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!

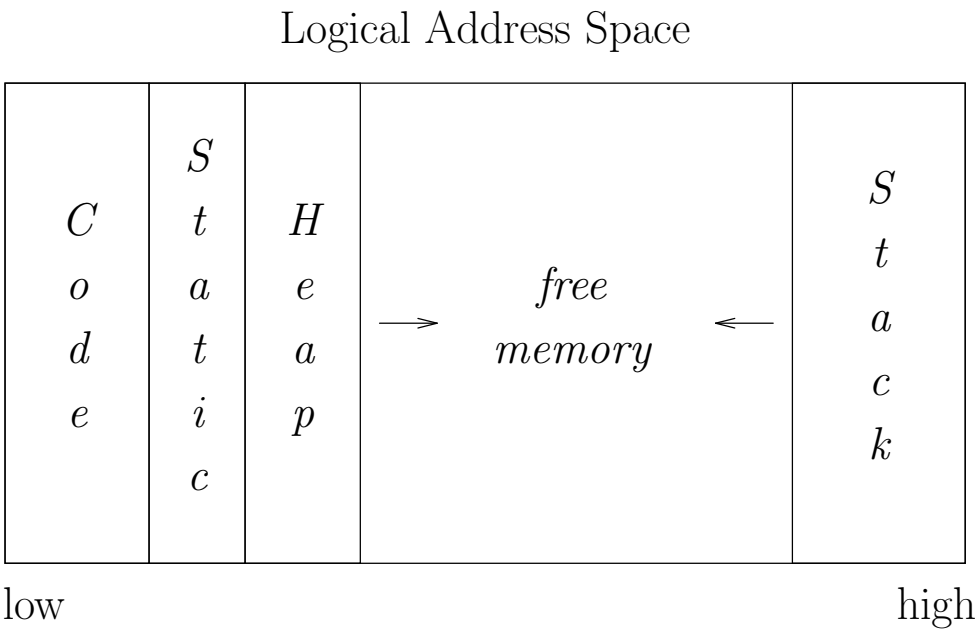
  Late submission policy: 20% penalty for every 24 hour period after the submission deadline.

- Warning grades due March 10 (only W1 since we do not take attendance)

- Fourth homework: Will cover pointers and scoping

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: Stack vs. Heap

Stack:

