211: Computer Architecture
C programming, part 1

Topic:
- C Programming
Introduction to C

TAs will cover C in more details in sections

- Will also help you with machine/compilation logistics

Learning C

- Is no big deal; you already know Java
- Start by coding and testing small programs
- Learn how to use a debugger!
Why Learn C?

You are learning to be a computer scientist
- Languages are just tools
- Choose tool appropriate to the task

Current task: learning computer architecture and how programs written in high-level language runs on computers
- C closer to machine so easier to see mapping

It’s fun
Comparison with Java

Java Program

Byte Code (.class)

Java Virtual Machine

C Program

Compiled Code

gcc ...

javac ...

Hardware and Operating System
Anatomy of a C Program

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

char cMessage[] = "Hello\n";

/* Execution will start here */
int main (int argc, char **argv)
{
    int i, count;

    count = atoi(argv[1]);
    for (i = 0; i < count; i++) {
        printf("Hello %d\n", i);
    }
}
```
Comments

Begins with /* and ends with */

Can span multiple lines.

Cannot have a comment within a comment or string

- Example:
  “my/*don't print this*/string”
  would be printed as:
  my/*don't print this*/string

Comments are critical

- How much and where is an art
Variable Declarations

Variables are used as names for data items

Each variable has a type, which tells the compiler how the data is to be interpreted (and how much space it needs, etc.)

```c
int counter;
int startPoint;
```

Variables can be global or local

- global: declare outside scope of any function accessible from anywhere
- local: declare inside scope of a function accessible only from inside of the function
## Basic Data Types

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Data Type</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>individual characters</td>
<td>‘a’, ‘b’, ‘\t’, ‘\n’</td>
</tr>
<tr>
<td>int</td>
<td>integers</td>
<td>-15, 0, 35</td>
</tr>
<tr>
<td>float</td>
<td>real numbers</td>
<td>-23.6, 0, 4.56</td>
</tr>
<tr>
<td>double</td>
<td>real numbers with double precision</td>
<td>-23.6, 0, 4.56</td>
</tr>
</tbody>
</table>

### Modifiers

- **short, long**: control size/range of numbers
- **signed, unsigned**: include negative numbers or not
### Arithmetic Operators

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

* / % have higher precedence than + -

**Rule of thumb:** remember only a few precedence rules

* Use () for everything else*
Special Operators: ++ and --

Changes value of variable before (or after) its value is used in an expression

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Operation</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>++</td>
<td>postincrement</td>
<td>x++</td>
</tr>
<tr>
<td>--</td>
<td>postdecrement</td>
<td>x--</td>
</tr>
<tr>
<td>++</td>
<td>preincrement</td>
<td>++x</td>
</tr>
<tr>
<td>--</td>
<td>predecrement</td>
<td>--x</td>
</tr>
</tbody>
</table>

Pre: Increment/decrement variable before using its value
Post: Increment/decrement variable after using its value

*Be careful when using these operators!*
# Relational Operators

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Operation</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;</td>
<td>greater than</td>
<td>x &gt; y</td>
</tr>
<tr>
<td>&gt;=</td>
<td>greater than or equal</td>
<td>x &gt;= y</td>
</tr>
<tr>
<td>&lt;</td>
<td>less than</td>
<td>x &lt; y</td>
</tr>
<tr>
<td>&lt;=</td>
<td>less than or equal</td>
<td>x &lt;= y</td>
</tr>
<tr>
<td>==</td>
<td>equal</td>
<td>x == y</td>
</tr>
<tr>
<td>!=</td>
<td>not equal</td>
<td>x != y</td>
</tr>
</tbody>
</table>

Result is 1 (TRUE) or 0 (FALSE)

Don't confuse equality (==) with assignment (=)
# Logic Operators

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Operation</th>
<th>Usage</th>
<th>Assoc</th>
</tr>
</thead>
<tbody>
<tr>
<td>!</td>
<td>logical NOT</td>
<td>!x</td>
<td>r-to-l</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>logical AND</td>
<td>x &amp;&amp; y</td>
<td>l-to-r</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>logical OR</td>
</tr>
</tbody>
</table>

Treats entire variable (or value) as TRUE (non-zero) or FALSE (zero)
Result is 1 (TRUE) or 0 (FALSE)
## Bit Operators

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Operation</th>
<th>Usage</th>
<th>Assoc</th>
</tr>
</thead>
<tbody>
<tr>
<td>~</td>
<td>complement</td>
<td>~x</td>
<td>r-to-l</td>
</tr>
<tr>
<td>&amp;</td>
<td>bit AND</td>
<td>x &amp; y</td>
<td>l-to-r</td>
</tr>
<tr>
<td></td>
<td></td>
<td>bit OR</td>
<td>x</td>
</tr>
</tbody>
</table>

Operate on bits of variables or constants

For example:
- \(~0101 = 1010\)
- \(0101 \& 1010 = 0000\)
- \(0101 \mid 1010 = 1111\)
Expressions and Assignments

Expression = “a computation” with a result

- $(x + y) \times z$
- Be careful of type conversion!
  
  ```
  int x, z; float y;
  the result of the expression $(x + y) \times z$ will have what type?
  ```

Assignment

- $x = (x + y) \times z$;
- The assignment statement itself is an expression and has a value. In this case, it’s the value assigned to $x$. 

Control Statements

Conditional
- if else
- switch

Iteration (loops)
- while
- for
- do while

Specialized “go-to”
- break
- continue
The if Statement

if (expression-1) {statements-1}
else if (expression-2) {statements-2}
else if (expression-n-1) {statements-n-1}|
else {statements-n}

Evaluates expressions until find one with non-zero result
  ▪ executes corresponding statements

If all expressions evaluate to zero, executes statements for “else” branch
The switch Statement

switch(expression) {
    case const-1: statements-1;
    case const-2: statements-2;
    default: statements-n;
}

Evaluates expression; results must be integer

Finds 1st “case” with matching constant

- Executes corresponding statements
- Continue executing until encounter a break or end of switch statement

“default” always matches
The switch Statement (Example)

