \[ A[1] == *(array_ptr+1) \] ??

- Yes. C automatically keeps pointer arithmetic in terms of the size of the variable type being pointed to.

- Be careful to keep track of what C does for you and what it does not.
Memory Management

- We discussed before that C does not like to initialize arrays with variable sizes.

- To get around this, you can use stdlib.h’s `malloc()` command.

- `malloc()` stands for memory allocation.

- `malloc(N)` returns a pointer to an allocated block of memory of N bytes.
malloc()

- Typical usage:
  ```c
  int N = 40000;
  char *giantString = malloc(N*sizeof(char));
  ```

- Returns a null pointer if malloc fails.

- When we are done with the memory, we can free it with:
  ```c
  free(giantString);
  ```
Memory Leaks

- int N = 40000;
  char *giantString = malloc(N*sizeof(char));
  strcpy(giantString, argv[1]);
  giantString = malloc(N*sizeof(char));

- Now a huge block of memory is allocated but the program has no way of finding it.

- If this code runs a lot, the amount of memory the program is using will keep growing.
Measuring Algorithms

- In Computer Science, we want to be able to describe the running time and memory requirements of our algorithms

- A couple challenges:
  - Running time and space typically depend on input size
  - Algorithms are run on different machines
Measuring Algorithms

- For varying input sizes, we can write our time and space requirements as functions of $N$.
- For varying implementation, we need our description to not care about constant factors.
Example

- What is the running time of a function that sums an array of size 5 on a machine that takes 2 seconds to add numbers? $4 \times 2 = 8$

- What if array is size $N$? $2(N-1)$

- What if it takes $c$ seconds to add? $c(N-1)$
Big-O

- \(g(n) = O(f(n))\)
  means that for some \(c\)
  \(g(n) \leq c(f(n))\)

- In other words, big-O means less than some constant scaling.

- In big-O notation, what is the running time to sum an array of size \(N\)?
  \(c(N-1) = O(N)\)
Sorting

- One of the most studied problems in CompSci
- We are given N numbers
- Put the numbers in order
  - least to greatest, greatest to least, alphabetical, etc.
  - compare two numbers at a time
Algorithm for Sorting

- In English: Given 50 index cards with numbers on them, how do you put them in order?

- Lots of different algorithms. We’ll go over three
Bubble Sort

- Worst algorithm ever
- Start at beginning of deck
- Compare current and next cards. If next card should be before current, swap. Move to next card.
- Keep passing through deck until no more swaps necessary.
Selection Sort

- Smarter cousin of Bubble Sort
- Find the smallest unsorted card
- Swap smallest with the first unsorted card
- Consider that card sorted, and repeat
Merge Sort

- If deck is 2 or less cards, just sort and return
- Split deck into two halves
- Merge Sort each half-deck (recursion!)
- Then, merge the two half-decks:
  - Look at top of each deck. Take the smallest of the two. Repeat until decks are combined.
Running time

- Bubble Sort: $O(N^2)$
- Selection Sort: $O(N^2)$
  But the algorithm seems better organized.
- Merge Sort: $O(N \log(N))$
Pseudocode

- Mix of English and programming language
- Use programming constructs to keep thoughts organized: loops, conditionals, variables
- But use any syntax that is clear and consistent
- And use functions that are obvious to abstract busywork
Pseudocode example

- print "Enter your friends’ names:"
  while input is not "quit"
    input = keyboardInput
    add input to array Contacts

  sort Contacts
  output Contacts

- Even though this is a simple piece of code, if it were written in C, it would be much harder to understand
**Modular Programming**

- **modular** - Designed with standardized units or dimensions, as for easy assembly and repair or flexible arrangement and use: *modular furniture; modular homes.*

- Organize programs into interchangeable parts

- Keep functions that deal with a certain type together, but separate them from functions that deal with other types.
calendar.c

struct appointment
    sort()
    addEvent()
    cancelEvent()
    printDate()
    printMonth()
    printWeek()
    ...
    main()
calendar.h
struct appointment
<function declarations>

event.c
#include "calendar.h"
sort()
addEvent()
cancelEvent()

print.c
#include "calendar.h"
printDate()
printMonth()
printWeek()

calendar.c
#include "calendar.h"
main()
Pointers to pointers

- Recall that C arrays and pointers are basically the same:
  ```c
  int A[10];
  int *A_ptr = A;
  ```

