Midterm 1
Review

CS270 - Spring Semester 2019

General
Bring student ID card
• Must have it to check into lab

Seating
• Randomized seating chart
• Front rows
• Check when you enter the room

Exam
• No time limit, 100 points
• NO notes, calculators, or other aides
• Put your smartwatch / phone in your pocket!

Turing Machine
Mathematical model of a device that can perform any computation – Alan Turing (1937)
• ability to read/write symbols on an infinite "tape"
• state transitions, based on current state and symbol

Every computation can be performed by some Turing machine. (Turing’s thesis)

For more info about Turing machines, see http://www.wikipedia.org/wiki/Turing_machine
For more about Alan Turing, see http://www.turing.org.uk/turing/
Universal Turing Machine
A machine that can implement all Turing machines -- this is also a Turing machine!

- inputs: data, plus a description of computation (other TMs)

U is programmable -- so is a computer!
- instructions are part of the input data
- a computer can emulate a Universal Turing Machine

A computer is a universal computing device.

Introduction to Programming in C

Compilation vs. Interpretation
Different ways of translating high-level language

**Interpretation**
- interpreter = program that executes program statements
- generally one line/command at a time
- limited processing
- easy to debug, make changes, view intermediate results
- languages: BASIC, LISP, Perl, Java, Matlab, C-shell

**Compilation**
- translates statements into machine language
  - does not execute, but creates executable program
  - performs optimization over multiple statements
  - change requires recompilation
    - can be harder to debug, since executed code may be different
- languages: C, C++, Fortran, Pascal
Compiling a C Program

Entire mechanism is usually called the “compiler”

Preprocessor
- macro substitution
- conditional compilation
- "source-level" transformations
  → output is still C

Compiler
- generates object file
  → machine instructions

Linker
- combine object files (including libraries) into executable image

Bits, Data Types, and Operations

How do we represent data in a computer?

At the lowest level, a computer is an electronic machine.
- works by controlling the flow of electrons

Easy to recognize two conditions:
1. presence of a voltage – we’ll call this state "1"
2. absence of a voltage – we’ll call this state "0"

Could base state on value of voltage, but control and detection circuits more complex.
- compare turning on a light switch to measuring or regulating voltage
What kinds of data do we need to represent?

- Numbers – signed, unsigned, integers, floating point, complex, rational, irrational, ...
- Logical – true, false
- Text – characters, strings, ...
- Instructions (binary) – LC-3, x-86 ...
- Images – jpeg, gif, bmp, png ...
- Sound – mp3, wav ...
- …

Data type:
- representation and operations within the computer

We’ll start with numbers…

Unsigned Integers

Non-positional notation

- could represent a number ("5") with a string of ones ("11111")
- problems?

Weighted positional notation

- like decimal numbers: "329"
  - "3" is worth 300, because of its position, while "9" is only worth 9

\[
\begin{array}{c}
329 \\
10^2 & 10^1 & 10^0 \\
\end{array}
\]  
\[
\begin{array}{c}
101 \\
2^2 & 2^1 & 2^0 \\
\end{array}
\]

\[
3 \times 100 + 2 \times 10 + 9 \times 1 = 329 \\
1 \times 4 + 0 \times 2 + 1 \times 1 = 5
\]

Unsigned Binary Arithmetic

Base-2 addition – just like base-10!

- add from right to left, propagating carry

\[
\begin{array}{c}
10010 \\
+ 1001 \\
\hline 11011 \\
\end{array}
\]

\[
\begin{array}{c}
1011 \\
+ 111 \\
\hline 10000
\end{array}
\]

Subtraction, multiplication, division,...
Signed Integers

With $n$ bits, we have $2^n$ distinct values.
- assign about half to positive integers (1 through $2^{n-1}$) and about half to negative ($-2^{n-1}$ through -1)
- that leaves two values: one for 0, and one extra

