

Default and Copy Constructors

Steven J. Zeil

June 17, 2013

Contents

1	The Default Constructor	2
2	Copy Constructors	9
2.1	Where Do We Use a Copy Constructor?	10
2.2	Compiler-Generated Copy Constructors	11
2.3	Example: Bid: Compiler-Generated Copy Constructor	11
2.4	Example: BidCollection: Compiler-Generated Copy Constructor	12
2.5	14
2.6	Writing a BidCollection Copy Constructor	21
2.7	Shallow & Deep Copying	26
3	Assignment	28
3.1	Compiler-Generated Assignment Ops	28

1 The Default Constructor

The Default Constructor

The *default constructor* is a constructor that takes no arguments. This is the constructor you are calling when you declare an object with no parameters. E.g.,

```
std::string s;
```

.....

Declaring a Default Constructor

It might be declared like this

```
class Time {
public:
    Time();
    :

```

or with defaults:

```
namespace std {
class string {
public:
    :
    string(char* s = "");
    :

```

Either way, we can *call* it with no parameters.

.....

Why 'default'?

- It's just an ordinary constructor
- But it is used (implicitly) to initialize elements of an array.

- It is also used (implicitly) in other ADTs' constructors when they do not explicitly initialize a data member.

.....

Implicit Use: Arrays

For example, if we declared:

```
std::string words[5000];
```

then each of the 5000 elements of this array will be initialized using the default constructor for string

.....

Implicit Use: Other Constructors

```

struct Bid {
    std::string bidderName;
    double amount;
    std::string itemName;
    Time bidPlacedAt;

    Bid ();
};

Bid::Bid ()
{
    amount = 0.0;
}

```

- bidPlacedAt is initialized using the Time default constructor.
- bidderName and itemName are initialized using the string default constructor.

.....



Explicit Construction in Other Constructors

If we *don't* want the default value, we need to explicitly perform some other initialization:

```
struct Bid {
    std::string bidderName;
    double amount;
    std::string itemName;
    Time bidPlacedAt;

    Bid ();
};

Bid::Bid ()
{
    amount = 0.0;
    bidPlacedAt = Time(8, 0, 0); // 8 AM
}
```

This actually constructs a Time object and then copies it.

- a bit inefficient
 - C++ provides a special way to directly call constructors
-

Explicit Construction: Initializer Lists

Alternate way to explicitly perform some other initialization:

```
Bid::Bid()
: bidPlacedAt(8, 0, 0) // 8 AM
```

```
{  
    amount = 0.0;  
}
```

- Called an *initializer list*
 - data member name followed by constructor arguments
-

Initializer Lists

- *Can* be used to initialize any data member

```
struct Bid {  
    std::string bidderName;  
    double amount;  
    std::string itemName;  
    Time bidPlacedAt;  
  
    Bid();  
};  
  
Bid::Bid()  
    : bidderName(""), amount(0.0),  
      itemName("knick-knack"), bidPlacedAt(8, 0, 0)  
{  
}
```

- *Must* be used to initialize data members that are

- constants (const)
- references
- members of classes that have no default constructors

.....

The Helpful Compiler

If we create no constructors at all for a class, the compiler generates a default constructor for us.

- Initializes each data member using their data types' default constructors
- For primitives such as int, double, pointers, etc., this does nothing at all

.....

Example: Name

```
class Name {  
public:  
    string getGivenName();  
    void setGivenName (string);  
  
    string getSurName();  
    void setSurName (string);  
private:  
    string givenName;  
    string surName;  
};
```

- Compiler will generate a default constructor Name()
- givenName and surName will be initialized using the default constructor of string
 - Probably just fine

.....

Example: Name 2

```
class Name {  
public:  
    Name (string gName, string sName)  
        : givenName(gName), surName(sName) {}  
  
    string getGivenName();  
    void setGivenName (string);  
  
    string getSurName();  
    void setSurName (string);  
private:  
    string givenName;  
    string surName;  
};
```

- Compiler will *not* generate a default constructor Name () because we provided a different constructor
- If we want one, we have to write our own
 - If we don't, we cannot have arrays of Names

.....