```c
int fork;
...
switch(fork) {
    case 1:
        printf("take left");
        break;
    case 2:
        printf("take right");
        break;
    case 3:
        printf("make U turn");
        break;
    default:
        printf("go straight");
}
```
## Loops

<table>
<thead>
<tr>
<th>Statement</th>
<th>Repeats set of statements</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>while (expression) {...}</code></td>
<td>zero or more times, while expression != 0, compute expression before each iteration</td>
</tr>
<tr>
<td><code>do {...} while (expression)</code></td>
<td>one or more times, while expression != 0, compute expression after each iteration</td>
</tr>
<tr>
<td><code>for (start-expression; cond-expression; update-expression) {...}</code></td>
<td>zero or more times while cond-expression != 0, compute start-expression before 1\textsuperscript{st} iteration, compute update-expression after each iteration</td>
</tr>
</tbody>
</table>
Specialized Go-to’s

break
- Force immediate exit from switch or loop
- Go-to statement immediately following switch/loop

continue
- Skip the rest of the computation in the current iteration of a loop
- Go-to evaluation of conditional expression for execution of next iteration
What does the following piece of code do?

```c
int index = 0;
int sum = 0;
while ((index >= 0) && (index <= 20))
{
    index += 1;
    if (index == 11) break;
    if ((index % 2) == 1) continue;
    sum = sum + index;
}
```
Functions

Similar to Java methods

Components:
- Name
- Return type
  - void if no return value
- Parameters
  - pass-by-value
- Body
  - Statements to be executed
  - return forces exits from function and resumes execution at statement immediately after function call

```c
int factorial(int n)
{
    int i;
    int result = 1;
    for (i = 1; i <= n; i++)
        result *= i;
    return result;
}
```
Function Calls

Function call as part of an expression

- $x + \text{factorial}(y)$
- Arguments evaluated before function call
  - Multiple arguments: no defined order or evaluation
- Returned value is used to compute expression
- Cannot have a void return type

Function call as a statement

- $\text{factorial}(y)$;
- Can have a void return type
- Returned value is discarded (if there is one)
Function Prototypes

Can declare functions without specifying implementation

- int factorial(int)
  - Can specify parameter names but don’t have to
  - This is called a function signature

Declarations allow functions to be “used” without having the implementation until link time (we’ll talk about linking later)

- Separate compilation
  - Functions implemented in different files
  - Functions in binary libraries

- Signatures are often given in header files
  - E.g., stdio.h gives the signatures for standard I/O functions
Input and Output

Variety of I/O functions in C Standard Library

#include <stdio.h>

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

- String contains characters to print and formatting directives for variables
- This call says to print the variable counter as a decimal integer, followed by a linefeed (\n)

scanf("%d", &startPoint);

- String contains formatting directives for parsing input
- This call says to read a decimal integer and assign it to the variable startPoint. (Don't worry about the & yet.)
C’s memory model matches the underlying (virtual) memory system

- Array of addressable bytes
C’s memory model matches the underlying (virtual) memory system
- Array of addressable bytes

Variables are simply names for contiguous sequences of bytes
- Number of bytes given by type of variable

Compiler translates names to addresses
- Typically maps to smallest address
- Will discuss in more detail later

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>int x</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>double y</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
</tr>
</thead>
</table>

27
Pointers

A pointer is just an address

Can have variables of type pointer

- Hold addresses as values
- Used for indirection

When declaring a pointer variable, need to declare the type of the data item the pointer will point to

- int *p; /* p will point to a int data item */

Pointer operators

- De-reference: *
  - *p gives the value stored at the address pointed to by p
- Address: &
  - &v gives the address of the variable v
int i;
int *ptr;

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

store the value 4 into the memory location associated with i
**Pointer Example**

```c
int i;
int *ptr;

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

store the address of `i` into the memory location associated with `ptr`
**Pointer Example**

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

- `i = 4;` sets the value of `i` to 4.
- `ptr = &i;` sets `ptr` to the address of `i`.
- `*ptr = *ptr + 1;` reads the contents of memory at the address stored in `ptr`, adds 1, and stores the result back into memory at the same address.

Diagram:
- `i` has a value of 5.
- `ptr` points to `i` at address 4300.
- Reading `ptr` (4300) results in 5.
- Writing `ptr` (4300) results in 6.
Example Use of Pointers

What does the following code produce? Why?

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

... int fv = 6, sv = 10;
Swap(fv, sv);
printf("Values: (%d, %d)\n", fv, sv);
```
Parameter Pass-by-Reference

Now what does the code produce? Why?

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

...
int fv = 6, sv = 10;
Swap(&fv, &sv);
printf("Values: (%d, %d)\n", fv, sv);
```
Null Pointer

Sometimes we want a pointer that points to nothing

In other words, we declare a pointer, but we’re not ready to actually point to something yet

```c
int *p;
p = NULL; /* p is a null pointer */
```

NULL is a predefined constant that contains a value that a non-null pointer should never hold

- Often, NULL = 0, because address 0 is not a legal address for most programs on most platforms