Emily Holmes

Precedent and Associativity

1. The precedent rules of a language specify which operator is evaluated first when two operators w/ different precedent are adjacent in an expression
   a. Adjacent operators are separated by a single operant
2. The associativity rules of a language specify which operator is evaluated first when two operators with the same precedent are adjacent in an expression.
3. Table of Precedent and Associativity
   (arranged from highest to lowest precedent)

operator	Associativity	Description
(expression)		Parentesis used for grouping
::	Right→Left	(unary) global scope resolution operator
::	Left→Right	class scope resolution operator
()	Right→Left	Parentheses used for a function call
()	Right→Left	value construction, ex: int type(expression)
.	Right→Left	member selection via struct or class object
->	Right→Left	member selection via pointer
[]	Right→Left	array element access
const_cast	Right→Left	specialized type cast
dynamic_cast	Right→Left	
reinterpret_cast	Right→Left	
static_cast	Right→Left	
typeid	Right→Left	type identification
++ --	Right→Left	Post fix versions of increment/decrement

(All operators in this section are unary (one argument) operators

Operator	Associativity	Description
++ --	Right→Left	Prefix version of increment/decrement
+ -	Right→Left	unary versions
!	Right→Left	logical NOT
~ bitwise	Right→Left	~~bitwise complement (~ones element)~~ dereference
&	Right→Left	address of
*	Right→Left	~~dereference~~ allocates memory to dynamic object
new	Right→Left	de-allocates memory allocated to dynamic object
delete	Right→Left	

⇒ Continued

(unary operators continued)

Operator	Associativity	Description
new	Right → Left	
delete	Right → Left	
new []	Right → Left	allocate memory to dynamic array
delete []	Right → Left	deallocate memory allocated to dynamic array
size of	Right → Left	for computing storage size of data
(type)	Right → Left	cast (C-style type conversion)
.*	Left → Right	struct/union/object pointer (member dereference)
->*	Left → Right	Pointer to struct/object/union pointer (indirect member dereference)
* / %	Left → Right	multiplication and division
+ -	Left → Right	addition and subtraction
>> <<	Left → Right	input and output operators
< <= > >=	Left → Right	inequalities relational
== !=	Left → Right	equality relational
&	Left → Right	bitwise AND
^	Left → Right	bitwise XOR
|	Left → Right	bitwise OR
&&	Left → Right	Logical AND
||	Left → Right	Logical OR
?:	Right → Left	conditional ()?:)
=	Right → Left	assignment
*=	Right → Left	multiplication and assignment
/=	Right → Left	division and assignment
%=	Right → Left	modulus (remainder) and assignment
+=	Right → Left	addition and assignment
-=	Right → Left	subtraction and assignment

Precedent and Associativity (continued)

3. continued
(unary operators table continued)

Operators	Associativity	Description
throw	Left → right	throw exception
,	Left → Right	the operator, not the separator (combine two expression into one)

4. what is this code trying to do?
`if ( a << 1 + 1 == c & 0xf) {...}`

a. list base on precedence:
(highest to lowest) `+` `<<` `==` `&`

b. `(a<<1+1 == c & 0xf)`
⇓
`(a << (1+1) == c & 0xf)`
⇓
`((a << (1+1)) == c & 0xf)`
⇓
`(((a<< (1+1)) == c) & 0xf)`

5. Precedent is which operator get applied first.

a. `x*y + a`

b. `* +`
i) y operant is bound to the * first: `(x*y) + a`

6. Associativity is which operator gets executed first when precedent is the same:

a. `a + b + c` + is Left to Right
⇓
`(a + b) + c`

b. `a - b + c`
⇓
`(a - b) + c`

62 c. a+b-c ⇓ (a+b)-c

d. a/b/c ⇓ (a/b)/c

e. i = j = k = 1234; ('right to left)

⇓

i = j = (k = 1234)

⇓

i = (j = (k = 1234))

7. Tip: look at the operand, not the operator, Example:
   a. A left associative operator means that operand goes w/ the operator on the left
      ~~a + b + c~~ a + b + c

8. How would these line print?

```
int & f(int i){
    cout << i << endl;
    static int x;
    return x;
}
int main(int argc, char **argv){
    ...
    f(1) = f(2) = f(3);
    ...
    func(f(1), f(2), f(3));
    ...
}
```

Linux	On Sun
1, 2, 3	1, 2, 3
3 2 1	1, 2, 3

9. Associativity doesn't mean evaluation order.
   you could have left associative and right-to-left evaluation order

10. C++ requires that arguments to a function be completely evaluated (and all side-effects posted) prior to entering the function, but the implementation is free to evaluated the arguments in any order, and presumably side-effect can be used to determine this order for a particular implementation

11. left-associative: operations are grouped from the left a∘b∘c ⇒ (a∘b)∘c
    right-associative: operations are grouped for the right a∘b∘c ⇒ a∘(b∘c)
    non-associative: no defined grouping

Templates: Function templates

1. Function templates can operate w/ generic types.

2. Allow to create a function template that can be adapted to more than one type or class without repeating the whole code.

