Midterm Review Packet

Contents

This packet contains a copy of all of the section notes distributed in Nicholas Boucher's CS50 Section (*Tuesdays, 4-5:30 CGIS-S-040*)

Advice

For materials any materials that may still not seem clear, do consult https://study.cs50.net/.

Do not hesitate to reach out to Nicholas at nboucher@college.harvard.edu with any questions, however please note that staff cannot answer technical questions after the testing period begins.

CS50 Section. Week 4. 9/20/16.

Tuesdays 4:00-5:30pm, CGIS S-040

> Nicholas Boucher nboucher@college.harvard.edu

Important links

- Sorting algorithm visualizations:
  - Bubble Sort
  - Insertion Sort
  - Merge Sort
- C language reference: https://reference.cs50.net/
- This week's material on Study50: Study50
- CS50 Discuss: https://cs50.harvard.edu/discuss
- CS50 Style Guide: https://manual.cs50.net/style/

Section Agenda

1. Suggestions from Last Pset
2. Review of Previous Concepts
3. Searching
4. Sorting
5. Recursion

Suggestions from Last Pset

Use Style50

Most of the general issues that I saw on psets could have been caught by style50. Everyone seems to be doing a good job of running check50, but do be sure to run style50 and correct any issues before submitting your code.

Comments

In the last pset, many people either entirely lacked or used significantly fewer comments than is expected. Be sure to document what you are doing in your code using comments - you should not just be explaining *what* but also *why* your code is the way it is.

Also, do not forget to include the required comment stubs at the top of each file and before each function you write (except perhaps main). The following is excerpted from the Style Guide:

Comments at the Top of Each File

```
/**
 * hello.c
 *
 * David J. Malan
 * malan@harvard.edu
 *
 * Says hello to the world.
 */
```

Comments Before Each function

```
/**
 * Returns n^2 (n squared).
 */
int square(int n)
{
    return n * n;
}
```

Magic Numbers

Magic numbers are any numbers which are "hardcoded" into your code. As a general rule, we try to avoid these hardcoded constants at all costs, since it:

- makes code harder to read and understand
- makes it more difficult and error-prone to change these values later

Instead of using magic numbers, we should instead use #define statements at the top of each pertinent file.

Spacing

It is important that your code stylistically conforms to the normal conventions of C. One recurring issue in the past pset was improper spacing between variables, costants, and operators. There should always be a space on either side of an operator. For example, adding one to the variable x should look like this:

```
x = x + 1;
```

NOT this:

```
x = x+1;
```

Debug50

Debug50 is a great way to locate issues within your program. It is a graphical debugger that will allow you to see the current value of variables at any point in the program. Use debug50 by placing breakpoints on the relevant lines of code (where you think the bug might be) and executing `debug50 <program name>`.

Review of Previous Concepts

Modulo

Modulo is a basic mathematical operator in C which calculates the "remainder" from division. It is denoted with the % symbol. For example, `7 % 3 = 1`, since `7 / 3 = 2 **remainder 1**`.

ASCII

ASCII, or "American Standard Code for Information Interchange", is a standard that equates human-readable sysmbols (such as the alphabet)

to numbers which can be represented in binary. C stores `char`s as ASCII `int`s under the hood, which is why you can do arithmetic on `char`s.

Functions

Functions are a way to *factor out* some of your code into named, reusable chunks. Functions can take an arbitrary numner of arguments (zero included), which allow you to pass information into them. Functions also have the ability to return a single up to one value.

Functions are created using the following syntatic form:

```
<return type> <name>(<argument type> <argument name>, ...)
{
    <function code here>
}
```

Function prototypes

Say you want to use a function besides `main`. It's not enough to just write the function; C reads top-to-bottom so you'll need to tell C some information about the function ahead of time.

```
#include <stdio.h>

// function prototype
// <return type> <name>(<argument type>, ...)
// this is the same as in the function definition, except
// you don't specify the argument names
float months_to_years(int);
```

```c
int main(void)
{
    // prints 1.50
    // (the "%.2f" means to print a float with
    // 2 decimal places)
    printf("%.2f\n", months_to_years(18));
}

// function definition
//<return type> <name>(<argument type> <argument name>,...)
// here's where you *actually* implement the function
float months_to_years(int months)
{
    return years / 12.0;
}
```

By the way, if you were actually writing this code, you wouldn't include any of these comments. They're just for illustration here

Command Line Arguments

Values can be passed into a program from the command line. These are interpreted as arguments to the `main` function. Traditionally, we have written `main` like this:

```c
int main(void) {
    <code here>
}
```

If we want to be able to accept values from the command line, however, we must change the structure of `main` to look like this:

```c
int main(int argc, string argv[])
```

```
{
    <code here>
}
```

`argc` is the argument count - the number of arguments passed to the program. `argc` is always greater than or equal to 1, since the name of the program is counted as the first argument.