Example: Name 3

```
class Name {  
public:  
    Name () {}  
  
    Name (string gName, string sName)  
        : givenName(gName), surName(sName) {}
```



```

string getGivenName();
void setGivenName (string);

string getSurName();
void setSurName (string);
private:
string givenName;
string surName;
};

```

.....

Example: BidCollectionBook implicit default constructor

```

struct BidCollection {
    int MaxSize;
    int size;
    Bid* elements; // array of bids

    /**
     * Read all bids from the indicated file
     */
    void readBids (std::string fileName);
};

```

- Compiler would generate a default constructor
 - that would leave random bits in all 3 data members

.....

Example: BidCollection explicit default constructor

```

struct BidCollection {
    int MaxSize;
    int size;
    Bid* elements; // array of bids

    BidCollection (int MaxBids = 1000);

    /**
     * Read all bids from the indicated file
     */
    void readBids (std::string fileName);
};
:
BidCollection::BidCollection (int theMaxSize)
{
    MaxSize = theMaxSize;
    size = 0;
    elements = new Bid[MaxSize];
}

```

.....

2 Copy Constructors

Copy Constructors

The copy constructor for a class Foo is the constructor of the form:

```

Foo (const Foo& oldCopy);

```

.....

2.1 Where Do We Use a Copy Constructor?

Where Do We Use a Copy Constructor?

The copy constructor gets used in 5 situations:

1. When you declare a new object as a copy of an old one:

```
Time time2 (time1);
```

or

```
Time time2 = time1;
```

2. When a function call passes a parameter “by copy” (i.e., the formal parameter does not have a &):

```
void foo (Time b, int k);
:
Time noon (12, 0, 0);
foo (noon, 0); // foo actually gets a copy of noon
```

3. When a function returns an object:

```
Time foo (int k);
{
    Time t (k, 0, 0);
    :
    return t;
}
```

4. When explicitly invoked in another constructor’s initialization list:

```
Name (string gName, string sName)
: givenName(gName), surName(sName) {}
```

5. When an object is a data member of another class for which the compiler has generated its own copy constructor.

.....

2.2 Compiler-Generated Copy Constructors

Compiler-Generated Copy Constructors

If we do not create a copy constructor for a class, the compiler generates one for us.

- copies each data member via *their* individual copy constructors.
 - For primitive types (int, double, pointers, etc.), just copies the bits.

.....

2.3 Example: Bid: Compiler-Generated Copy Constructor

Example: Bid: Compiler-Generated Copy Constructor

```
struct Bid {
    std::string bidderName;
    double amount;
    std::string itemName;
    Time bidPlacedAt;
};
```

Bid does not provide a copy constructor, so the compiler generates one for us, just as if we had written:

```
struct Bid {
    std::string bidderName;
    double amount;
    std::string itemName;
    Time bidPlacedAt;
```



```

    Bid (const Bid&);
};
:
Bid::Bid (const Bid& b)
    : bidderName(b.bidderName), amount(b.amount),
      itemName(b.itemName), bidPlacedAt(b.bidPlacedAt)
{}

```

and that's probably just fine.

.....

2.4 Example: BidCollection: Compiler-Generated Copy Constructor

Example: BidCollection: Compiler-Generated Copy Constructor

```

struct BidCollection {
    int MaxSize;
    int size;
    Bid* elements; // array of bids

    /**
     * Create a collection capable of holding the indicated number of bids
     */
    BidCollection (int MaxBids = 1000);

    ~BidCollection ();

    /**
     * Read all bids from the indicated file
     */
    void readBids (std::string fileName);
};

```

BidCollection does not provide a copy constructor, so the compiler generates one for us, just as if we had written:



```

struct BidCollection {
    int MaxSize;
    int size;
    Bid* elements; // array of bids

    /**
     * Create a collection capable of holding the indicated number of bids
     */
    BidCollection (int MaxBids = 1000);
    BidCollection (const BidCollection&);
};
:
BidCollection::BidCollection (const BidCollection& bc)
    : MaxSize(bc.MaxSize), size(bc.size),
      elements(bc.elements)
{}

```

which is not good at all!

