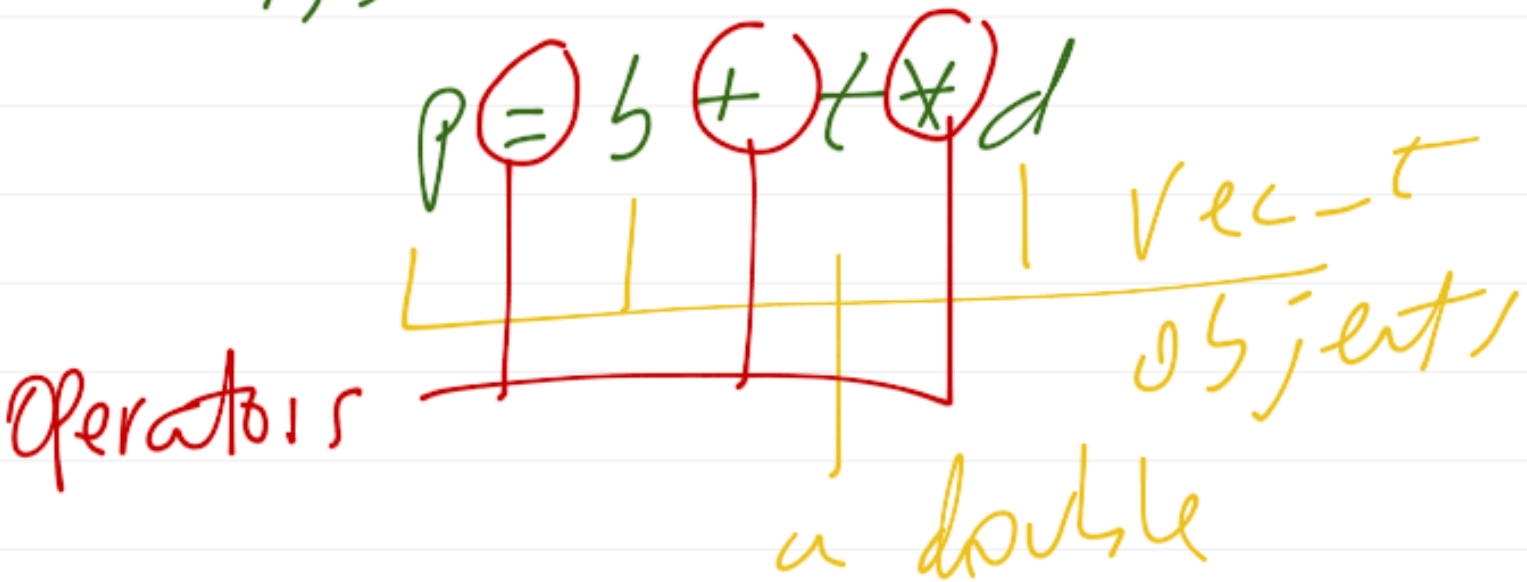
Objective-C

- C with objects
- based on Smalltalk: message-passing between objects
- object-oriented idea (one of them, pertaining to usage)

$$\vec{p} = \vec{b} + t\vec{d}$$

I'd like to just write this in my program

- A digression into C++:
this is ok in C++:



- in C, compiler does not know how to assign (=) a Vec_t, or add (+) a Vec_t, or scale (*) by a float

- C++: you write these operators for the class vec_t

- you would write

vec_t & operator= (vec_t & rhs)

pass by reference

vec_t & operator+ (vec_t &rhs)

vec_t &operator* (double, vec_t &rhs)

- C++ syntax introduces operators, pass by reference, templates, various other ideas
- Objective-C is a gentler intro to objects I think
- C with objects basically
- Quirk:
  - EVERY OBJECT IS A POINTER, DYNAMICALLY ALLOCATED
  - MEMORY MGMT CRITICAL

- idea still is:

$$p = b + t * d$$

in C it's:

```
vec_t  p, b, d;
double t;

vec_scale(t, d, d)
vec_sum(b, d, p)
```