`argv` is the argument vector - the actual values passed into the program. It is an array of strings, so it should be accessed using the array indexing syntax (i.e. `argv[n]`, where n is the index of the element you are trying to access). `argv[0]` is always the name of the program being executed.

Arrays

Arrays let you store a bunch of related pieces of data in a clean, organized way.

Here's the primary way to create an array:

```
// <datatype> <name>[<size>];
string my_classes[4];
// you don't have to fill in all the elements of the array
my_classes[0] = "CS50";
my_classes[1] = "Ec 10";
my_classes[2] = "Expos 20";
my_classes[3] = "Math 55";
```

You can use this shorthand if you know all the elements of the array ahead of time. Notice that C will infer the array size for you, so you don't need to write it!

```
// <datatype> <name>[] = {elements};
int important_years[] = {1636, 1776, 2015};
```

You can access and change the elements of an array like this:

```
// prints "This is CS50." (without quotes)
printf("This is %s.\n", my_classes[0]);

// prints 1787
important_years[1] = 1787;
printf("%i\n", important_years[1]);
```

Strings

Strings of text are really just arrays of characters. You can work with them just like normal arrays, for the most part:

```
// prints "u" (without quotes)
string band = "Young the Giant";
char third_letter = band[2];
printf("%c\n", third_letter);
```

Note that strings are immutable - that is, you cannot change a string once it has been created. For instance, the following code will cause an error.

```
band[2] = 'u';
```

Do keep in mind that *replacing* a string with a complete new string is not the same as modifying an existing string, which is why we can do this:

```
band = "Green Day";
```

Arrays, the General Case

So far, we have given examples of arrays using `int[]` and `char[]` (read as `string`). Know, however, that arrays can hold any single data type, so constructs like `string[]` and `float[]` are perfectly valid.

Asymptotic Notation (O, Ω)

We use specific notation to describe the theoretical *running time* of a given algorithm. `O` typically refers to the *Upper Bound* of the running time and `Ω` (Omega) typically refers to the `Lower Bound` of the running time. You can think of these as best case and worst case running time - or *time complexity* - scenarios.

As we introduce new algorithms, we will practice determining the Upper and Lower runtime complexity bounds.

Searching

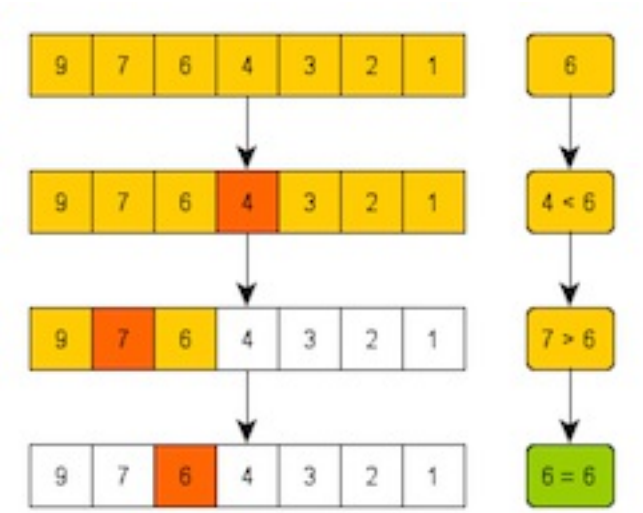
Linear Search

Linear Search is the most basic type of search. It involves simply examing each element in an array in the order that they appear and comparing it to the value we are looking for.

Excercise: Implement a linear search function for searching arrays of `int`s.

Binary search

Binary search finds a value in a *sorted* array. You cut the problem in half at each stage by eliminating the half of the array that couldn't possibly contain the element you're looking for.




```
while there are still elements to search:
    find middle element
    if middle element = what we're looking for:
        return true
    if middle element < what we're looking for:
        look in lower half
    if middle element > what we're looking for:
        look in upper half

return false (give up)
```

Think back to David's first lecture containing the phone book example.