.....

Example: BidCollection is hard to copy

To see why, suppose we had some application code:

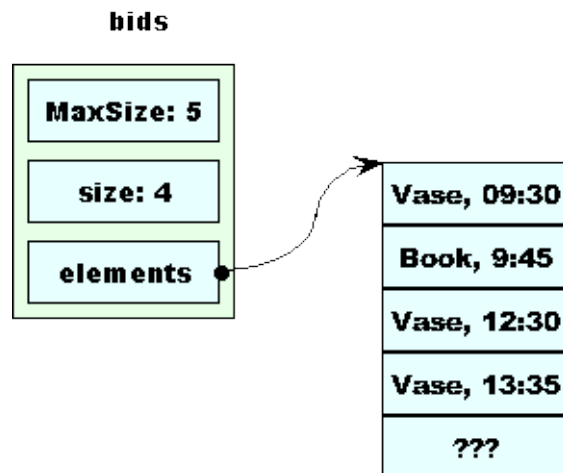
```

BidCollection removeLate (BidCollection bc, Time t)
{
    for (int i = 0; i < bc.size; )
    {
        if (bc.elements[i].bidPlacedAt.noLaterThan(t))
            ++i;
        else
            removeElement (bc.elements, bc.size, i);
    }
}

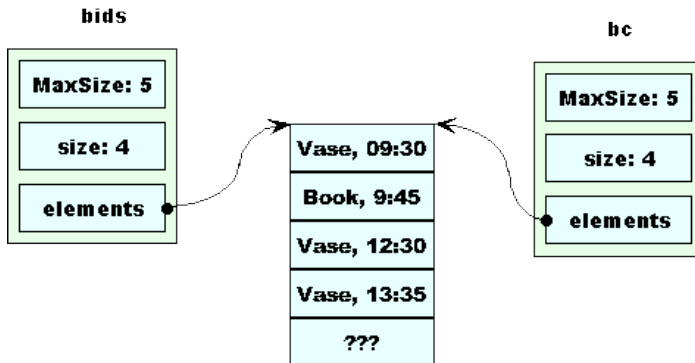
```

```
    }  
    return bc;  
}  
  
<:::>  
BidCollection afterNoonBids =  
    removeLate (bids, Time(12,0,0));
```

Assume we start with this in bids.



2.5



When removeLate is called, we get a copy of bids .

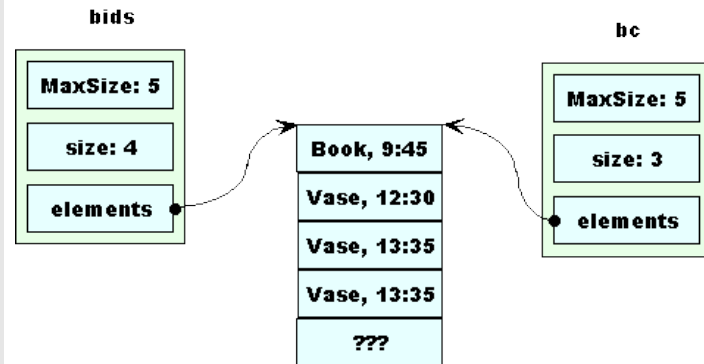
```

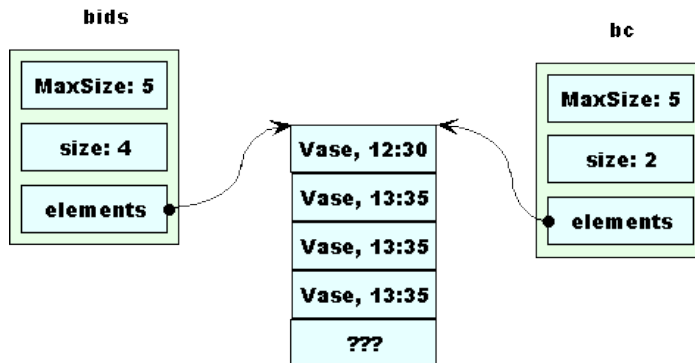
BidCollection removeLate (BidCollection bc, Time t)
{
    for (int i = 0; i < x.size; )
    {
        if (bc.elements[i].bidPlacedAt.noLaterThan(t))
            ++i;
        else
            removeElement (elements, size, i);
    }
    return bc;
}
:
BidCollection afterNoonBids =
    removeLate (bids, Time(12,0,0));
    
```