3. Format for declaring function templates w/ type parameters:
   template <class identifier> function_declaration;
   template <typename identifier> function_declaration;
   a. template <class mytype>
      mytype GetMax (mytype a, mytype b){
        return (a>b ? a : b);
      }
      i) to use this function template we use this call type:
        function_name <type> (parameters);
        example:
          int x,y;
          GetMax <int> (x,y);

   b. Full code example:

```
#include <iostream>
using namespace std;
template <class T>
T GetMax (T a, T b){
  T result;
  result = (a>b)? a : b;
  return (result);
}
int main(){
  int i=5, j=6, k;
  long L=10, m=5, n;
  k=GetMax <int>(i,j);
  n=GetMax <long>(l,m);
  cout << k <<endl;
  cout << n << endl;
  return 0;
}
```

4. Incase we call a template function w/out specifing the type the compiler automatically defines what type is needed on each call.

5. Since our function only used one template parameter, and our function accept two parameters of same type T, we cannot use objects of different types as arguments.

```
int i;
long L;
h = GetMax(i,L);
```
} wrong

6. In order to accept more than one type parameter; ~~simply~~

```
template <class T, class U>
T GetMin (T a, U b){
  return (a<b? a:b);
}
int main(){
 int i, j;
 long L;
 i = GetMin <int, long> GetMin (j, L);
 return 0;
}
```

8.7.

Templates: Function templates (continued)

8. Function template example: Quicksort

```
#include <iostream>
#include <cstdlib>
using namespace std;
template <class T> vo
void quicksort (T a[], const int & leftarg, const int & rightarg){
  if (leftarg < rightarg){
    T pivotvalue = a[leftarg];
    int left = leftarg -1;
    int right = rightarg +1;
    for(;;){
      while (a[--right] > pivotvalue);
      while (a[++left] < pivotvalue);
      if (left >= right) break;
      T temp = a[right];
      a[right] = a[left];
      a[left] = temp;
    }
    int pivot = right;
    quicksort (a, leftarg, pivot);
    quicksort (a, pivot+1, rightarg);
  }
}
int main (){
  int sortme[10];
  for (int =0; i<10; ++i){
    sortme[i] = rand();
    cout << sortme[i] << " ";
  };
  cout << endl;
  quicksort<int> (sortme, 0, 10-1);
  for (int i =0; i<10; ++i){
    cout << sortme[i] << " ";
  }
  cout << endl;
  return 0
}
```

Sample-midterm 2:

☐ In point 2 & 3 what do you mean with ind. cons the principle/rules used?

① Example?

② ☐

CS446/CS646 Fall 2010 midterm:

① ☐ Point 1: what do you mean with all data must be aligned to a multiple of it size?

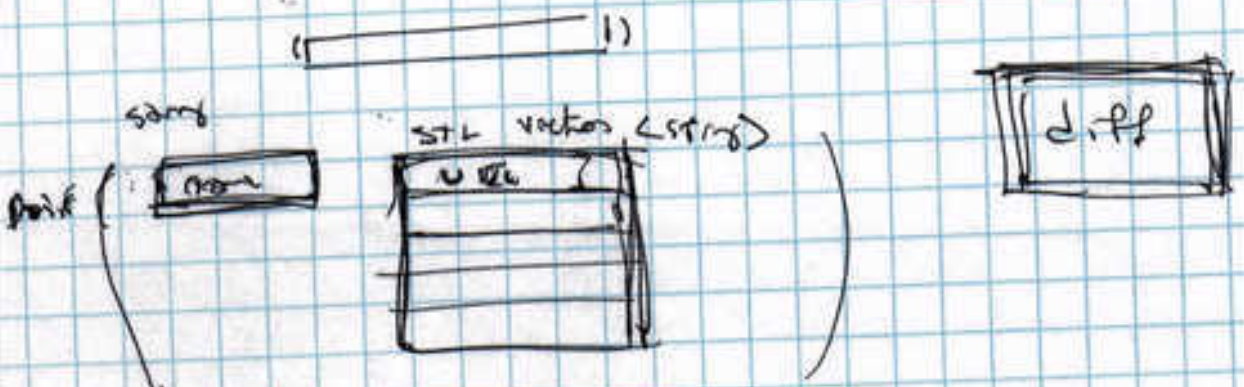
☐ Point 1: Do we have to indicate sizeof in C and then C++?
eg: empty struct = 0 size in C
" " = 1 size in C++

