Class Information

REMINDERS

• Fourth homework will be posted next Tuesday.

• First project has been posted. Look at it NOW! Deadline: Thursday, October 22, at 11:59pm.

• Midterm: Tuesday, October 27, in class.
Review: 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
Run-time storage organization

Typical memory layout

Logical Address Space

<table>
<thead>
<tr>
<th>Code</th>
<th>Static</th>
<th>Heap</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>low</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>high</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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
  
  ```
  while ( <expr> ) <stmt>
  do <stmt> while ( <expr> )
  for ( <expr> ; <expr> ; <expr> ) <stmt>
  ```

- switch statements

- goto with labelled branch targets
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 '	':
        delim++;
        break;
    ...
}

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

- Enumerations: collection of sequenced values

- Pointers:
  
  ```c
  int *p, i;
  p = &i;
  *p = *p + 1;
  ```
<table>
<thead>
<tr>
<th>C</th>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic types:</td>
<td>Primitive types:</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>Aggregates:</td>
<td>Aggregates:</td>
</tr>
<tr>
<td>array, <strong>struct</strong></td>
<td>array, object (<strong>class</strong>)</td>
</tr>
<tr>
<td>Control flow:</td>
<td>Control flow</td>
</tr>
<tr>
<td>if-else, switch, while,</td>
<td>if-else, switch, while,</td>
</tr>
<tr>
<td>break, continue, for, return, goto</td>
<td>break, continue, for, return</td>
</tr>
<tr>
<td>Logic operators:</td>
<td>Logic operators:</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Logical comparisons:</td>
<td>Logical comparisons:</td>
</tr>
<tr>
<td>==, !=</td>
<td>==, !=</td>
</tr>
<tr>
<td>Numeric comparisons:</td>
<td>Numeric comparisons:</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:**

```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;
```

![Diagram of pointer operations](image)
Pointers in C

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

```c
int *p, x;
int **r;

r = &x;
p = &x;
*p = 5;
**r = 10;
r = &p;
```

```plaintext
int *p, x;
int **r;

p = &x;
*p = 5;
**r = 10;
r = &p;
```
# Pointers in C vs. References in Java

<table>
<thead>
<tr>
<th>C</th>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td>Need explicit dereference operator *</td>
<td>are implicitly dereferenced</td>
</tr>
<tr>
<td>Can mutate R-value of pointer</td>
<td>Cannot mutate R-value</td>
</tr>
<tr>
<td>through pointer arithmetic <code>p=p+1</code></td>
<td>Casting just satisfies the type checker; no type conversion</td>
</tr>
<tr>
<td>Casting means type conversion</td>
<td>Casting just satisfies the type checker; no type conversion</td>
</tr>
<tr>
<td>Special relation to arrays</td>
<td>No special relation to arrays</td>
</tr>
</tbody>
</table>

CS 314  
lecture 11, fall’15, page 13
**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;
```

![Singly-linked list diagram]
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
Maintaining Free List

- **allocate**: continuous block of memory; remove space from **free list** (here: singly-linked list).

- **free**: return to free list after coalescing with adjacent free storage (if possible); may initiate compaction.
Heap Storage

void * malloc(size_t n) (defined in stdlib.h)

• returns pointer to block of contiguous storage of \( n \) bytes on the heap, if possible

• returns NULL pointer if not enough memory is available

⇒ you should check for ==NULL after each malloc

NOTE: we didn’t do this in the example!

• to allocate storage of a desired type, call malloc with the needed size in bytes, and then cast the return pointer to the desired type

\[
\text{head} = (\text{listcell} *) \text{malloc(sizeof(listcell))};
\]

void free(void *ptr) (defined in stdlib.h)

• data structure that ptr points to is released, i.e., returned to the free memory and may be (partially) reused by a subsequent malloc.
Next Lecture

Things to do:

- Work on the project!

Read Scott: Chap. 3.1 - 3.4; 8.1 - 8.2; ALSU Chap. 7.1 - 7.3

Next time:

- Procedure abstractions; run time stack; scoping.