Positive integers
- just like unsigned – zero in most significant (MS) bit

Negative integers: formats
- sign-magnitude – set MS bit to show negative, other bits are the same as unsigned
  \[10101 = -5\]
- one’s complement – flip every bit to represent negative
  \[11010 = -5\]
- in either case, MS bit indicates sign: 0=positive, 1=negative

Two’s Complement Representation

If number is positive or zero,
- normal binary representation, zeroes in upper bit(s)
If number is negative,
- start with positive number
- flip every bit (i.e., take the one’s complement)
- then add one

\[
\begin{align*}
00101 &= 5 \\
11010 &= \text{one’s comp} \\
+1 &= \text{add one} \\
11011 &= -5
\end{align*}
\]

Converting Binary (2’s C) to Decimal

1. If leading bit is one, take two’s complement to get a positive number.
2. Add powers of 2 that have “1” in the corresponding bit positions.
3. If original number was negative, add a minus sign.

\[
X = 01101000_{\text{two}} = 2^7 + 2^5 + 2^3 = 128 + 32 + 8 = 168_{\text{ten}}
\]

Assuming 8-bit 2’s complement numbers.
Sign Extension
To add two numbers, we must represent them with the same number of bits.
If we just pad with zeroes on the left:

<table>
<thead>
<tr>
<th>4-bit</th>
<th>8-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0100</td>
<td>00000100</td>
</tr>
<tr>
<td>1100</td>
<td>00001100</td>
</tr>
</tbody>
</table>

Instead, replicate the MS bit -- the sign bit:

<table>
<thead>
<tr>
<th>4-bit</th>
<th>8-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0100</td>
<td>00000100</td>
</tr>
<tr>
<td>1100</td>
<td>11111100</td>
</tr>
</tbody>
</table>

Overflow
If operands are too big, then sum cannot be represented as an n-bit 2's comp number.

```
 01000 (8) + 01001 (9) + 10111 (-9) = 10001 (-15)
```

We have overflow if:
• signs of both operands are the same, and
• sign of sum is different.

Another test -- easy for hardware:
• carry into MS bit does not equal carry out

Examples of Logical Operations
AND
• useful for clearing bits
  > AND with zero = 0
  > AND with one = no change

```
11000101 AND 00001111 = 00000101
```

OR
• useful for setting bits
  > OR with zero = no change
  > OR with one = 1

```
11000101 OR 00001111 = 11001111
```

NOT
• unary operation -- one argument
  • flips every bit

```
NOT 11000101 = 00111010
```
Hexadecimal Notation

It is often convenient to write binary (base-2) numbers as hexadecimal (base-16) numbers instead.

- fewer digits -- four bits per hex digit
- less error prone -- easy to corrupt long string of 1's and 0's

<table>
<thead>
<tr>
<th>Binary</th>
<th>Hex</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0010</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>0011</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>0100</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>0101</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>0110</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>0111</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>1000</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>1001</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>1010</td>
<td>A</td>
<td>10</td>
</tr>
<tr>
<td>1011</td>
<td>B</td>
<td>11</td>
</tr>
<tr>
<td>1100</td>
<td>C</td>
<td>12</td>
</tr>
<tr>
<td>1101</td>
<td>D</td>
<td>13</td>
</tr>
<tr>
<td>1110</td>
<td>E</td>
<td>14</td>
</tr>
<tr>
<td>1111</td>
<td>F</td>
<td>15</td>
</tr>
</tbody>
</table>

Floating Point Example

Single-precision IEEE floating point number:

1011111010000000000000000000000

- Sign is 1 – number is negative.
- Exponent field is 10111110 = 126 (decimal).
- Fraction is 0.100000000000… = 0.5 (decimal).

Value = -1.5 x 2^(126-127) = -1.5 x 2^-1 = -0.75.
**Data Types**

C has three basic data types

- **int** integer (at least 16 bits)
- **double** floating point (at least 32 bits)
- **char** character (at least 8 bits)

Exact size can vary, depending on processor

- **int** was supposed to be "natural" integer size; for LC-3, that's 16 bits
- **int** is 32 bits for most modern processors, **double** usually 64 bits

---

**Scope: Global and Local**

Where is the variable accessible?