Sorting

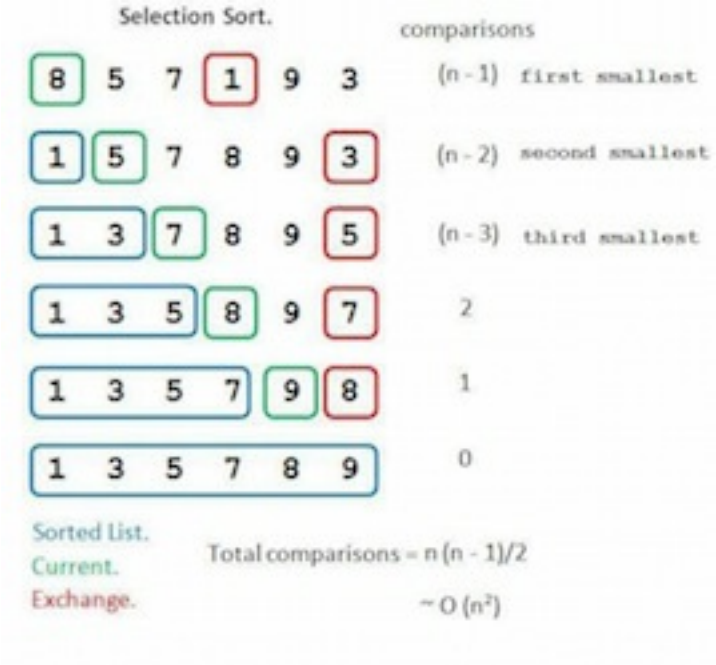
Bubble sort

In a sorted array, every element will be smaller than the one to its right, so this sort swaps two neighboring elements if they're out of order. This way, the biggest elements end up *bubbling* to the top.


```
for each element X in the array:
    for each remaining element Y in array:
        swap X and Y if X > Y
```

Selection sort

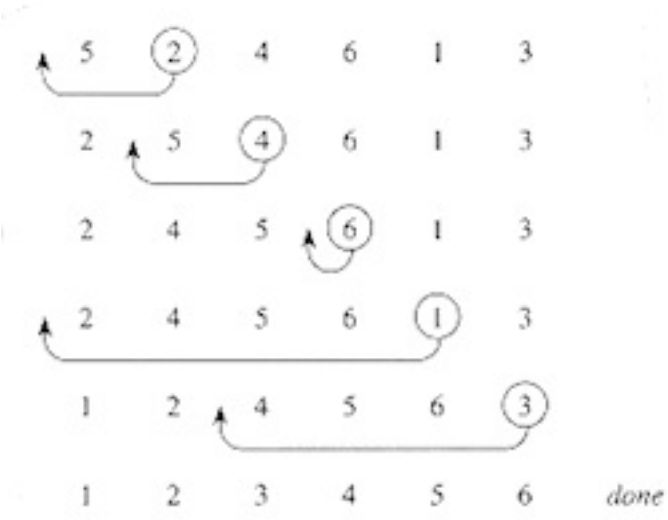
This sort *selects* the smallest elements remaining in the array and swaps them so they're in the front.




```
for each element in the array:
    find the smallest element in the rest of the array
    swap that element with the current element
```

Insertion sort

This sort sorts the array as it sees it, *inserting* each element in its proper place in the array.

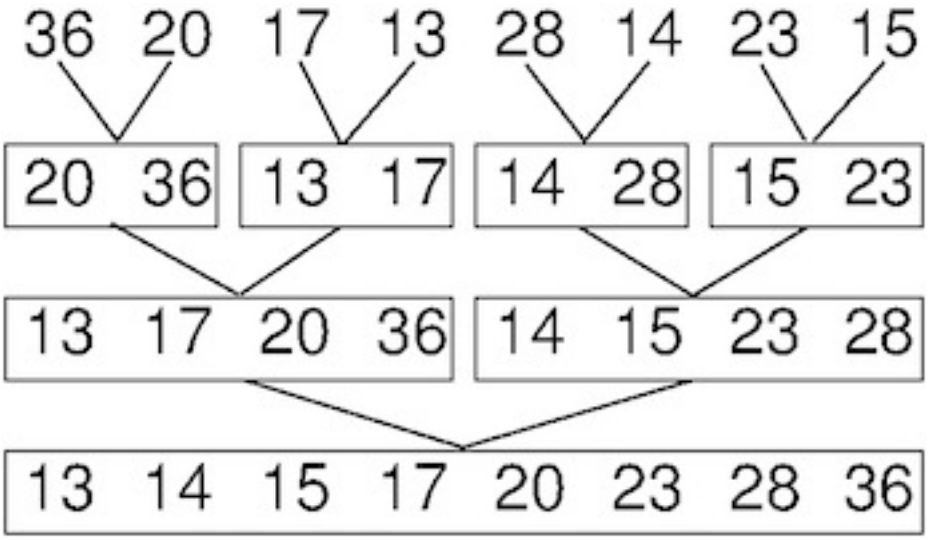



```
for each element in the array:
    look at the element X directly to our right
```

```
    shift all elements on our left to the
          right if they're > X
    put X in the vacated spot
```

Merge sort

This sort recursively sorts its sub-arrays and then *merges* them together to produce one larger sorted sub-array.




```
function sort:
    sort left half
    sort right half
    merge two halves

function merge:
    create temporary array
    while elements in left & right sub-arrays:
        compare first elements of two sub-arrays;
            place smaller one in temp array
    dump any remaining elements in longer sub-array
            in temp array
    copy temp array onto original array
```

Algorithm cheatsheet

Name	What it does	Worst-case	Best-case
Linear search	Finds an element in a list by searching left-to-right	O(n)	Ω(1)
Binary search	Finds an element in a sorted list using divide-and-conquer	O(log n)	Ω(1)
Bubble sort	Sorts a list by bubbling biggest elements to end	O(n^2)	Ω(n)
Selection sort	Sorts a list by moving smallest elements to front	O(n^2)	Ω(n^2)
Insertion sort	Sorts a list by moving elements to properly sorted place	O(n^2)	Ω(n)
Merge sort	Recursively sorts a list by partitioning and merging	O(n log n)	Ω(n log n)

More at http://www.bigocheatsheet.com/.

