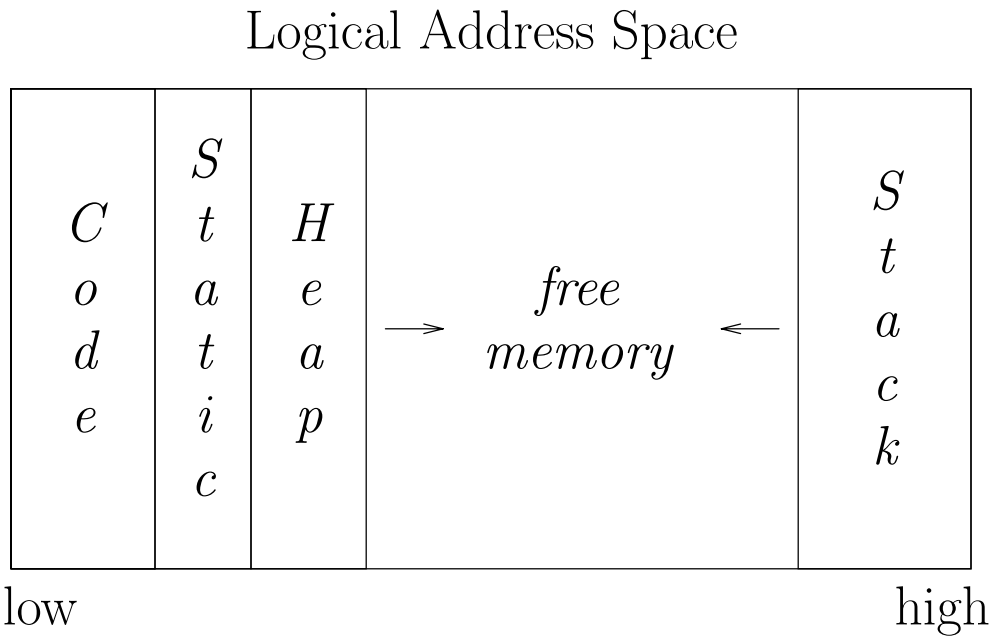
Class Information

- Fourth homework will be due on Friday.
- Midterm exam: Friday, March 14, in class, closed book, closed notes.

Review: Run-time storage organization

Typical memory layout



The classical scheme

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

Review: What went wrong?

"Aliasing" and freeing memory

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
  int *a = NULL; int *b = NULL; int *c = NULL;

  a = (int *) malloc(sizeof(int));
  b = a; *a = 12;
  printf("%x %x: %d\n", &a, a, *a);
  printf("%x %x: %d\n", &b, b, *b);
  free(a);
  printf("%x %x: %d\n", &b, b, *b);

  c = (int *) malloc(sizeof(int));
  *c = 10;
  printf("%x %x: %d\n", &c, c, *c);
  printf("%x %x: %d\n", &b, b, *b);
}
```

> a.out

effff60c 209d0: 12
effff608 209d0: 12
effff608 209d0: 12
effff604 209d0: 10
effff608 209d0: 10

What went wrong?

Use a subroutine to create an object

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

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

listcell *head = NULL;

listcell *create_listcell() {
  listcell new;
  new.num = -1; new.next = NULL;
  return &new;
}

int main (void) {
  head = create_listcell();
  printf("head->num = %d\n", head->num);
}
```

> gcc stack.c
stack.c: In function 'create_listcell':
stack.c:17: warning: function returns address of local variable
> ./a.out

head→num = -1

What went wrong?

Use a subroutine to create an object: malloc

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

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

listcell *head = NULL;

listcell *create_listcell() {
  listcell *new;
  new = (listcell *) malloc(sizeof(listcell));
  new->num = -1; new->next = NULL;
  return new;
}

int main (void) {
  head = create_listcell();
  printf("head->num = %d\n", head->num);
}
```

> gcc heap.c
> ./a.out

head→num = -1

Pointers and Arrays in C

Pointers and arrays are similar in C:

- array name is pointer to a[0]:

  after

  ```
  int a[10];
  int *pa;
  pa = &a[0];
  ```

  pa and a have the same semantics

- pointer arithmetic is array indexing

  pa+1 and a+1 point to a[1]

- exception: an array name is not a variable

  a++ is ILLEGAL

  a=pa is ILLEGAL (pa=a is LEGAL!)

Review - Names, Bindings, and Memory

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

What's in a name? — each name "means" something!

- denotes a programming language construct
- has associated "attributes" (e.g.: type, memory location, read/write permission, storage class, access restrictions, etc.)
- has a meaning, i.e., represents a semantic object (e.g.: a type description, an integer value, a function value, a memory address, etc.)

Review - Names, Bindings, and Memory

Binding – association of a name with an attribute (e.g., a name and a memory location, a function name and its "meaning", a name and a value)

- **Compile time** – during compilation process – static (e.g.: macro expansion, type definitions)
- **Link time** – separately compiled modules/files are joined together by the linker (e.g., adding the standard library routines for I/O (stdio.h), external variables)
- **Run time** – when program executes – dynamic

Compiler needs bindings to know meaning of names during translation and execution.

Binding Times - Choices

- **Early binding** times – more efficient (faster) at runtime
- **Late binding** times – more flexible (postpone binding decision until more "information" is available)
- Examples of static binding (early):
  - functions in C
  - types in C
- Examples of dynamic binding (late):
  - method calls in Java or virtual function calls in C++
  - actual types of objects pointed to by a Java reference variable (class hierarchy)
  - dynamic typing in Scheme

Note: dynamic linking is somewhat inbetween static and dynamic binding; the funtion signature has to be known (static), but the implementation is linked and loaded at run time (dynamic).

How to Maintain Bindings

- symbol table: maintained by compiler during compilation
  names ⇒ attributes
- environment: maintained by compiler generated code during program execution
  names ⇒ memory locations
- memory: maps memory locations to values
  memory locations ⇒ values

Questions

- How long do bindings for a name hold in a program?
- What initiates a binding?
- What ends a binding?

Scope Example

Block Structures Programming Languages

```
program L;
   var n: char;     {n declared in L}

   procedure W;
   begin
     write(n);      {n referenced in W}
   end;

   procedure D;
      var n: char; {n declared in D}
   begin
      n:= 'D';      {n referenced in D}
      W
   end;

begin
  n:= 'L';          {n referenced in L}
  W;
  D
end.
```

Lexical Scope

- Non-local variables are associated with declarations at *compile* time
- Find the smallest block syntactically enclosing the reference and containing a declaration of the variable

- Example:
  - The reference to `n` in `W` is associated with the declaration of `n` in `L`
  - The output is?

Dynamic Scope

- Non-local variables are associated with declarations at *run* time
- Find the most recent, currently active run-time stack frame containing a declaration of the variable
- Example:
  - The reference to `n` in `W` is associated with two different declarations at two different times
  - The output is?

Next Lecture

Things to do:

Continue working on the project. Due Friday March 7!

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

Next time:

- Activation trees, activation records, maintaining a lexically scoped runtime environment.
- Parameter passing styles and their implementation.