- **Global**: accessed anywhere in program
- **Local**: only accessible in a particular region

Compiler infers scope from where variable is declared in the program

- programmer doesn't have to explicitly state

**Variable is local to the block in which it is declared**

- block defined by open and closed braces `{ }`
- can access variable declared in any "containing" block
- global variables are declared outside all blocks

---

**Arithmetic Operators**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Operation</th>
<th>Usage</th>
<th>Precedence</th>
<th>Assoc</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>multiply</td>
<td>x * y</td>
<td>6</td>
<td>l-to-r</td>
</tr>
<tr>
<td>/</td>
<td>divide</td>
<td>x / y</td>
<td>6</td>
<td>l-to-r</td>
</tr>
<tr>
<td>%</td>
<td>modulo</td>
<td>x % y</td>
<td>6</td>
<td>l-to-r</td>
</tr>
<tr>
<td>+</td>
<td>add</td>
<td>x + y</td>
<td>7</td>
<td>l-to-r</td>
</tr>
<tr>
<td>-</td>
<td>subtract</td>
<td>x - y</td>
<td>7</td>
<td>l-to-r</td>
</tr>
</tbody>
</table>

All associate left to right.

- * / % have higher precedence than + -.
- Full precedence chart on page 602 of textbook
Bitwise Operators

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Operation</th>
<th>Usage</th>
<th>Precedence</th>
<th>Assoc</th>
</tr>
</thead>
<tbody>
<tr>
<td>~</td>
<td>bitwise NOT</td>
<td>~x</td>
<td>4</td>
<td>r-to-l</td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>left shift</td>
<td>x &lt;&lt; y</td>
<td>8</td>
<td>l-to-r</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>right shift</td>
<td>x &gt;&gt; y</td>
<td>8</td>
<td>l-to-r</td>
</tr>
<tr>
<td>&amp;</td>
<td>bitwise AND</td>
<td>x &amp; y</td>
<td>11</td>
<td>l-to-r</td>
</tr>
<tr>
<td>^</td>
<td>bitwise XOR</td>
<td>x ^ y</td>
<td>12</td>
<td>l-to-r</td>
</tr>
<tr>
<td></td>
<td>bitwise OR</td>
<td>x</td>
<td>y</td>
<td>13</td>
</tr>
</tbody>
</table>

Operate on variables bit-by-bit.
• Like LC-3 AND and NOT instructions.
Shift operations are logical (not arithmetic).
• Operate on values -- neither operand is changed.

Control Structures

Conditional
• making a decision about which code to execute, based on evaluated expression
• if
• if-else
• switch

Iteration
• executing code multiple times, ending based on evaluated expression
• while
• for
• do-while
Functions

Function
Smaller, simpler, subcomponent of program
Provides abstraction
• hide low-level details
• give high-level structure to program, easier to understand overall program flow
• enables separable, independent development

C functions
• zero or multiple arguments passed in
• single result returned (optional)
• return value is always a particular type

In other languages, called procedures, subroutines, ...

Functions in C

Declaration (also called prototype)
int Factorial(int n);

Function call -- used in expression
a = x + Factorial(f + g);

1. evaluate arguments
2. execute function
3. use return value in expression
Function Definition

State type, name, types of arguments
- must match function declaration
- give name to each argument (doesn't have to match declaration)

```cpp
int Factorial(int n)
{
    int i;
    int result = 1;
    for (i = 1; i <= n; i++)
        result *= i;
    return result; // gives control back to calling function and returns value
}
```

Why Declaration?

Since function definition also includes return and argument types, why is declaration needed?