Recursion

Recursive code is code in which a function makes a call to itself. It is sort of the *Inception* (excuse my now outdated pop culture references) of code. Recursion can be a difficult concept to wrap your head around when you first see it.

Recursive algorithms need to have at least one *base case* (in which the function does not call itself) and at least one *recursive case* (in which the function does call itself).

Some programming problems are simply easiest/cleanest to implement recursively. Beyond that, recursive code is beautiful once you get used to seeing it.

Here is an example of implementing the Fibonacci sequence in C using recursion:

```
int Fibonacci(int n)
{
   if ( n == 0 )
      return 0;
   else if ( n == 1 )
      return 1;
   else
      return ( Fibonacci(n-1) + Fibonacci(n-2) );
}
```

CS50 Section. Week 5. 9/27/16.

Tuesdays 4:00-5:30pm, CGIS S-040

> Nicholas Boucher nboucher@college.harvard.edu

Important links

- C language reference: https://reference.cs50.net/
- This week's material on Study50: Study50
- CS50 Discuss: https://cs50.harvard.edu/discuss
- CS50 Style Guide: https://manual.cs50.net/style/

Section Agenda

1. Suggestions from Last Pset
2. Review of Previous Concepts
3. Redirection
4. File I/O
5. Memory Management & malloc()
6. Pointers
7. Structures

Suggestions from Last Pset

Use Style50

Most of the general issues that I saw on psets could have been caught by style50. Everyone seems to be doing a good job of running check50, but do be sure to run style50 and correct any issues before submitting your code.

Required Multiline Comments

Do not forget to include the required comment stubs at the top of each file and before each function you write (except perhaps main). This is an entirely avoidable way to lose points on your psets. The following is excerpted from the Style Guide:

Comments at the Top of Each File

```
/**
 * hello.c
 *
 * David J. Malan
 * malan@harvard.edu
 *
 * Says hello to the world.
 */
```

Comments Before Each function

```
/**
 * Returns n^2 (n squared).
 */
int square(int n)
```

```
{
    return n * n;
}
```

Magic Numbers

Magic numbers are any numbers which are "hardcoded" into your code. As a general rule, we try to avoid these hardcoded constants at all costs, since it:

- makes code harder to read and understand
- makes it more difficult and error-prone to change these values later

Instead of using magic numbers, we should instead use #define statements at the top of each pertinent file. If the values you are trying to reference could be written more sensible in ASCII, then use `char`s. For example, on pset 2 it was much better to write `'A'` than `'65'`.

Review of Previous Concepts

Command Line Arguments

Use the following structure for `main`:

```
int main(int argc, string argv[])
{
    <code here>
}
```

`argc` is the argument count - the number of arguments passed to the program. `argc` is always greater than or equal to 1, since the name of the program is counted as the first argument.

`argv` is the argument vector - the actual values passed into the program. It is an array of strings, so it should be accessed using the array indexing syntax (i.e. `argv[n]`, where n is the index of the element you are trying to access). `argv[0]` is always the name of the program being executed.

Arrays

Arrays let you store a bunch of related pieces of data in a clean, organized way.

They can be created like this:

```
string my_classes[4];

my_classes[0] = "CS50";
my_classes[1] = "Ec 10";
my_classes[2] = "Expos 20";
my_classes[3] = "Math 55";
```

Or, you can use the following shortcut method if you know all of the data in advance:

```
int important_years[] = {1636, 1776, 2015};
```

Recursion

Recursive code is code in which a function makes a call to itself. Recursive algorithms need to have at least one *base case* (in which the function does not call itself) and at least one *recursive case* (in which the function does call itself).

Good example scenarios that are good candidates for recursion include a function to calculate the Fibonacci sequence and to exponentiate numbers.

Redirection

Redirection refers to the ability to modify where a program gets its input and where it puts its output. Most of the programs in C that we have dealt with thus far scan input from the command line and output to the console. Using the magic of `bash`, which is the name for the syntax we use on the "command line", we can change the input and output locations for an existing program.

First, we must define the terms Standard Input (`STD_IN`) to be the normal method of input specified at the start of a program, which is typically the command line by default. Likewise, Standard Output (`STD_OUT`) is the normal output location specified at the start of a program. Finally, Standard Error (`STD_ERR`) is the normal location to output error messages, which by default is the same as `STD_OUT`.

