All Arrays of Given Result Become One Class
Array Assignment Updates Given Array at Given Index

```
class Array {
  int length;
  data : int[]
}
a[i] = x
```

length : Array -> int

data : Array -> (Int -> Int)

or simply: Array x Int -> Int

➔ a.data[i] = x

➔ data= data((a,i):= x)

Assignments to Java arrays: Now including All Assertions (safety ensured, or your models back)

```
class Array {
  int length;
  data : int[]
}
```

length : Array -> int

data : Array -> (Int -> Int)

or simply: Array x Int -> Int

a[i] = x →

assert $(a \neq null)$;
assert $(0 \leq i \land i < length(a))$;
data= data((a,i):= x)

y = a[i] →

assert $(a \neq null)$;
assert $(0 \leq i \land i < length(a))$
y = data((a, i))

Variables in C and Assembly

Can this assertion fail in C++ (or Pascal)?

```
void funny(int& x, int& y) {
  x= 4;
  y= 5;
  assert(x==4);
}
int z;
funny(z, z);
```

Memory Model in C

Just one global array of locations:

mem : int → int // one big array

each variable x has address in memory, xAddr, which is &x

We map operations to operations on this array:

```
int x;
int y;
int* p;
y= x        → mem[yAddr]= mem[xAddr]
x=y+z       → mem[xAddr]= mem[yAddr] + mem[zAddr]
y = *p      → mem[yAddr]= mem[mem[pAddr]]
p = &x      → mem[pAddr] = xAddr
*p = x      → mem[mem[pAddr]]= mem[xAddr]
```

Variables in C and Assembly

Can this assertion fail in C++ (or Pascal)?

```
void funny(int& x, int& y) {
  x= 4;
  y= 5;
  assert(x==4);
}
int z;
funny(&z, &z);
```

```
void funny(xAddr, yAddr) {
  mem[xAddr]= 4;
  mem[yAddr]= 5;
  assert(mem[xAddr]==4);
}
zAddr = someNiceLocation
funny(zAddr, zAddr);
```

Disadvantage of Global Array

In Java:

wp(x=E, y > 0) =

In C:

wp(x=E, y > 0) =

Disadvantage of Global Array

In Java:

wp(x=E, y > 0) = y > 0

In C:

wp(x=E, y > 0) =
wp(mem[xAddr]=E', mem[yAddr]>0) =
wp(mem= mem(xAddr:=E'), mem(yAddr)>0) =
(mem(yAddr)>0)[mem:=mem(xAddr:=E')] =
(mem(xAddr:=E'))(yAddr) > 0

Each assignment can interfere with each value!
This is a problem with the language, not our model

More About Allocation

New Objects Point Nowhere

class C { int f; C next; C prev; }

this should work:

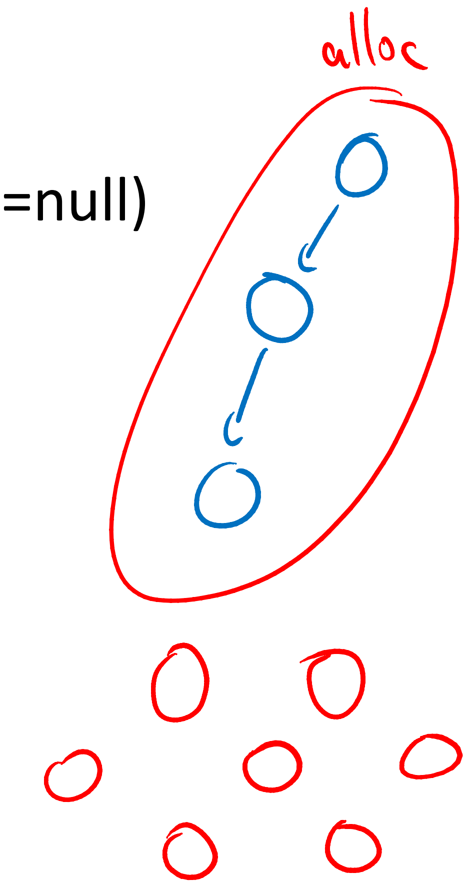
```
x = new C();
assert(x.f==0 && c.next==null  && c.prev==null)
```

x = new C(); →

havoc(x)
assume (x ∉ alloc)
alloc = alloc ∪ {x}
assume (f(x)==0 ∧
next(x)=null ∧
prev(x)=null);

1) use assignment
f= f(x:=0)