- **Use might be seen before definition.** Compiler needs to know return and arg types and number of arguments.
- **Definition might be in a different file, written by a different programmer.**
  - include a "header" file with function declarations only
  - compile separately, link together to make executable

Storing local variables for a function

For each function call
- A stack-frame ("activation record") is inserted ("pushed") in the runtime stack
- It holds
  - local variables,
  - arguments
  - values returned
- If the function is recursive, for each iteration inserts a stack-frame.
- When a function returns, the corresponding stack-frame is removed ("popped")
- When a function returns, its local variables are gone.
Implementing Functions: Overview

Activation record
- Information about each function, including arguments and local variables
- Stored on run-time stack

Calling function
- Push new activation record
- Copy values into arguments
- Call function
- Get result from stack

Called function
- Execute code
- Put result in activation record
- Pop activation record
- Return

Pointers and Arrays

We've seen examples of both of these in our LC-3 programs; now we'll see them in C.

Pointer
- Address of a variable in memory
- Allows us to indirectly access variables
  - In other words, we can talk about its address rather than its value

Array
- A list of values arranged sequentially in memory
- Example: a list of telephone numbers
- Expression `a[4]` refers to the 5th element of the array `a`
Address vs. Value

Sometimes we want to deal with the **address** of a memory location, rather than the **value** it contains.

Recall example from Chapter 6: adding a column of numbers.
- R2 contains address of first location.
- Read value, add to sum, and increment R2 until all numbers have been processed.

R2 is a pointer – it contains the address of data we’re interested in.

---

Another Need for Addresses

Consider the following function that’s supposed to swap the values of its arguments.

```c
void Swap(int firstVal, int secondVal) {
    int tempVal = firstVal;
    firstVal = secondVal;
    secondVal = tempVal;
}
```

---

Pointers in C

C lets us talk about and manipulate pointers as variables and in expressions.

**Declaration**

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

A pointer in C is always a pointer to a particular data type:
- int*, double*, char*, etc.

**Operators**

```c
*p  -- returns the value pointed to by p
&z  -- returns the address of variable z
```
Example

```c
int i;
int *ptr;
i = 4;
ptr = &i;
*ptr = *ptr + 1;
```

- Store the value 4 into the memory location associated with `i`.
- Store the address of `i` into the memory location associated with `ptr`.
- Read the contents of memory at the address stored in `ptr`.
- Store the result into memory at the address stored in `ptr`.

Pointers as Arguments

Passing a pointer into a function allows the function to read/change memory outside its activation record.

```c
void NewSwap(int *firstVal, int *secondVal)
{
    int tempVal = *firstVal;
    *firstVal = *secondVal;
    *secondVal = tempVal;
}
```

Arguments are integer pointers. Caller passes addresses of variables that it wants function to change.

Array Syntax

Declaration

```
type variable[num_elements];
```

- All array elements are of the same type.
- Number of elements must be known at compile-time.

Array Reference

```
variable[index];
```

- i-th element of array (starting with zero).
- No limit checking at compile-time or run-time.
Array as a Local Variable

Array elements are allocated as part of the activation record.

```c
int grid[10];
```

First element (`grid[0]`) is at lowest address of allocated space.

If `grid` is first variable allocated, then R5 will point to `grid[9]`.

---

Pointer Arithmetic

Address calculations depend on size of elements

- In our LC-3 code, we've been assuming one word per element.
  - e.g., to find 4th element, we add 4 to base address
  - It's ok, because we've only shown code for int and char, both of which take up one word.
  - If double, we'd have to add 8 to find address of 4th element.

C does size calculations under the covers, depending on size of item being pointed to:

```c
double x[10];
double *y = x;
*(y + 3) = 13;
```

allocates 20 words (2 per element)

same as `x[3]` -- base address plus 6 (3*sizeof(double))

---

Data Structures

- struct
- dynamic memory allocation
Data Structures
A data structure is a particular organization of data in memory.
• We want to group related items together.
• We want to organize these data bundles in a way that is convenient to program and efficient to execute.

An array is one kind of data structure.
In this chapter, we look at two more:
struct – directly supported by C
linked list – built from struct and dynamic allocation

Structures in C
A struct is a mechanism for grouping together related data items of different types.
• Recall that an array groups items of a single type.