☐ Point 2: what do you mean to write the parts of the ambiguous call? foo(123.00) foo(char(123))

☐ Point 3: By header files should be indented & I don't understand are you refering to headers guards? → include header of full declaration

☐ Point 3:

☐ An example of a functor



dim

swap #(cat)

&((1- dog x) 0) → Hello me

□

---

anything with Recursion

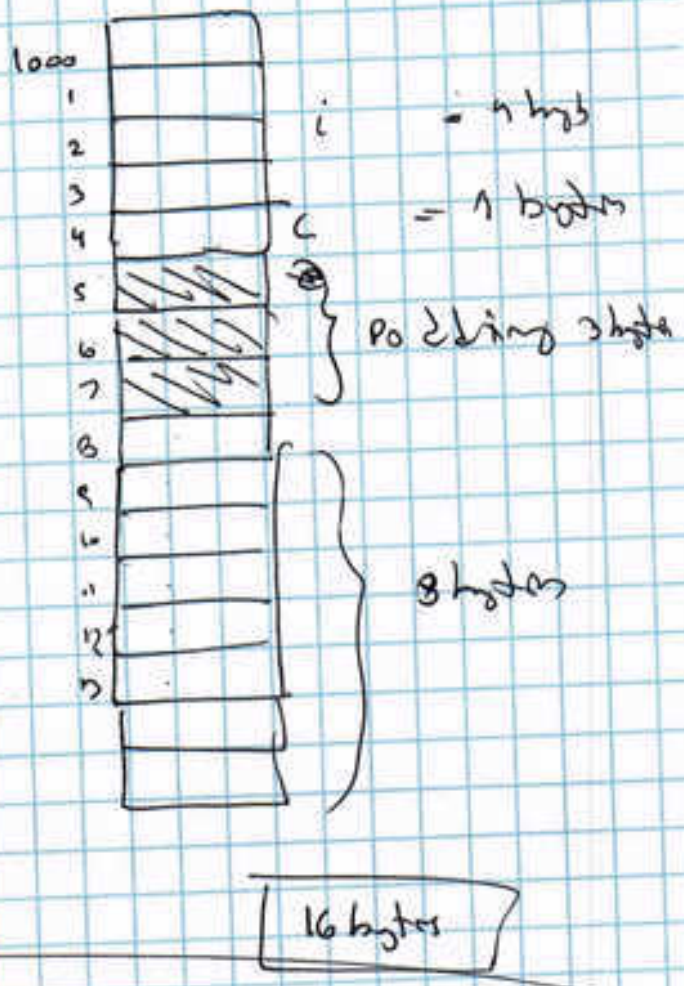
you cannot do Recursion without stack

atom int i;

r();