- How does C store 2d arrays?
  ```c
  int B[10][10];
  ```

- B is a pointer to an array of pointers
Pointers to pointers

```
int **
   B
   |-> B[0]
   |   |-> B[0][0] B[0][1] B[0][2] B[0][3] B[0][4]
   |   |-> B[0][1] B[0][2] B[0][3] B[0][4]
   |   |-> B[0][2] B[0][3] B[0][4]
   |   |-> B[0][3] B[0][4]
   |   |-> B[0][4]
   |-> B[1]
   |   |-> B[1][4]
   |-> B[2]
   |   |-> B[2][4]
   |-> B[3]
   |   |-> B[3][4]
   |-> B[4]
   |   |-> B[4][4]
```
malloc()

- We can dynamically allocate multi-dimensional arrays

- int **C;
  
  C = (int**) malloc(N*sizeof(int*));

- for (i=0; i<N; i++) {
    
    C[i] = (int*)malloc(N*sizeof(int));
}
Pointers to functions

- It is occasionally useful to use pointers to functions

- Since functions are stored in memory, we can reason about their addresses too

- This allows us to say, “run the function at address _____ on these arguments”

- Useful for being truly general, e.g. stdlib qsort
**Function Pointer Syntax**

- `int (*f_ptr)();
  /* pointer to function that returns an int */`

- Parentheses are important. Without parentheses, `f_ptr` looks like it returns a pointer to an int.

- `int (*f_ptr)(int, int);
  /* function takes 2 ints as arguments */`

- `int greater_than(int a, int b);
  f_ptr = greater_than;`
qsort example

- Stdlib’s qsort function is a general sorting function.
- Sort an array of any type, using any comparison criterion.
- Define that comparison as a function pointer.

```c
void qsort(void *base, size_t n, size_t size, 
           int (*cmp)(const void *, const void *));
```
qsort example

- Compare function should take two entries A and B,
  - return +1 if A>B
  - return -1 if A<B
  - return 0 if A==B
int greater_than(const void *x, const void *y)
{
    float *a = (float*)x, *b = (float*)y;

    if (*a>*b)
        return 1;
    if (*a<*b)
        return -1;
    return 0;
}

void mySort(float A[], int N)
{
    int (*f_ptr)(const void *, const void *) = greater_than;
    qsort((void*)A, N, sizeof(float), f_ptr);
}
Linked Lists

- Store each element in a struct that contains the data and a pointer to the next struct: a **node**
- Keep a pointer to the first node
- Following a linked list is like a scavenger hunt
Linked Lists

- \texttt{struct node} {
  \hspace{1em} int data;
  \hspace{1em} struct node * next;
}\;

struct node * start;

- How do we add a node at beginning of list?

\begin{itemize}
  \item Allocate new node, set \texttt{next} pointer to \texttt{start},
  set \texttt{start} to new node.
\end{itemize}
Linked Lists

- How do we add a node to the end of the list?
  - Follow pointers to last node, allocate new node, set last node’s `next` to new node.
- How do we add in the middle of the list?
  - Set previous node’s `next` to new node, set new node’s `next` to next node.
- How do we delete a node?
Doubly Linked Lists

- Keep a **next** pointer and a **previous** pointer.
- A little extra work for adding and removing, but allows for faster backtracking.
Binary Trees

- Finding an item in a list or array is usually an $O(N)$ operation.

- We can create a structure that makes it faster (at a cost; a tradeoff)

- We use a tree structure, which is like a linked list, except each node has more than one pointer.
Binary Trees

- Binary tree: Each node has left and right child.
- Left child is less than, right child is greater than

```c
struct node {
    int data;
    struct node *left;
    struct node *right;
}

struct node *root;
```
Binary Trees

Inserting number $x$ into a Binary Tree:

0. Start at root

1. If current node is NULL, create new node and set node to $x$

2. Otherwise, if $x \geq$ current node, follow right pointer, else follow left pointer. Goto 1.
Binary Trees

- Finding an item $x$ in a binary tree:
  
  0. Start at root
  
  1. If current node is $x$, return
  
  2. If $x \geq$ current node, follow right pointer else, follow left pointer
  
  3. If node is NULL, return "not found", otherwise goto 1.
Binary Trees

- On average, lookup and insertion take $O(\log N)$ time
- But worst case is still $O(N)$