Class Announcements

- First programming project has been posted. Code base is available in Files/Projects on canvas. Deadline: Friday, March 24 (week after spring break). START WORKING ON PROJECT NOW!

  Late submission policy: 20% penalty for every 24 hour period after the submission deadline.

- Reminder: There are no late submissions for homeworks. Homeworks are preparation for exams. The two lowest grade homeworks out of all homeworks offered will be dropped. If you do not turn in a homework, you will receive 0 points on that homework.

- Reminder: Two exams will count out of three offered. The lowest exam grade will be dropped. If you do not take an exam, you will receive 0 points on that exam. There are no make-up exams.
Imperative Programming Languages

Imperative:
Sequence of state-changing actions.

- Manipulate an abstract machine with:
  1. Variables naming memory locations
  2. Arithmetic and logical operations
  3. Reference, evaluate, assign operations
  4. Explicit control flow statements
- Key operations: Assignment and “Goto”
- Fits the von Neumann architecture closely

Von Neumann Architecture

The given bus widths are examples only!
Run-time storage organization

Typical memory layout

<table>
<thead>
<tr>
<th>Code</th>
<th>Stack</th>
<th>Heap</th>
<th>Free Memory</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>St</td>
<td>H</td>
<td></td>
<td>S</td>
</tr>
<tr>
<td>o</td>
<td>at</td>
<td>e</td>
<td></td>
<td>t</td>
</tr>
<tr>
<td>d</td>
<td>ic</td>
<td>a</td>
<td></td>
<td>a</td>
</tr>
<tr>
<td>e</td>
<td></td>
<td>p</td>
<td></td>
<td>c</td>
</tr>
</tbody>
</table>

Logical Address Space

low                      high

The classical scheme

- allows both stack and heap maximal freedom
- code and static may be separate or intermingled

Will talk about this in more detail in a later lecture!
C: An Imperative Programming Language

**Expressions:** include procedure and function calls and assignments, and thus can have side-effects

**Control Structures:**

- if statements, with and without else clauses
- loops, with break and continue exits
  - \textbf{while} ( \textless expr\textgreater ) \textless stmt\textgreater
  - \textbf{do} \textless stmt\textgreater \textbf{while} ( \textless expr\textgreater )
  - \textbf{for} ( \textless expr\textgreater ; \textless expr\textgreater ; \textless expr\textgreater ) \textless stmt\textgreater
- switch statements
- goto with labelled branch targets
C Examples

while ( ( c = getchar() ) != EOF ) putchar(c);

for ( i = 0 ; s[i] == ' ' ; i++ );

for ( i = 0 ; i < n ; i++ ) {
    if ( a[i] < 0 ) continue; /*skip neg elems*/
    ....
}

c = getchar();
switch(c) {
    case '0': case '1': case '2': case '3':
    case '4': case '5': case '6': case '7':
    case '8': case '9':
        digit[c-'0']++;
        break;
    case ' ': case '
': case 't':
        delim++;
        break;
    ...
}

CS 314
Spring’23 lecture 12, page 5
Data Types in C

- Primitives: `char, int, float, double`
  no Boolean—any nonzero value is true

- Aggregates: arrays, structures

  ```c
  char a[10], b[2][10];
  ```

  ```c
  struct rectangle {
    struct point p1;
    struct point p2;
  }
  ```

- Enumerations: collection of sequenced values

- Pointers:
  ```c
  &i     address of i
  *p     dereferenced value of p
  p+1    pointer arithmetic
  ```

  ```c
  int *p, i;
  p = &i;
  *p = *p + 1;
  ```
## Basic Comparison (incomplete!)

<table>
<thead>
<tr>
<th>C</th>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Basic types:</strong></td>
<td><strong>Primitive types:</strong></td>
</tr>
<tr>
<td>int, double, char</td>
<td>int, double, char, <strong>boolean</strong></td>
</tr>
<tr>
<td><strong>Pointer (to a value)</strong></td>
<td><strong>Reference (to an object)</strong></td>
</tr>
<tr>
<td>array, <strong>struct</strong></td>
<td>array, object (<strong>class</strong>)</td>
</tr>
<tr>
<td><strong>Control flow:</strong></td>
<td><strong>Control flow</strong></td>
</tr>
<tr>
<td>if-else, switch, while,</td>
<td>if-else, switch, while,</td>
</tr>
<tr>
<td>break, continue, for, return, <strong>goto</strong></td>
<td>break, continue, for, return</td>
</tr>
<tr>
<td><strong>Logic operators:</strong></td>
<td><strong>Logic operators:</strong></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Logical comparisons:</strong></td>
<td><strong>Logical comparisons:</strong></td>
</tr>
<tr>
<td>==, !=</td>
<td>==, !=</td>
</tr>
<tr>
<td><strong>Numeric comparisons:</strong></td>
<td><strong>Numeric comparisons:</strong></td>
</tr>
<tr>
<td>&lt;&gt;, &lt;=, &gt;=</td>
<td>&lt;&gt;, &lt;=, &gt;=</td>
</tr>
<tr>
<td>string as <strong>char * array</strong></td>
<td><strong>String</strong> as an object</td>
</tr>
</tbody>
</table>
Compile and Run a C program