A(11) ?

```
struct A {
    int i;
    char c;
    double x;
};
```



int 4 }
char 1 } 5

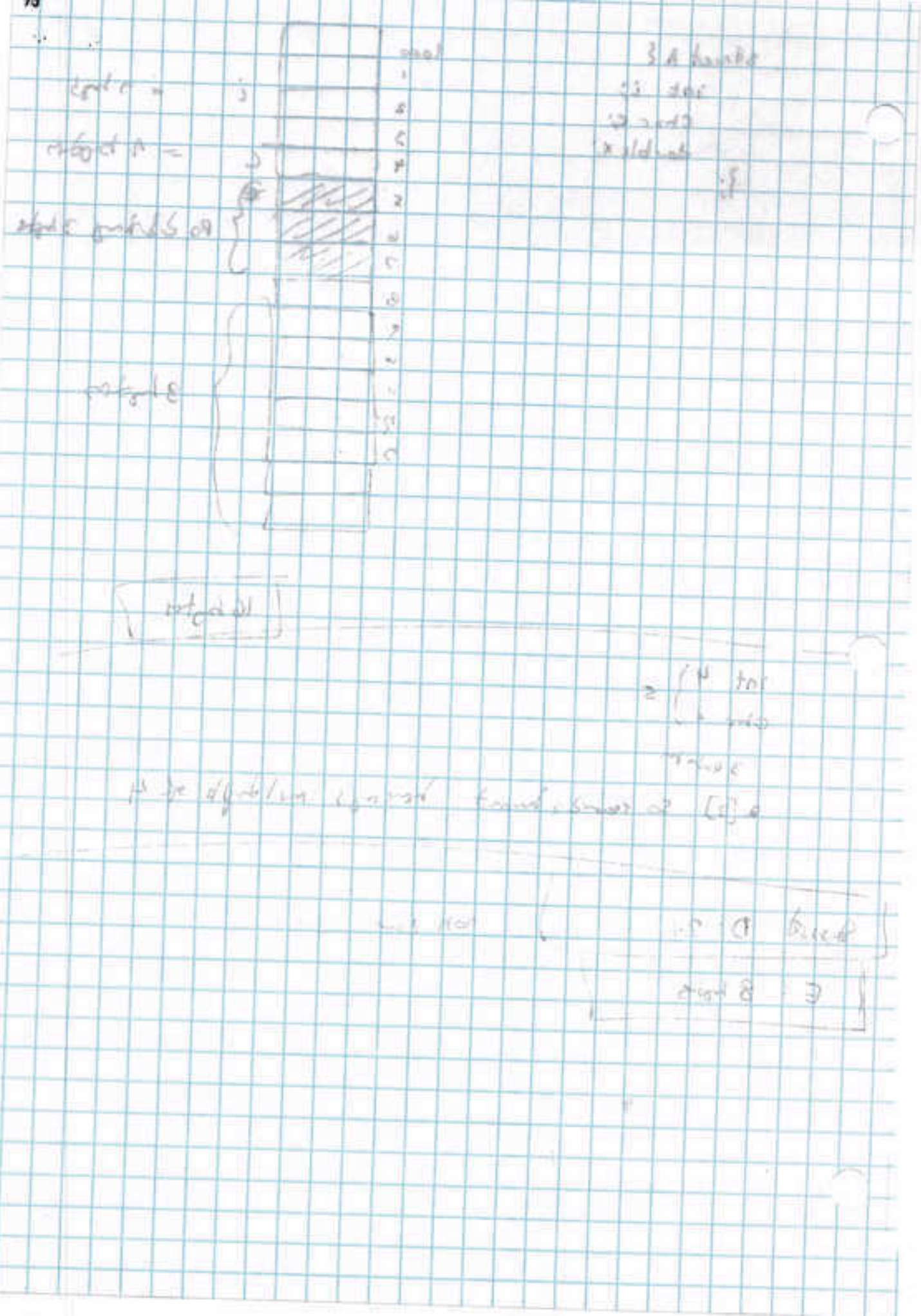
3 extra

a[2] so sizeof must be multiple of 4

struct D: 24

E: 8 bytes

non zero



struct B: public A {
B(int i) : A(i),

second to exec

first base constr

check if instantiated

A
{ m = m2, n1
m3 = m3
A 2;
} first execute

}

143 error because has no constant?

148 Any solution : to many temporaries

V. X = V1. X + V 2 * V3. X;

no temporaries (but gpr registers)

two is not
no solution

219 298 The value of f 2 gave a problem

int i = f(1), f(2);

int i; i = f(1), f(2);

Why didn't print f ?

How is zero (0)?

int f(int a)

int f(2); same as f = 2;

int

```
int f (int a){
    cout << "f\n";
    return a;
}
int f1( int a)
    cout "f1
    ran a
}
int main()
    int i = f(1), f(2);
    cout << i << a
```

→ ff = 2

E)

E *p;

E a[4];

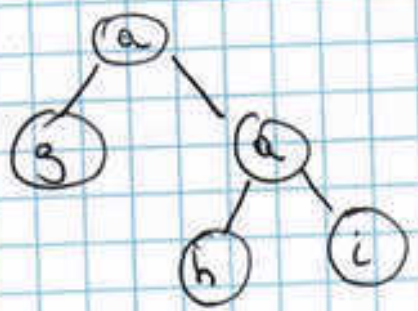
p = a;

~~p + 1 most~~

*(p+1). this would be wrong

---

order of execution right → left



① f # g @ (h @ i) # a %. b %. c

② f # (g @ (h @ i)) # a %. b %. c

(f # (g @ (h @ i)) # a) %. b %. c

((f # (g @ (h @ i))) # a) %. (b %. c)