works but looks clumsy,
not very readable

$$p = b + t * d$$

- in Obj-C:

```
p = [[d scale: t] plus: b];
```

where declarations are:

```
vec_t *p = [(vec_t *)
    [vec_t alloc] init];
```

save for b, d

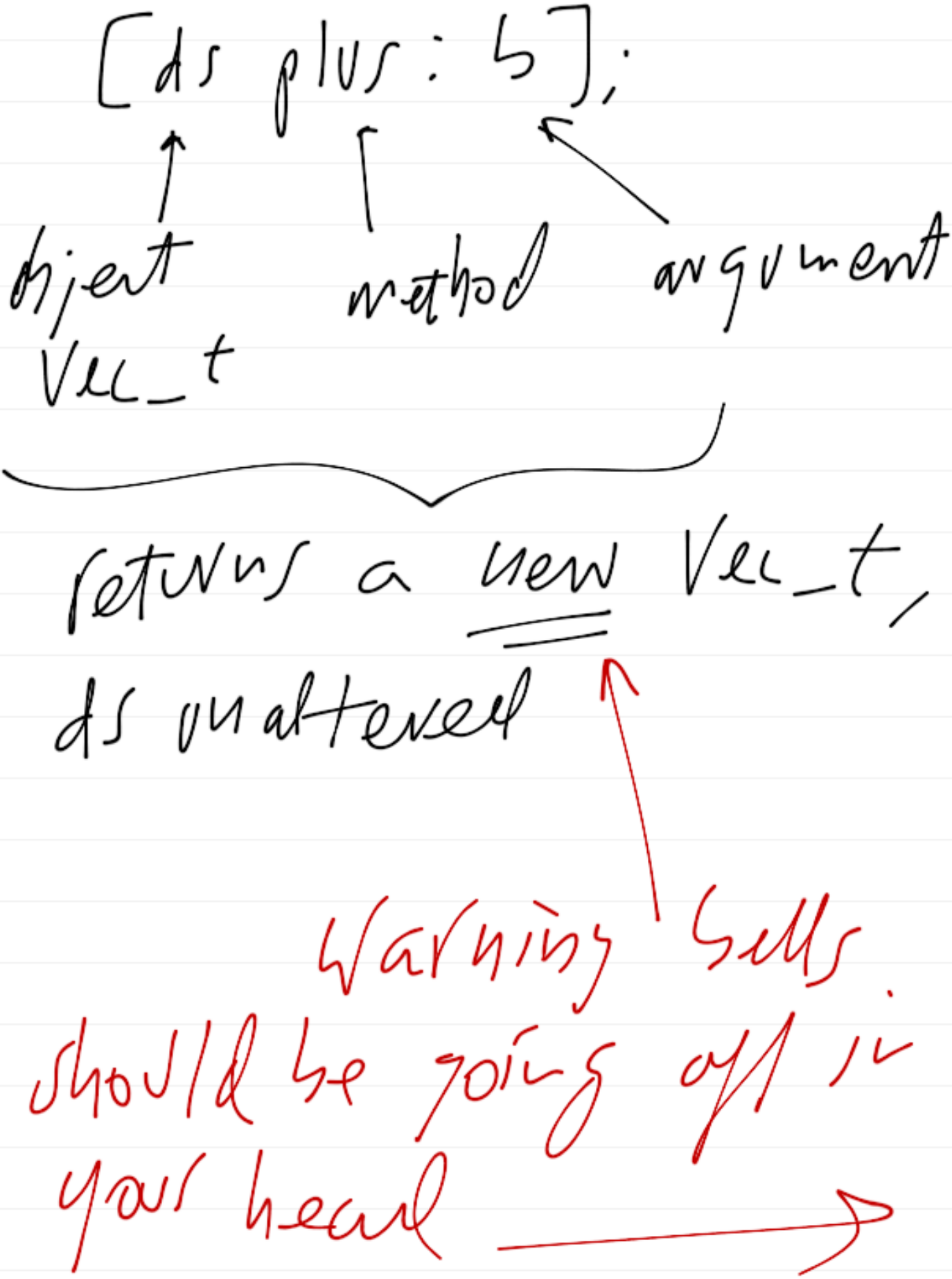
```
p = [[d scale: t] plus: b];
```


```
[d scale: t]
```

returns a new Vec_t,
does not alter d

you can rewrite as

```
ds = [d scale: t];
p = [ds plus: b];
```


[ds plus: b];
object
Vec_t
method
argument
returns a new Vec_t,
ds unaltered
warning bells
should be going off in
your head

```
p = [[ [dscale: t] plus: 5 ];
```

if this is a new Vec_t
(pointer to since all
objects allocated dynamically,
what happens to memory
pointed to by p?)

MEMORY LEAK.

unless you free p just
before assignment

[p autorelease];
p = [[[d scale:t] plus:5];

- obj-C "free" replaced by [autorelease] or [release] for objects allocated with [<obj> alloc]

- other items you malloc can & should be free'd, but you shouldn't have to malloc anything

- the onus is on you, the programmer, to consciously think about memory usage

- What about "garbage collection"?
  1. don't rely on it
  2. it's going away (deprecated)
     iOS 6.x I believe

- what happens when you [alloc], [release], [autorelease]
- User Reference Counting (RC)
- every time you [alloc] mem. usage for that object increments by 1
- every time you [release] mem. count decrements by 1
- [autorelease]: decrement "later"

- What is meant by "later"?
- after next usage
- for example, here is the intent:
- fprintf(stderr, "v2-v1 = ", [v2 minus: v1]);