Example:
We want to represent an airborne aircraft:
char flightNum[7];
int altitude;
int longitude;
int latitude;
int heading;
double airSpeed;

We can use a struct to group these data together for each plane.

Defining a Struct
We first need to define a new type for the compiler and tell it what our struct looks like.

struct flightType {
    char flightNum[7];  /* max 6 characters */
    int altitude;        /* in meters */
    int longitude;       /* in tenths of degrees */
    int latitude;        /* in tenths of degrees */
    int heading;         /* in tenths of degrees */
    double airSpeed;     /* in km/hr */
};

This tells the compiler how big our struct is and how the different data items ("members") are laid out in memory. But it does not allocate any memory.
Defining and Declaring at Once

You can both define and declare a struct at the same time.

```c
struct flightType {
    char flightNum[7]; /* max 6 characters */
    int altitude;      /* in meters */
    int longitude;     /* in tenths of degrees */
    int latitude;      /* in tenths of degrees */
    int heading;       /* in tenths of degrees */
    double airSpeed;   /* in km/hr */
} maverick;
```

And you can use the flightType name to declare other structs.

```c
struct flightType iceMan;
```

typedef

C provides a way to define a data type by giving a new name to a predefined type.

**Syntax:**

```c
typedef <type> <name>;
```

**Examples:**

```c
typedef int Color;
typedef struct flightType Flight;
typedef struct ab_type {
    int a;
    double b;
} ABGroup;
```

Using typedef

This gives us a way to make code more readable by giving application-specific names to types.

```c
Color pixels[500];
Flight plane1, plane2;
```

**Typical practice:**

Put typedef’s into a header file, and use type names in main program. If the definition of Color/Flight changes, you might not need to change the code in your main program file.
Array of Structs
Can declare an array of structs:
    Flight planes[100];

Each array element is a struct (7 words, in this case). To access member of a particular element:
    planes[34].altitude = 10000;

Because the [] and . operators are at the same precedence, and both associate left-to-right, this is the same as:
    (planes[34]).altitude = 10000;

Pointer to Struct
We can declare and create a pointer to a struct:
    Flight *planePtr;
    planePtr = &planes[34];

To access a member of the struct addressed by planePtr:
    (*planePtr).altitude = 10000;

Because the . operator has higher precedence than *, this is NOT the same as:
    *planePtr.altitude = 10000;

C provides special syntax for accessing a struct member through a pointer:
    planePtr->altitude = 10000;

Passing Structs as Arguments
Unlike an array, a struct is always passed by value into a function.

• This means the struct members are copied to the function’s activation record, and changes inside the function are not reflected in the calling routine’s copy.

Most of the time, you’ll want to pass a pointer to a struct.

    int Collide(Flight *planeA, Flight *planeB)
    {
        if (planeA->altitude == planeB->altitude) {
            ...%
        } else
            return 0;
    }
Dynamic Allocation
Suppose we want our weather program to handle a variable number of planes – as many as the user wants to enter.

- We can’t allocate an array, because we don’t know the maximum number of planes that might be required.
- Even if we do know the maximum number, it might be wasteful to allocate that much memory because most of the time only a few planes’ worth of data is needed.

Solution:
Allocate storage for data dynamically, as needed.

malloc
The Standard C Library provides a function for allocating memory at run-time: malloc.

    void *malloc(int numBytes);

It returns a generic pointer (void*) to a contiguous region of memory of the requested size (in bytes). The bytes are allocated from a region in memory called the heap.

- The run-time system keeps track of chunks of memory from the heap that have been allocated.

Example
int airbornePlanes;
Flight *planes;

    printf("How many planes are in the air?\n");
    scanf("%d", &airbornePlanes);

    planes = (Flight*) malloc(sizeof(Flight) * airbornePlanes);
    if (planes == NULL) {
        printf("Error in allocating the data array.\n");
        ...
    }
    planes[0].altitude = ...

Note: Can use array notation or pointer notation.
Free and Calloc
Once the data is no longer needed, it should be released back into the heap for later use.