The following commands are used for redirection in `bash`:

- `>`: **output**; print the output of a program to a file instead of stdout e.g. `./hello > output.txt`
    - `>>`: append to an output file instead of writing over data

    - `2>` : this is just like the above, instead it will only print out error messages to a file
- `<` : **input**; use the contents of some file as input to a program e.g. `./hello < input.txt`
- `|` : **pipe**; take the output of one program and use it as input in another e.g. `echo "hello" | wc`

File I/O

Thus far, we have only printed to `STD_OUT`, but it is not much more difficult to write to actual files in C. To read a file, we must open it, read the data, probably process the data in some way, and then close the file. To write a file, we must open the file, write the data we want to write, and close the file.

The following program illustrates concrete examples of these tasks:

```
#include <stdio.h>

int main(void)
{
    // open file "input.txt" in read only mode
    FILE* in = fopen("input.txt", "r");

    // always make sure fopen() doesn't return NULL!
    if(in == NULL)
        return 1;
    else
    {
        // open file "output.txt" in write only mode
        FILE* out = fopen("output.txt", "w");

        // make sure you could open file
```

```
        if(out == NULL)
            return 2;

        // get character
        int c = fgetc(in);

        while(c != EOF)
        {
            // write chracter to output file
            fputc(c, out);
            c = fgetc(in);
        }

        // close files to avoid memory leaks!
        fclose(in);
        fclose(out);
    }
}
```

The key operations here are:

- `fopen` - opening files
- `fgetc` - reading a character
- `fputc` - writing a character
- `fclose` - closing a file

Additionally, we can use the following other file operation methods:

- `fread` - reading a specifically-sized chunk of data
- `fwrite` - writing a specifically-sized chunk of data.
- `fgets` - reading a null-terminated string
- `fputs` - writing a null-terminated string

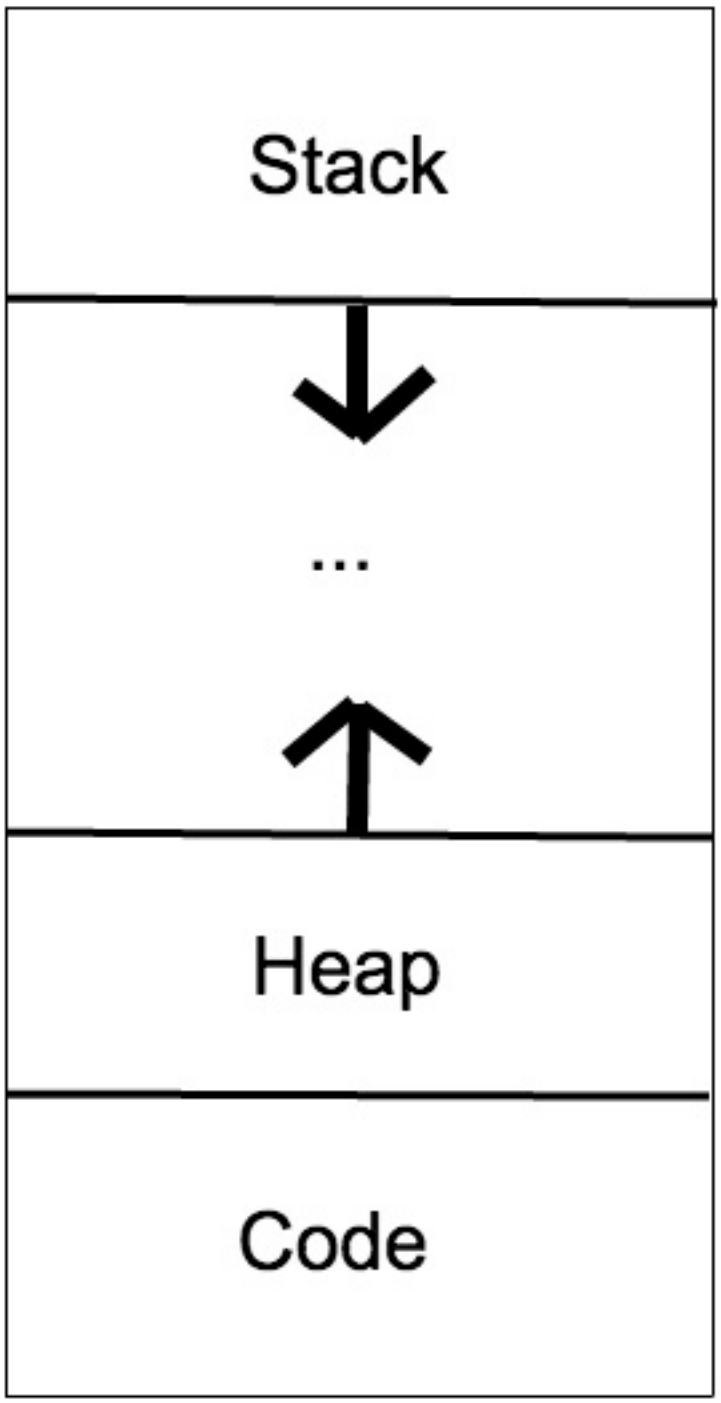
There are a couple key points to know with these functions. Regardless of what operation we are doing, we must first `fopen` the file. `fopen` returns a *pointer* to the file in memory, that is - it returns the location of

the file as interpreted by the program.