removeLate removes the first morning bid from bc.

```

BidCollection removeLate (BidCollection bc, Time t)
{
    for (int i = 0; i < x.size; )
    {
        if (bc.elements[i].bidPlacedAt.noLaterThan(t))
            ++i;
        else
            removeElement (elements, size, i);
    }
    return bc;
}
:
BidCollection afterNoonBids =
    removeLate (bids, Time(12,0,0));
    
```





Then `removeLate` removes the remaining morning bid from `bc`.

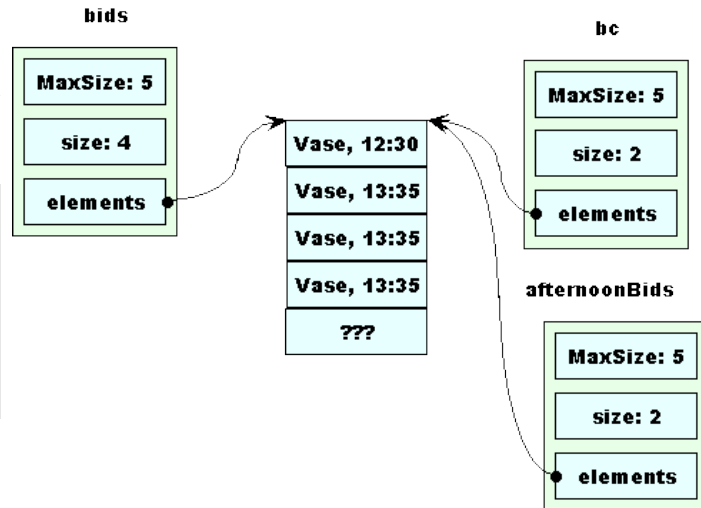
```

BidCollection removeLate (BidCollection bc, Time t)
{
    for (int i = 0; i < x.size;)
    {
        if (bc.elements[i].bidPlacedAt.noLaterThan(t))
            ++i;
        else
            removeElement (elements, size, i);
    }
    return bc;
}
:
BidCollection afterNoonBids =
    removeLate (bids, Time(12,0,0));
    
```


The return statement makes a copy of bc, which is stored in afterNoonBids
 removeLate removes the remaining morning bid from bc.

```

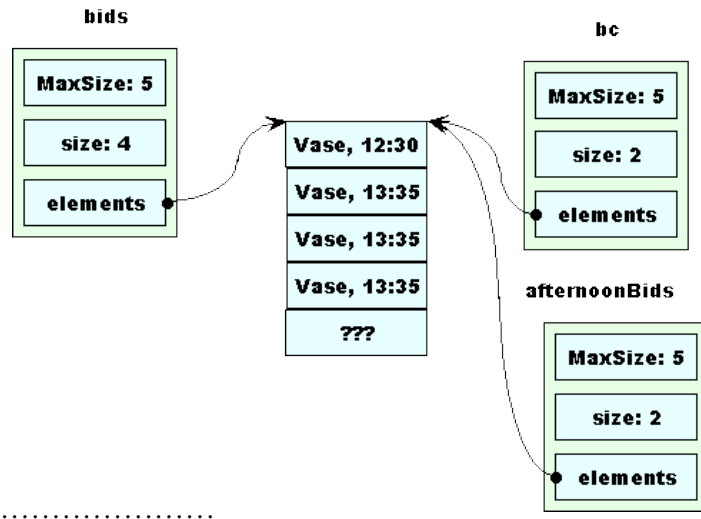
BidCollection removeLate (BidCollection bc, Time t)
{
    :
    return bc;
}
:
BidCollection afterNoonBids =
    removeLate (bids, Time(12,0,0));
    
```



Trouble: bids is corrupted

Note that we have corrupted the original collection, bids

- It thinks it has more (according to size) bids than are left in the array
- The bids that it has are now changed



That's not the worst of it!