(This won't work because fprintf doesn't know how to print a vec_t; vec_t needs to print itself)

```
[v3 write: stderr: "v3 = "];
```

write method

argument (FILE *)

argument (char *)

so, we'd like to write

```
[[v2 minus: v1] write: ...]
```

a new rect_t

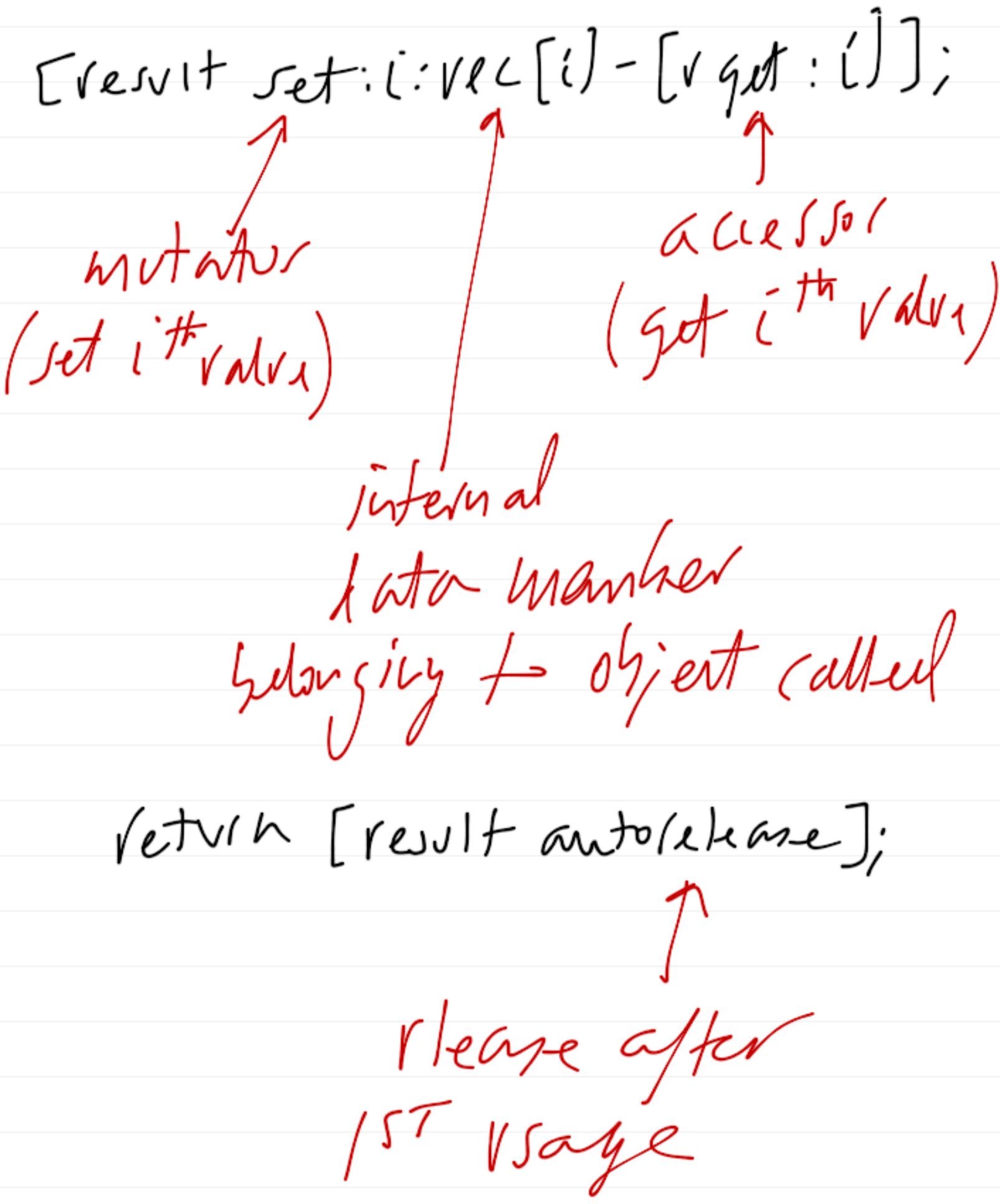
writes itself

AND THEN RELEASES ITSELF

"LATER" (after this call)

- the vec_t minus method returns a vec_t object that will be autoreleased
- here's the minus function:


```
-(vec_t *)minus:(vec_t *)v
{
 vec_t *result=[[vec_t alloc] init];
 int i;
 for(i=0; i<3; i++)
  [result set:i:vec[i]-[v get:i]];
  return [result autorelease];
}
```


[result set:i:vec[i]-[v get:i]];
mutator
(set ith value)
accessor
(get ith value)
internal
data manager
belonging to object called
return [result autorelease];
release after
1st usage

- ok, but what if we want:

```
vec_t *v3 = [[vec_t] alloc] init];
[v3 autorelease];
v3 = [v2 minus: v1];
```

RC = 0 (autorelease)

RC = 1 (alloc)

implicit autorelease, RC = 0

seg fault!!

