ANNOUNCEMENTS

- Third homework due TODAY at 11:59pm. Extension?
- First project has been posted, due Monday October 23, 11:59pm.
- **Midterm exam**: Friday, October 27, in class.
- Don’t forget to work on your C and Linux skills (ilab).
- **Final exam**: Wednesday, December 20, 4:00-7:00pm.
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!
## Basic Comparison (incomplete!)

<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,</td>
<td>break, continue, for, return</td>
</tr>
<tr>
<td>return, goto</td>
<td></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

```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);
}
```

```
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)

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

p = &x;
*p = 5;

r = &p;
**r = 10;
```
# 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 p=p+1</td>
<td></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>
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;
```

![Diagram of Singly-linked list](image)
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

```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;
    }
    /* *** 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**: contiguous 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

```c
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
  ```c
  head = (listcell *) malloc(sizeof(listcell));
  ```

```c
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`. 
Problems with Explicit Control of Heap

- **Dangling references**
  - Storage pointed to is freed (put on free list), but there are still pointer values (or reference) that allow access the “freed” storage
  - Able to access storage whose values are not meaningful

- **Garbage**
  - Objects in heap that cannot be accessed by the program any more
  - Example
    ```c
    int *x, *y;
    x = (int *) malloc(sizeof(int));
    y = (int *) malloc(sizeof(int));
    x = y;
    ```

- **Memory leaks**
  - Failure to release (reclaim) memory storage builds up over time
Example: Singly-linked list

Let’s deallocate, i.e., free all list elements.

```c
#include "list.h"
/* GLOBAL VARIABLES */
    listcell *head, *new_cell, *current_cell;
int main (void) {
    int j;

    /* CREATE FIRST LIST ELEMENT */
    ...

    /* CREATE 9 MORE ELEMENTS */
    ...

    /* DEALLOCATE LIST */
    for (current_cell = head;
        current_cell != NULL;
        current_cell = current_cell->next)
        free(current_cell);
}
    ...
}
```

Does this work?
Next Lecture

Things to do:

Keep working on the project!

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

Next time:

- More about pointers
- Dynamic vs. static scoping
- Runtime environment
- access links and control links management