When we exit removeLate,

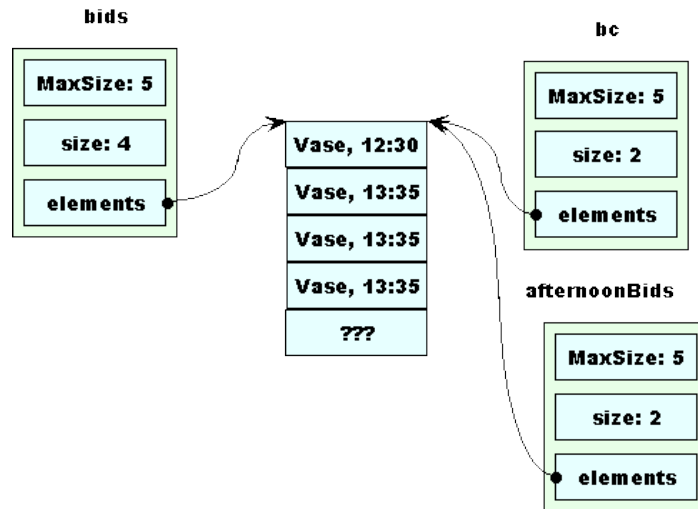
```

BidCollection removeLate (BidCollection bc, Time t)
{
  for (int i = 0; i < x.size;)
  {
    if (bc.elements[i].bidPlacedAt.noLaterThan(t))
      ++i;
    else
      removeElement (elements, size, i);
  }
  return bc;
}
    
```

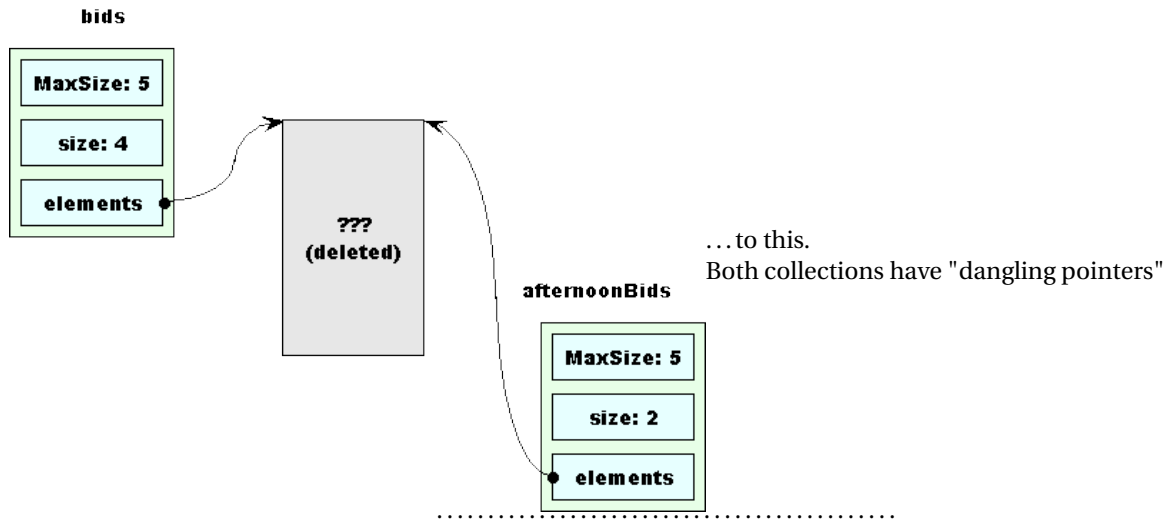
the destructor for BidCollection is called on bc

```
BidCollection::~BidCollection ()  
{  
    delete [] elements;  
}
```

Taking us from this...