- in this case, need to retain

```
v3 = [[v2 minus: v1] retain];
```

autorelease not called, so RC = 1

- what keeps track of these reference counts?
- the NSAutoreleasePool
- main() starts with:

```
NSAutoreleasePool *pool =
[[NSAutoreleasePool alloc] init];
```

and ends with:

```
[pool drain];
return 0;
```

— lab 7:

```
vec_t *v1 =
  [(vec_t*)[vec_t alloc]
      init: 3.0: 4.0: 5.0];
vec_t *v2 =
  [(vec_t*)[vec_t alloc] init];
vec_t *v3 = nil;
vec_t *v4 = nil;
vec_t *v5 = [v1 copy];
```

overloaded

Obj-C

NULL

- at end of main, before
[pool drain];

```
[v1 release];
[v2 release];
[v3 release];
[v4 release]; // if alloc'ed
[v5 release];
```

- on top of main.m :

(not .c anymore)

#import <Foundation/Foundation.h>

#import "Vector.h"

interface

Vector.m : implementation

Vector.h:

```
@interface vec_t : NSObject<NSCopying>
```

class (object)

parent (vec_t inherits NSObject's data, methods)

protocol (virtual functions)

"NSCopying says object vec_t promises to implement copyWithZone method

vector.h contd:

```
@interface vect : NSObject<NSCopying>
{
    // instance variable
       (data members)
    double vec[3];
}

// class methods

// constructors (overloaded)
-(id) init: (double) x;
-(id) init: (double) x: (double) y;
-(id) init: (double) x: (double) y: (double) z;
```

- (id) init; inherited from NSObject
- above 3 init variants can be called with 1, 2, or 3 args

```
- (id) init: (double) x
    { return [self init: x: 0: 0]; }
- (id) init: (double) x: (double) y
    { return [self init: x: y: 0]; }
```

= ptr to self

(like a (void *))

```
- (id) initWithX:(double)x Y:(double)y Z:(double)z
{
    if(!(self = [super init]))
        return nil;
    vec[0]=x; vec[1]=y; vec[2]=z;
    return self;
}
```

parent

```
//protocols
-(id) copyWithZone:(NSZone *)zone
{
    Vec_t *clone =
        [[Vec_t allocWithZone:zone]
                    init];
    [clone set:v[0]:v[1]:v[2]];
    return clone;
}
```

set clone's data with arg

// accessor / mutator

```
- (double) get: (int) i
    { return vec[i]; }
- (void) set: (int) i : (double) d
    { vec[i] = d; }
```

= should really check i ok.

```
- (void) set: (double) x : (double) y : (double) z
{
  vec[0] = x; vec[1] = y; vec[2] = z;
}
```

- I/O:

```
-(void) write: (FILE*)f: (char*)msg
{ int i;
  fprintf(f, "%s", msg);
  for(i=0; i<3; i++)
    fprintf(f, "%8.3lf", vec[i]);
fprintf(f, "\n");
}
```

- you need to write:

  plus:

  minus:

  scale:

  unit:

  dot:

  len:

- read: would be nice too

Makefile gets a makeover:

```
.SUFFIXES: .m
```

} so make knows what .m files are

```
UNAME = $(shell uname)
```

(on Linux 'uname' gives Linux on Mac Darwin)

```
ifeq ($(UNAME), Linux)
CFLAGS = -g `gnustep-config \
        --objc-flags` \
INCLUDE = -I. \
        -I/usr/include/GNUstep
LDFLAGS = -L. \
        -L/usr/lib -L/usr/lib/GNUstep
LDLIBS = -lgnuster -lbase
endif
```

backward quotes

```
ifeq ($(UNAME), Darwin)
CFLAGS = -g \
  -mmacosx-version-min=10.6
INCLUDE = -I.
LDFLAGS = -L. -L/usr/lib
LDLIBS = -framework Foundation
endif
```

These two sections make code portable across Linux/Mac

.M.O:

```
$(CC) -c $(INCLUDE) $(CFLAGS) $<
```

how compile (but not yet link) .m files

```
OBJS = \
  vector.o \
  main.o

all: main
```

I dispensed with libvec here, but you can put it in

```
main: main.o $(OBJS)
    $(CC) -o $@ $(OBJS)
        $(LDFLAGS) $(LDLIBS)

clean:
    rm -f ¥.0 core
    rm -f main
```