test.c:
#include <stdio.h>

int main(void)
{
    int x, y;

    printf("First number:\n"); scanf("%d", &x);
    printf("Second number:\n"); scanf("%d", &y);

    printf("%d+%d = %d\n", x, y, x+y);
    printf("%d-%d = %d\n", x, y, x-y);
    printf("%d*%d = %d\n", x, y, x*y);

    return 0;
}

gcc test.c: calls the GNU C compiler, and
    generates executable a.out
./a.out runs the executable
gcc -o run test.c compiles program, and
    generates executable run
gcc -g test.c generates a.out with debugging info
gdb a.out run debugger on a.out;
    online documentation man gdb
Compile and Run a C program

> gcc test.c
> a.out

First number:
4

Second number:
12

4+12 = 16
4-12 = -8
4*12 = 48
>

START PROGRAMMING IN C NOW!
Debugging C programs

```
rhea% gdb a.out
(gdb) list
1     #include <stdio.h>
2     int main(void)
3     {
4         int x, y;
5         printf("First number:\n"); scanf("%d", &x);
6         printf("Second number:\n"); scanf("%d", &y);
7         printf("%d+%d = %d\n", x, y, x+y);
(gdb) break 7
Breakpoint 1 at 0x1052c: file test.c, line 7.
(gdb) run
Starting program: /.../a.out
First number:
4
Second number:
12
Breakpoint 1, main () at test.c:7
7         printf("%d+%d = %d\n", x, y, x+y);
(gdb) print x
$1 = 4
(gdb) print y
$2 = 12
(gdb) cont
Continuing.
4+12 = 16
4-12 = -8
4*12 = 48
Program exited normally.
(gdb) quit
```
Pointers in C

**Pointer**: Variable whose R-values (content) is the L-value (address) of a variable

- “address-of” operator &
- dereference (“content-of”) operator *

```c
int *p, x;
p = &x;
*p = 5;
x = 12;
```
Pointers in C

- Pointers can point to pointer variables (multi-level pointers)

```
int *p, x;
int **r;
p = &x;
*p = 5;
r = &p;
**r = 10;
```
Example: Singly-linked list

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

/* TYPE DEFINITION */
typedef struct cell listcell;
struct cell
{
    int num;
    listcell *next;
};

/* GLOBAL VARIABLES */
listcell *head, *new_cell, *current_cell;
```
Example: Singly-linked list

```c
int main (void)
{
    int j;

    /* CREATE FIRST LIST ELEMENT */
    head = (listcell *) malloc(sizeof(listcell));
    head->num = 1;
    head->next = NULL;

    /* CREATE 9 MORE ELEMENTS */
    for (j=2; j<=10; j++) {
        new_cell = (listcell *) malloc(sizeof(listcell));
        new_cell->num = j;
        new_cell->next = head;
        head = new_cell;
    }

    /* PRINT ALL ELEMENTS */
    for (current_cell = head;
         current_cell != NULL;
         current_cell = current_cell->next)
        printf("%d ", current_cell->num);

    printf("\n");
}
```
Example: Singly-linked list

int main (void)
{
    int j;

    /* CREATE FIRST LIST ELEMENT */
    head = (listcell *) malloc(sizeof(listcell));
    head->num = 1;
    head->next = NULL;

    /* CREATE 9 MORE ELEMENTS */
    for (j=2; j<=10; j++) {
        new_cell = (listcell *) malloc(sizeof(listcell));
        new_cell->num = j;
        new_cell->next = head;
        head = new_cell;
    }
    /* *** HERE *** */
Example: Singly-linked list

- What is the output of the program
- Where do the cell objects get allocated?
Review: Stack vs. Heap

Stack:

- Procedure activations, statically allocated local variables, parameter values
- Lifetime same as subroutine in which variables are declared
- Stack frame is pushed with each invocation of a subroutine, and popped after subroutine exit

Heap:

- Dynamically allocated data structures, whose size may not be known in advance
- Lifetime extends beyond subroutine in which they are created
- Must be explicitly freed or garbage collected
Next Lecture

Things to do:
Start working on the project!

Read Scott: Chap. 3.1 - 3.4; 8.3; ALSU Chap. 7.1 - 7.3

Next time:

- Dynamic memory allocation: malloc and free
- What can go wrong
- Static and dynamic scoping
- Activation records, maintaining a lexically scoped runtime environment using access links and displays.
- Parameter passing styles and their implementations.