What a Mess!



Avoiding this Problem

We could

- Decide never to pass a BidCollection by copy, never return one from a function, never to create a new BidCollection as a copy of an old one
 - We would have to remember this in all future applications
- or, write our own copy constructor that actually works
 - Every BidCollection should have its own unique array

2.6 Writing a BidCollection Copy Constructor

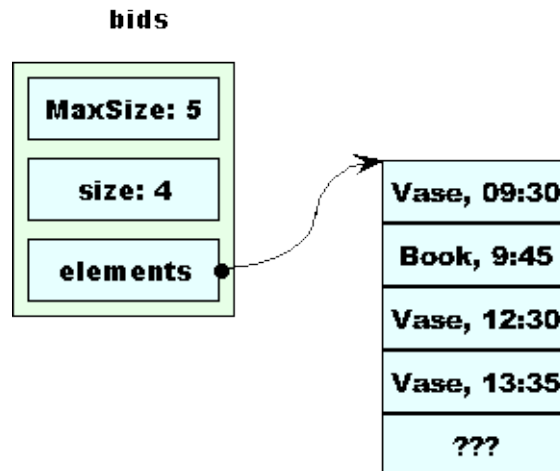
Writing a BidCollection Copy Constructor

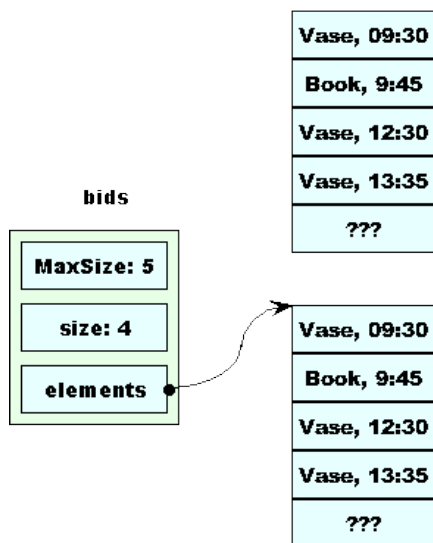
```
BidCollection::BidCollection (const BidCollection& bc)
: MaxSize (bc.MaxSize), size (bc.size)
{
    elements = new Bid[MaxSize];
    for (int i = 0; i < size; ++i)
        elements[i] = bc.elements[i];
}
```

.....

Once More, With Feeling!

With this copy constructor in place, things should go much more smoothly.





When `removeLate` is called, we get a copy of bids.

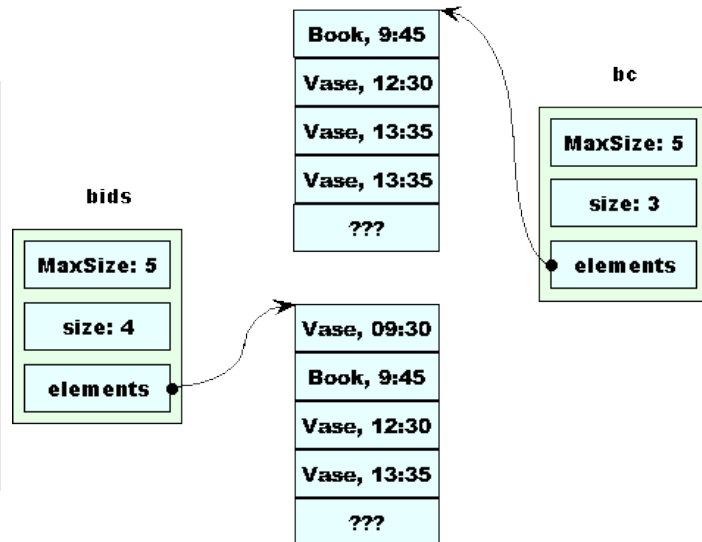
```

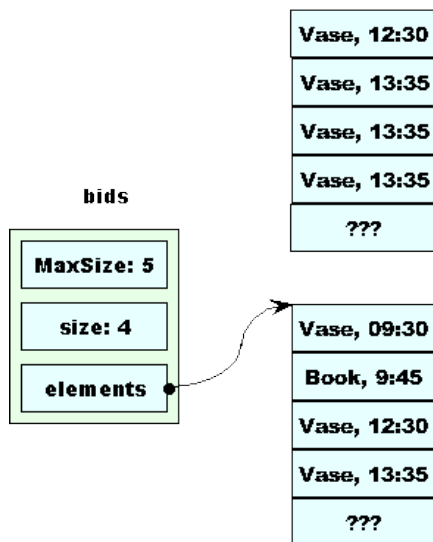
BidCollection removeLate (BidCollection bc, Time t)
{
    for (int i = 0; i < x.size;)
    {
        if (bc.elements[i].bidPlacedAt.noLaterThan(t)
            ++i;
        else
            removeElement (elements, size, i);
    }
    return bc;
}
<:::>
BidCollection afterNoonBids =
    removeLate (bids, Time(12,0,0));
    
```

