Notes on Obj-C Foundation classes

- NSArray
- NSNumber
- NSDictionary
- NSSet
- NSData
- NSFileManager

} and their Mutable counterparts

- suppose we have an arbitrary data class (object), like our data_t:

```
@interface data_t : NSObject
{
    int num;
    NSString *name;
}
```

and its constructors, accessors/mutators, etc.

- We want to put these on a list, e.g., like lab 12
- You could use your own, or use NSArray or NSMutableArray
- what's the difference:
  - NSArray, once initialized with data, cannot be altered - a const list
  - can add/delete to NSMutableArray

- allocation:

```
NSMutableArray *list =
[[NSMutableArray alloc] init];
```

- to add to list, can use:

addObject:

addObjectsFromArray:

insertObject: atIndex:

insertObjects: atIndexes:

- so to redo lab 12 with NSMutableArray:

```
while(scanf("%s %d", nam, &num) == 2) {
    str = [[NSString alloc]
        initWithUTF8String: nam];

    [list addObject:
        [(data_t *)[data_t alloc]
            init: str: num]];
    [str release];
}
```

- addObject: is like our push_back function
- what about iterating thru list?
- foundation classes provide NSEnumerator which acts just like our list_t list iterator

```
NSEnumerator *litr = nil;
litr = [list objectEnumerator];
while(data = [litr nextObject])
    [data fprint: stderr];
```

—— OR ——

```
for(litr = [list objectEnumerator],
    data = [litr nextObject];
    data != nil;
    data = [litr nextObject])
    [data fprint: stderr];
```

- What about our
  [list empty] call?

[list count] returns no.
of items on the list,
so if 0 is returned, list
empty

- to delete all objects:

```
while([list count])
    [list removeObjectAtIndex:0];
```

(popping from front)

- Suppose we just want to look at first on the list:

```
[[list objectAtIndex: 0]
    print: stderr];
```

like our old [list data]

- But wait! there's more ...

- suppose I want to sort the list:

```
NSArray *sortedList = nil;
```

(not mutable)

```
sortedList =
    [list sortedArrayUsingSelector:
        @selector(compareByName:)];
```

What's on the list must provide this method

- our data_t class must provide a method to compare two data_t objects

- Let's write two: (in data.h)

```
- (NSComparisonResult)
    compareByName:(data_t *)rhs;
- (NSComparisonResult)
    compareByNum:(data_t *)rhs;
```

```
-(NSComparisonResult)
    compareByNum:(data_t *)rhs
{
        accessors
  if([self num] < [rhs num])
      return NSOrderedAscending;
  else
  if([self num] < [rhs num])
      return NSOrderedDescending;
  else
      return NSOrderedSame;
}
```

```
-(NSComparisonResult)
    compareByName:(data_t *)rhs
{
  return
  [[self name] compare:[rhs name]];
}
```

- this one's easier because NSString already has a compare: method so we just use it

- point is that objects should provide own compare: methods

- in C++ you would provide

operator==( )

a function so that you could write

if (data1 == data2)

- Another detail that has bugged me:

- our pixel vector uses

  drgb_t

  a double array to store R, G, B values that we then later need to convert to unsigned char for I/O purposes

- Foundation Framework provides NSNumber that does the conversion
- relevant Methods:
  - initWithDouble:
  - doubleValue:
  - unsignedIntValue:
- give NSNumber a double, look at it as an unsigned int

- our pixel_t could be:

```
@interface pixel_t : NSObject
                    <NSCopying>
{
    NSNumber *rgb[3];
}
```

- init routine receives double arguments:

```
- (id) init: (double) r:
                (double) g:
                (double) b:
{ if (!(self = [super init]))
        return nil;
  drgb[0] = [[NSNumber alloc]
      initWithDouble: r];
      :
  return self;
}
```

- every method that uses

NSNumber accessor,

including pixel_t's accessor:

```
- (double) get: (int) i
  {
    return [rgb[i] doubleValue];
  }
```

- and the other "operators" too:

```
-(pixel_t *) dif:(pixel_t *)p
{
  pixel_t *result = [[(pixel_t alloc]
                      init];
  for(i=0; i<3; i++)
  [result set:i:
     [self get:i] - [p get:i]];
  return [result autorelease];
}
```

- other interesting NS classes:

- NS Dictionary:

  - a list of key, value pairs

  - keys are unique objects but usually NSStrings

  - so we have an array with string indeces

  => associative map (map in C++)

- NSMutableData

  - basically a buffer for storing data to facilitate I/O

  - we could represent our ppm image as a NSMutableData block of memory →

```
NSMutableData *data =
    [[NSMutableData alloc] init];
char *s;
asprintf(&s, "P6 %d %d 255",
                width, height);
[data appendBytes: s :
    length: strlen(s)];
free(s);
```

```
for(y=h-1; y>=0; y--) {
    for(x=0; x<w; x++) {
        for(i=0; i<3; i++)
            icolor[i] = 255.0 * color[i];
        [data appendBytes:
            icolor length:3];
    }
}
```

- then can use NSFileManager to write entire data block.

```
NSFileManager *fm =
[NSFileManager defaultManager];
if([fm createFileAtPath:
    @"test.ppm"
Contents: data
attributes: nil] == NO)
  //error
```

- You've seen:

NSString

NSNumber

NSArray

NSEnumerator

NSImage

NSData

NSFileManager

NSDictionary

- Next time:
  Intro to C++