2) use assume



If you are new, you are known by few

class C { int f; C next; C prev; }

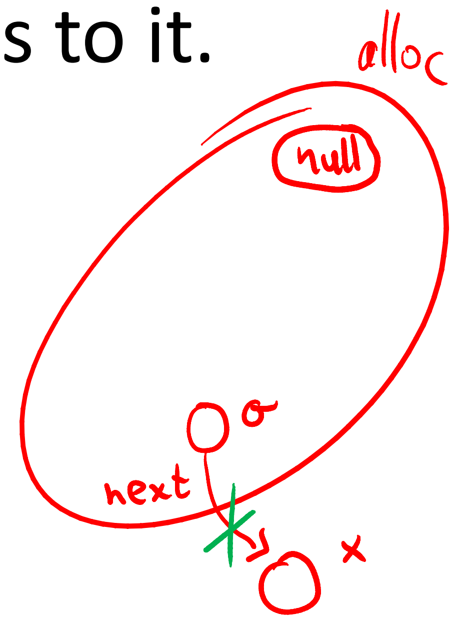
Assume C is the only class in the program

Lonely object: no other object points to it.

Newly allocated objects are lonely!

x = new C(); → [

$\forall \sigma, \sigma \in alloc \rightarrow next(\sigma) \neq x$



$\forall \sigma. \quad \sigma \in alloc \rightarrow next(\sigma) \in alloc \wedge prev(\sigma) \in alloc$

Remember our Model of Java Arrays

```
class Array {
  int length;
  data : int[]
}
```

length : Array -> int

data : Array -> (Int -> Int)

or simply: Array x Int -> Int

a[i] = x → assert $(a \neq null)$;

assert $(0 \leq i \land i < length(a))$;

data= data((a,i):= x)

y = a[i] → assert $(a \neq null)$;

assert $(0 \leq i \land i < length(a))$

y = data((a, i))

Allocating New Array of Objects

```
class oArray {
  int length;
  data : Object[]
}
x = new oArray[100] →
```

E

havoc (x);
assume ($x \notin alloc$);
alloc = alloc ∪ {x};
assume (length (x) = E (100) ∧

$\forall i.\ 0 \leq i < E \rightarrow$
data (x, i) = null ∧

$\forall o \in alloc.\ \wedge\ f(o) \neq x$
f ∈ fields(C(o)))

length = length (x := E)

Procedure Contracts

Suppose there are fields and variables f_1, f_2, f_3 (denoted f)

```
        procedure foo(x):
          requires P(x,f)
          modifies f3
          ensures Q(x,old(f),f)
foo(E) →
  assert(P(E,f));
  old_f = f;
  havoc(f3);
  assume Q(E,old_f, f)
```

Modification of Objects

Suppose there are fields and variables $\underbrace{f_1,f_2,f_3}_{f}$ (denoted f)

procedure foo(x):

requires P(x,f)

modifies $x.f_3$

ensures Q(x,f,f')

$x.f_3 = y \rightsquigarrow f_3 = f_3(x := y)$

foo(E) →

assert(P(E,f));

old_f = f; ← $\begin{cases} old\_f_1 = f_1 \\ old\_f_3 = f_3 \end{cases}$

havoc($x.f_3$); → havoc(f_3); assume $\forall z \neq x.\ f_3(z) = old\_f_3(z)$

assume Q(E,old_f, f)

Example

```
class Pair { Object first; Object second; }
void printPair(p : Pair) { ... }
void printBoth(x : Object, y : Object)
modifies first, second  // ?
{
  Pair p = new Pair();
  p.first = x;
  p.second = y;
  printPair(p);
}
```

printBoth (x1,y1)

Allowing Modification of Fresh Objects

Suppose there are fields and variables f_1,f_2,f_3 (denoted f)

procedure foo(x):
  requires P(x,f)
  modifies x.f_3
  ensures Q(x,f,f')

foo(E) →
  assert(P(E,f));
  old_f = f;   old_alloc = alloc;
  havoc f_3, f_2, f_1, alloc
  assume $\forall z \in$ old_alloc. $f_1(z) = old\_f_1(z) \wedge f_2(z) = old\_f_2(z)$
    $(z \neq x \rightarrow f_3(z) = old\_f_3(z))$
  assume Q(E,old_f, f)
  assume (old_alloc ⊆ alloc)

Data remains same if: 1) existed and 2) not listed in m.clause