This is done using the `free` function, passing it the same address that was returned by `malloc`.
```c
void free(void*);
```
If allocated data is not freed, the program might run out of heap memory and be unable to continue.

Sometimes we prefer to initialize allocated memory to zeros, `calloc` function does this:
```c
void *calloc(size_t count, size_t size);
```

The Linked List Data Structure
A linked list is an ordered collection of nodes, each of which contains some data, connected using pointers.

- Each node points to the next node in the list.
- The first node in the list is called the head.
- The last node in the list is called the tail.

Linked List vs. Array
A linked list can only be accessed sequentially. To find the 5th element, for instance, you must start from the head and follow the links through four other nodes.

Advantages of linked list:
- Dynamic size
- Easy to add additional nodes as needed
- Easy to add or remove nodes from the middle of the list (just add or redirect links)

Advantage of array:
- Can easily and quickly access arbitrary elements
Standard C Library

- I/O commands are not included as part of the C language.
- Instead, they are part of the **Standard C Library**.
  - A collection of functions and macros that must be implemented by any ANSI standard implementation.
  - Automatically linked with every executable.
  - Implementation depends on processor, operating system, etc., but interface is standard.
- Since they are not part of the language, compiler must be told about function interfaces.
- Standard **header files** are provided, which contain declarations of functions, variables, etc.

Basic I/O Functions

The standard I/O functions are declared in the `<stdio.h>` header file.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>putchar</td>
<td>Displays an ASCII character to the screen.</td>
</tr>
<tr>
<td>getchar</td>
<td>Reads an ASCII character from the keyboard.</td>
</tr>
<tr>
<td>printf</td>
<td>Displays a formatted string.</td>
</tr>
<tr>
<td>scanf</td>
<td>Reads a formatted string.</td>
</tr>
<tr>
<td>fopen</td>
<td>Open/create a file for I/O.</td>
</tr>
<tr>
<td>fprintf</td>
<td>Writes a formatted string to a file.</td>
</tr>
<tr>
<td>fscanf</td>
<td>Reads a formatted string from a file.</td>
</tr>
</tbody>
</table>
What is Recursion?

A *recursive function* is one that solves its task by calling itself on smaller pieces of data.

- Similar to recurrence function in mathematics.
- Like iteration -- can be used interchangeably; sometimes recursion results in a simpler solution.

**Example: Running sum (∑ \(n\))**

<table>
<thead>
<tr>
<th>Mathematical Definition</th>
<th>Recursive Function:</th>
</tr>
</thead>
<tbody>
<tr>
<td>RunningSum(1) = 1</td>
<td>int RunningSum(int n) {</td>
</tr>
<tr>
<td>RunningSum(n) = n + RunningSum(n-1)</td>
<td>if (n == 1)</td>
</tr>
<tr>
<td></td>
<td>return 1;</td>
</tr>
<tr>
<td></td>
<td>else</td>
</tr>
<tr>
<td></td>
<td>return n + RunningSum(n-1);</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
</tbody>
</table>

High-Level Example: Binary Search

Given a sorted set of exams, in alphabetical order, find the exam for a particular student.

1. Look at the exam halfway through the pile.
2. If it matches the name, we’re done;
   if it does not match, then...
3a. If the name is greater (alphabetically), then
    search the upper half of the stack.
3b. If the name is less than the halfway point, then
    search the lower half of the stack.
Binary Search: Pseudocode

Pseudocode is a way to describe algorithms without completely coding them in C.

```c
FindExam(studentName, start, end) {
    halfwayPoint = (end + start)/2;
    if (end < start)  
      ExamNotFound(); /* exam not in stack */
    else if (studentName == NameOfExam(halfwayPoint))  
      ExamFound(halfwayPoint); /* found exam! */
    else if (studentName < NameOfExam(halfwayPoint))  
      /* search lower half */
      FindExam(studentName, start, halfwayPoint - 1);
    else /* search upper half */
      FindExam(studentName, halfwayPoint + 1, end);
}
```