The second, and very important, point is that whenever we `fopen` a file we must also remember to `fclose` it when we are done. Forgetting to `fclose` constitutes a *memory leak*, which is both bad programming practice and can pose security risks to your program.

Memory Management & malloc()

Underneath your code, memory is laid out in a specific manner. The following chart visualizes this layout (*note - this diagram is simplified from reality*):



The **stack** is a contiguous block of memory set aside when a program starts running. Held in every stack frame is some metadata, any variables held in read-only memory, and most importantly, all local variables.

Each function that's called gets its own stack frame, meaning, if my `main()` function called a function `sum()`, then the stack frame for `sum()` would be on top of `main()`! We have effectively pushed `sum()`

on top of `main()`. If we ever want to get to the variables held in `main()` again, we'd first have to pop the `sum()` stack frame off by returning. Since each function has its own stack frame, its variables are protected from other functions and localized with their own scope.

Memory allocated during runtime is called dynamically allocated memory and it's held on the **heap**.

For now, just try to get the general idea of memory layout.

malloc()

Malloc is the name of a function ("Memory Allocate") where we can place data onto the heap. Remember how we got a *pointer* to the address of files when we `fopen`ed them earlier - that was an address living on the heap!

An example of this in code is:

```
int* ptr = malloc(sizeof(int));
```

Pointers

Watch this video: https://youtu.be/yOdd3uYC--A

Pointer are like addresses for location in memory. We use them to talk about *where* something important is.

There are three fundamental pointer operations, illustrated here with `a`, `b`, and `c`. Explain what each operation does.

Dereference

```
int x = *a;
```

Address of

```
int* x = &b;
```

Assignment

```
*c = 5;
```

CS50 Section. Week 6. 10/4/16.

Tuesdays 4-5:30 PM, CGIS S-040

> Nicholas Boucher nboucher@college.harvard.edu

Important links

- This week's material on Study50: Study50
- C language reference: https://reference.cs50.net/
- CS50 Discuss: https://cs50.harvard.edu/discuss
- CS50 Style Guide: https://manual.cs50.net/style/

Section Agenda

1. Notes from Past P-Set
2. Structures
3. Linked Lists
4. Hash Tables
5. Trees/Tries
6. Stacks/Queues & Huffman Coding

Notes from Past P-Set

Newlines

Beware of placing unnecessary newlines into your code. It is good to put one newline between functions, but you do not need more than one. Likewise, it is okay to separate out logical sections of your code within a function by using a newline. However, do not get in that habit of placing lots of blank lines in your code. There is no need for code to be longer than it has to be.

Many students are doing this:

```
int main(void)
{
  foo();
}



void foo()
{


  printf("My code is messy.\n");

}
```

However, this is how it should be formatted:

```
int main (void)
{
  foo();
}

void foo()
```

```
{
  printf("My code is clean.\n", );
}
```

Use `style50` before submitting your code.

Debugging

Many of the problems that are coming up over email and in office hours can be solved by simply tracing through the program with `debug50`. Try to get in the habit of tracing through the execution flow of your program before asking for help -- you will become a better programmer for it.

However, as always, do not hesitate to reach out over email if you do need help.

Structures

Structures are a tool that we can use to create new forms of data. These structures are comprised of a collection of other pre-existing types of data (such as `int`s and `char`s). Structures encapsulate data of different data types together because those different pieces of information comprise part of a larger unit.

We normally declare structures in the global scope do that they are accessible throughout the entire program.

Structures are declared using the `struct` keyword. This is an example of a structure declaration:

```
struct student
{
   char name[40];
   char house[20];
   int year;
};
```

The data type of the structure we have just created is struct student, so if we wish to create a variable of this type, we must:

```
struct student nicholas;
```

From there, we can assign the *fields* of that variable using the dot (.) operator:

```
strcpy(nicholas.name, "Nicholas Boucher");
strcpy(nicholas.house, "Mather");
nicholas.year = 2019;
```

Typedefing

If we do not want to have to type `struct` before the name of our structure every time we declare a variable of that type (e.g. `struct student nicholas;`), we can `typedef` the struct.

`Typedef`ing allows you to declare variables of any type you have designed as if it was an in-built, native type in C. Here is an example of what `typedef`ing looks like:

```
typedef struct student
{
```

```
    char name[40];
    char house[20];
    int year;
} student;
```

This will allow us to simply create a variable of this type such as:

```
student nicholas;
```

Linked Lists

So far, we have been forced to use arrays for any time we want to deal with a series of data. Arrays have the drawback that they are a fixed size and cannot change sizes after declaration. It thus becomes difficult to deal with series of an unknown size.