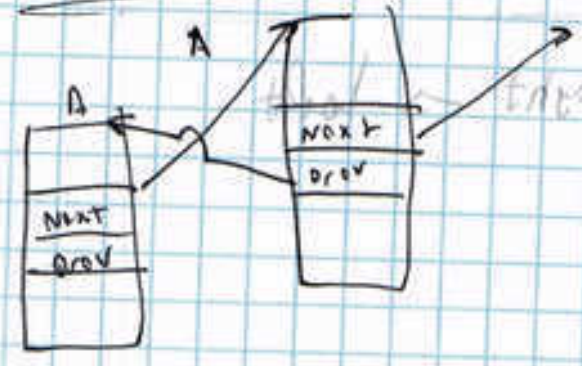
(f # (g @ (h @ i)) # a) %. b %. c)

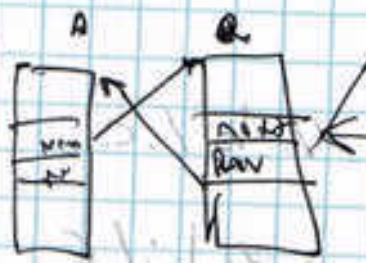
34

7. foo can't have arguments

(*(*foo())[2])()

Pointers & Expr





⊖ a + 4bit gives you 4bits

⊖ a + 4bit + 8 address of previous pointer

⇓

(a + 4bit + 8) = 0; ← dereference

⇓

No space → *(char *) (a + 4bit + 8)

To give can add 1 in addition to 2 char*

⇓

* (char **) (a + 4bit + 8) = 0;

do
(( ) (

while(0)

5.

[redacted]

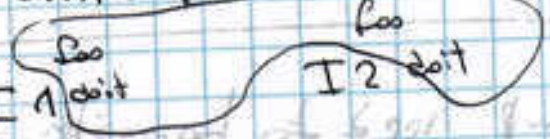
Linker error

① Rd. C

②

③

* ④ not class template are virtual



C1

~~Foo(virtual)~~
virtual Foo

C1

C2

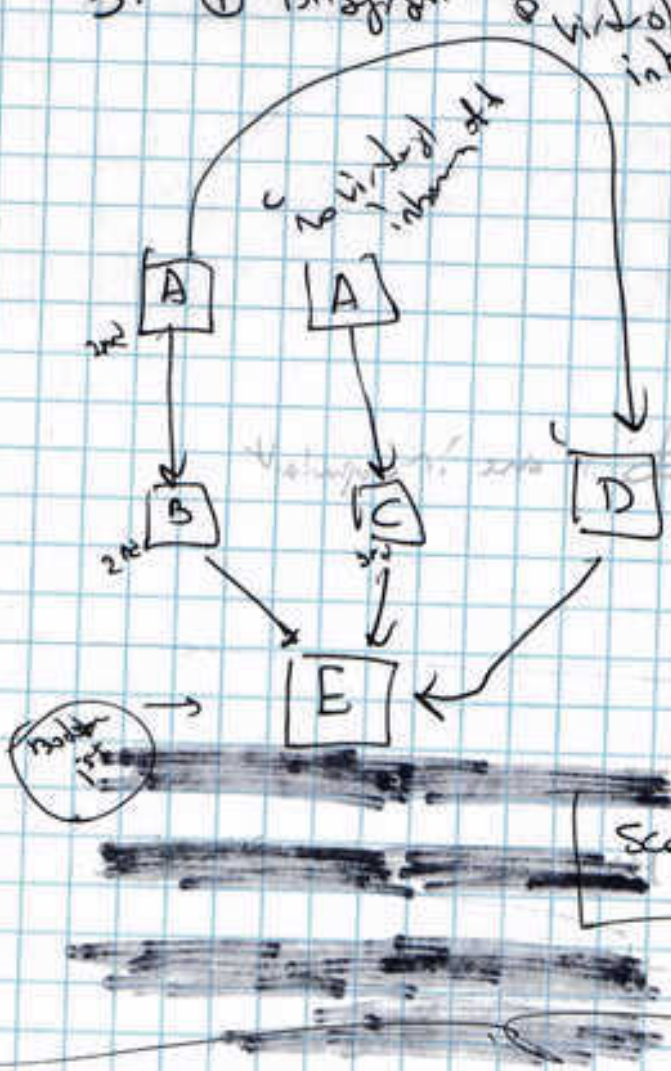
A

C2

virtual Foo

called do:t

5. ① Diagram



two subobject
↳ each # constructor

B writing A data

Scope resolution operator

Limits do not override function symbols and parameters

No temp need to handle it