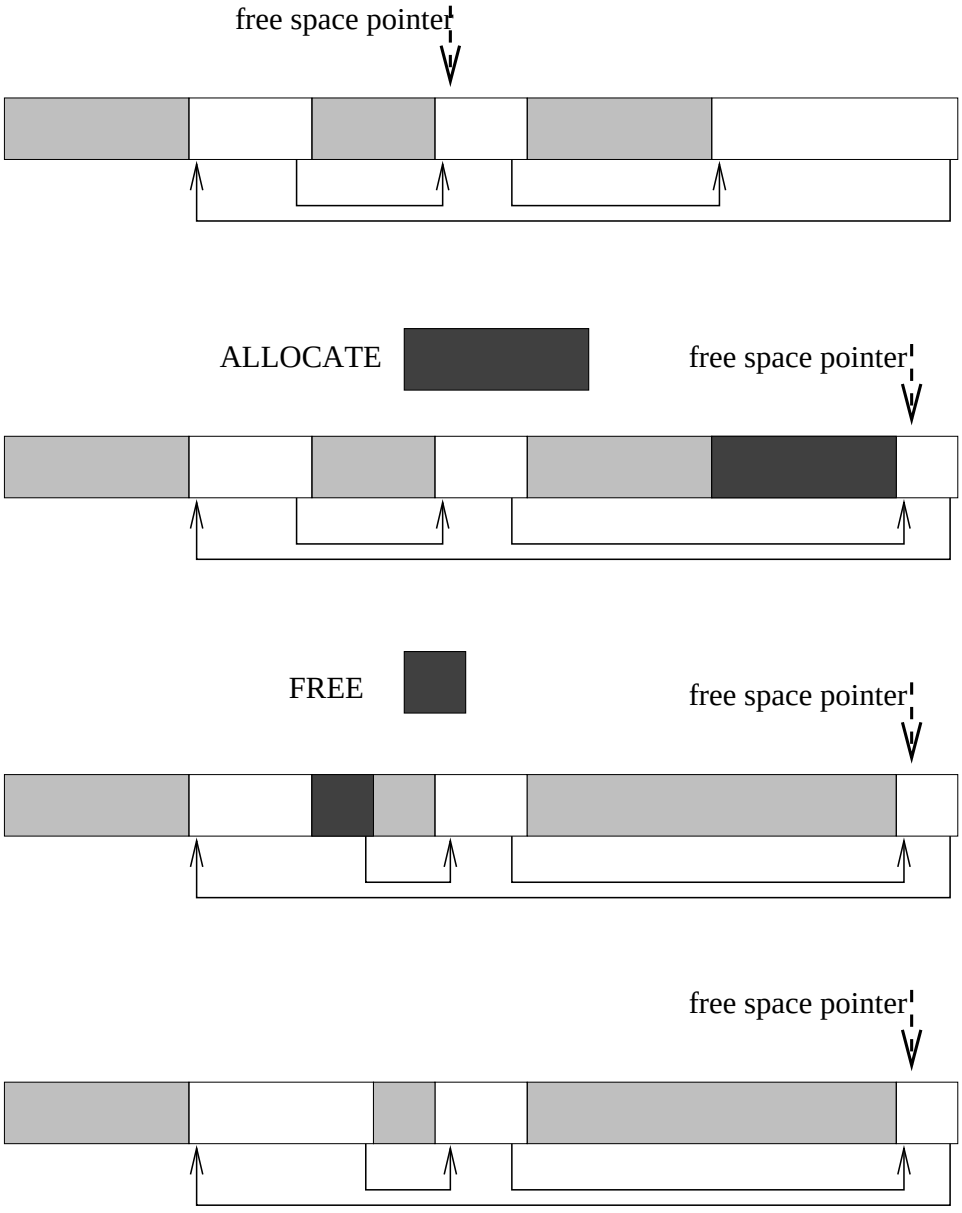
- Procedure activations, statically allocated local variables, parameter values
- Lifetime same as subroutine in which variables are declared
- Stack frame is pushed with each invocation of a subroutine, and poped after subroutine exit

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

`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
  head = (listcell *) 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`.

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

```
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

What went wrong?

Uninitialized variables and "dangerous" casting

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

  *a = 12;
  printf("%x %x: %d\n", &a, a, *a);

  a = (int *) 12;
  printf("%d\n", *a);
}
```

> a.out
effff60c effff68c: 12
Segmentation fault (core dumped)

Note: Segmentation faults result in the generation of a core file which can be rather large. Don't forget to delete it.

What went wrong?

That's better!

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

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

> a.out

effff60c 20900: 12

What went wrong?

The machine or compiler must be broken!?!!

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

int main (void)
{
  int i;

  char *string = "Hello, how     are   you   today.";
  printf("\n%s\n", string);

  for (i=0; string[i] != '.'; i++) {
    if (string[i] = ' ')
      for (; string[i] = ' ';i++);
    printf("%c", string[i]);
  }
  printf(".\n");
}
```

> a.out

Hello, how are you today.

Segmentation fault (core dumped)

What went wrong?

"=" is not the same as "=="

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

int main (void)
{
  int i;

  char *string = "Hello, how     are   you   today.";
  printf("\n%s\n", string);

  for (i=0; string[i] != '.'; i++) {
    if (string[i] == ' ')
      for (; string[i] == ' ';i++);
    printf("%c", string[i]);
  }
  printf(".\n");
}
```

> a.out

Hello, how are you today.

Hello,howareyoutoday.

What went wrong?

Aliasing: Two or more "names" for the same memory location

"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

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!)

Next Lecture

Things to do:

Keep on working on the project!

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

Next time:

- Dynamic and lexical scoping
- Runtime environment: stack frame, control links, access link
- Parameter passing styles and their implementations.