Introducing: Linked Lists. Linked lists allow us to store a virtually unlimited amount of data in a series with the ability to modify, insert, and remove data during runtime.

Linked lists are a clever combination of two recently introduced topics: pointers and structures. Linked lists work by creating a structure which include an element that is a pointer to something *of its own type*. That is, it points to another version of itself. We call each element within this sort of list a *node*. Consider the structure declared below.

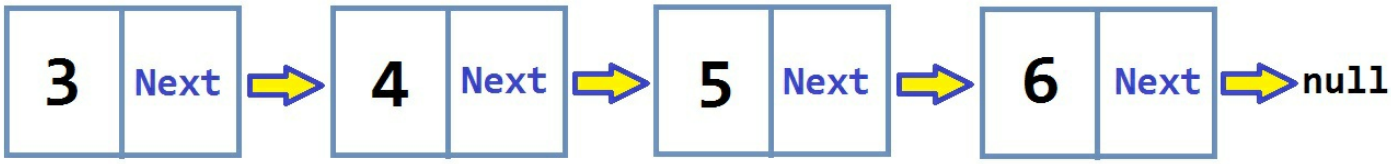
```
typedef struct node
{
    // just some form of data; could be a char* or whatever
    int val;
```

```
    // pointer to next node; have to include `struct`
    // since this is a recursive definition
    struct node *next;

}
node;
```

Each node contains a piece of data (or perhaps multiple pieces of data) and a pointer to the next node. When we reach an element whose `next == NULL`, then we know we have reached the end of the list.

The following visualization depicts the high-level picture of how a linked list containing numbers 3,4,5,6 may look:



Checking if an element is in a list

To find elements within a linked list, we must iterate over the array. This can be done using a `while` loop as follows:

```
bool find(node* ptr, int val)
{
  while(ptr != NULL)
```

```
    {
        if(ptr->val == val)
        {
            return true;
        }
        else
        {
            ptr = ptr->next;
        }
    }
    return false;
}
```

Searching through a list can also be done using recursion:

```
bool find(node* ptr, int val)
{
  if (ptr == NULL)
  {
    return false;
  }
  if (ptr->val == val)
  {
    return true;
  }
  else
  {
    return find(ptr->next, val);
  }
}
```

Adding Elements

To add an element to a linked list, you must follow these steps:

1. Malloc space for the new list element and populate the structure with data.
2. Change the `next` pointer of the element you would like to place this new element after to point to this new element.
3. Update the `next` pointer of the new element to point to the element that was previously after the element before the new element.

This is illustrated the following code, which inserts the value at the end of the existing list:

```
void insert(node* head, int val)
{
  node* element = malloc(sizeof(node));
  element->val = val;

  while (head->next != NULL)
  {
    head = head->next;
  }

  head->next = element;
  element->next = NULL;
}
```

Using the above code, we could create a linked list by continuing to add elements to that list:

```
int main(void)
{
  node* head = malloc(sizeof(node));
  head->val = 1;

  insert(head, 2);
  insert(head, 3);
  insert(head, 4);
```

```
  return 0;
}
```

Of course, you could do this more efficiently by adding elements "in one pass" while looping through the linked list without having to traverse the entire list each time you add an element.

Removing an Element

To remove an element:

1. Update the previous element's pointer to point to the following element.
2. `Free` the element you are removing.

This is illustrated in the following code, which removes one occurrence of a given value `val` from the list (returning true if it is removed and false if it is not):

```
/* Assumption: The element we are removing is not in the
 * first node of the array. You could design code that
  * would do this, but it would be more verbose. */
bool remove (node* head, int val)
{
  node* prev = head;
  head = head->next;

  while (head != NULL)
  {
    if (head->val == val) {
      prev->next = head->next;
      free(head);
      return true;
```

```
    }
    else {
      prev = head;
      head = head->next;
    }
  }
  return false;
}
```

Delete an Entire Linked list

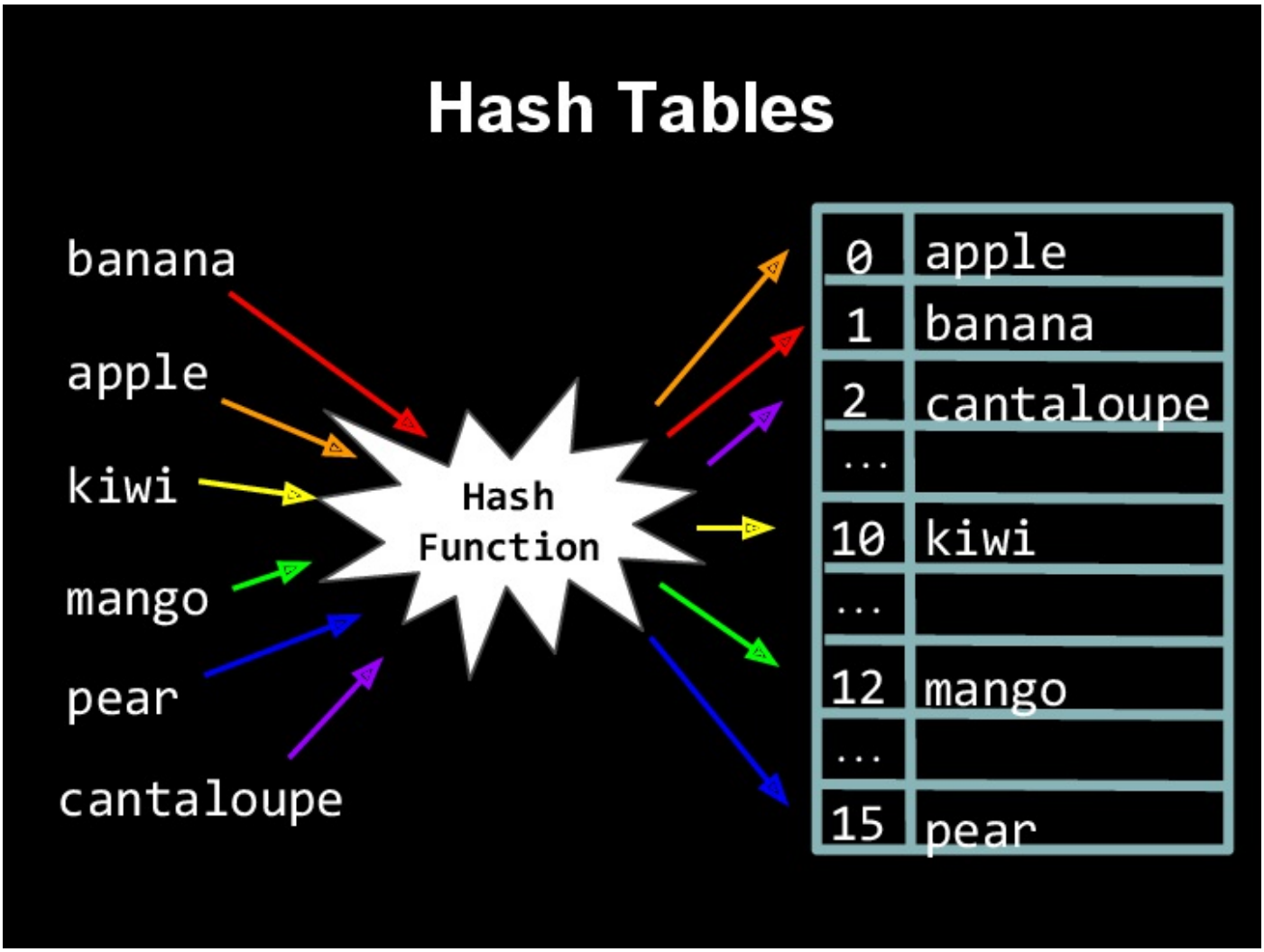
To delete an entire linked list, do the following:

```
void delete(node* head)
{
  while (head != NULL)
  {
    node* next = head->next;
    free(head);
    head = next;
  }
}
```

Hash Tables

A hash table is basically an associative array where the position of each element in the array is decided by a hash function. A hash function can be anything; for example, if placing strings into a hash table, your hash table could be of size 26 while your function distributes words based on their first ASCII character. Alternatively, you may choose to create a larger hash table and calculate where to put a word based on the

summation of a word's characters.



A hash function describes where to insert a word and, when necessary, where to look up a word. In an ideal world a hash table will provide constant time lookup (which is possible if there are no *collisions* when inserting!) Collisions occur when a hash function places two different elements into the same "slot" in an array.

Linked Lists are, for all operations, an upper-bound of O(n) time and a lower bound of Ω(1) time.

Hash tables are, for all operations, the same. The difference is in a hash table, in the real world the upper-bound effectively becomes O(n / k), which still theoretically reduces to O(n), though has clearly noticeable optimizations with respect to real world running time.

Trees/Tries

Trees are something that we have seen before - tries are a special kind of tree. They contain an array of pointers to the children of the current node.

See on-board demonstration in section, or for more information see the study50 links at the top of this document.

Stacks/Queues & Huffman Coding

Stacks and queues are data structures that are essentially special kinds of linked lists. A stack is First-In-Last-Out, or FILO. That is, the most recent element added (*pushed*) to a stack is the first element to be removed (*popped*). A queue is First-In-First-Out, or FIFO. That is, the first element added (*enqueued*) is the first element to be removed (*dequeued*).

Huffman coding is a special technique that we can use to compress data (think, for example, of .zip files on your computer). Huffman coding works by replacing the ASCII values of frequently used characters with shorter values and keeping a "key" table to associate these values with their actual meaning.

For more information on these topics, see the study50 links at the top of this document.