removeLate removes the first morning bid from bc.

```

BidCollection removeLate (BidCollection bc, Time t)
{
  for (int i = 0; i < x.size;)
  {
    if (bc.elements[i].bidPlacedAt.noLaterThan(t))
      ++i;
    else
      removeElement (elements, size, i);
  }
  return bc;
}
:
BidCollection afterNoonBids =
  removeLate (bids, Time(12,0,0));
    
```





Then `removeLate` removes the remaining morning bid from `bc`.

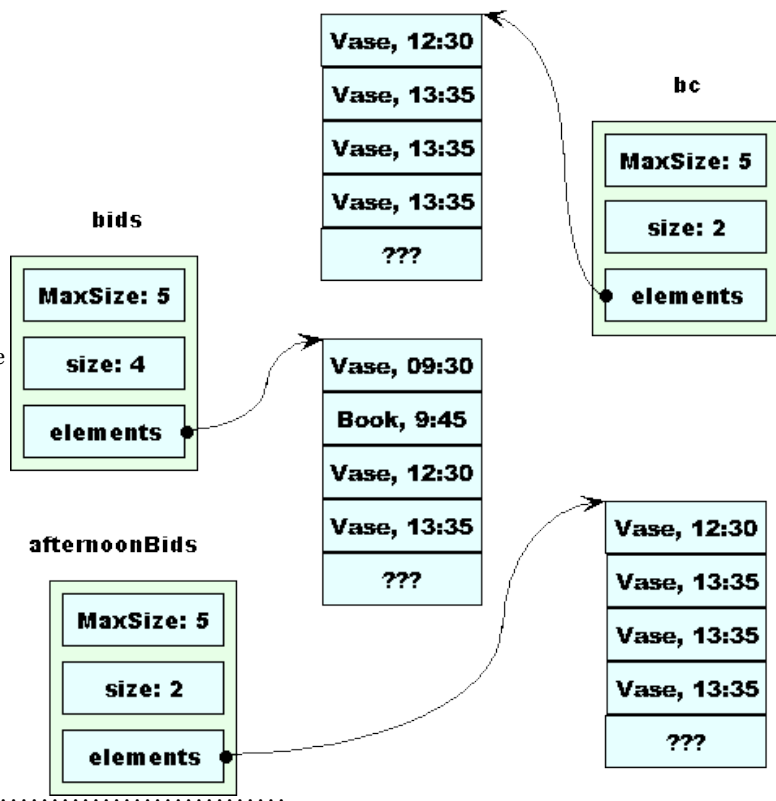
```

BidCollection removeLate (BidCollection bc, Time t)
{
    for (int i = 0; i < x.size;)
    {
        if (bc.elements[i].bidPlacedAt.noLaterThan(t))
            ++i;
        else
            removeElement (elements, size, i);
    }
    return bc;
}
:
BidCollection afterNoonBids =
    removeLate (bids, Time(12,0,0));
    
```

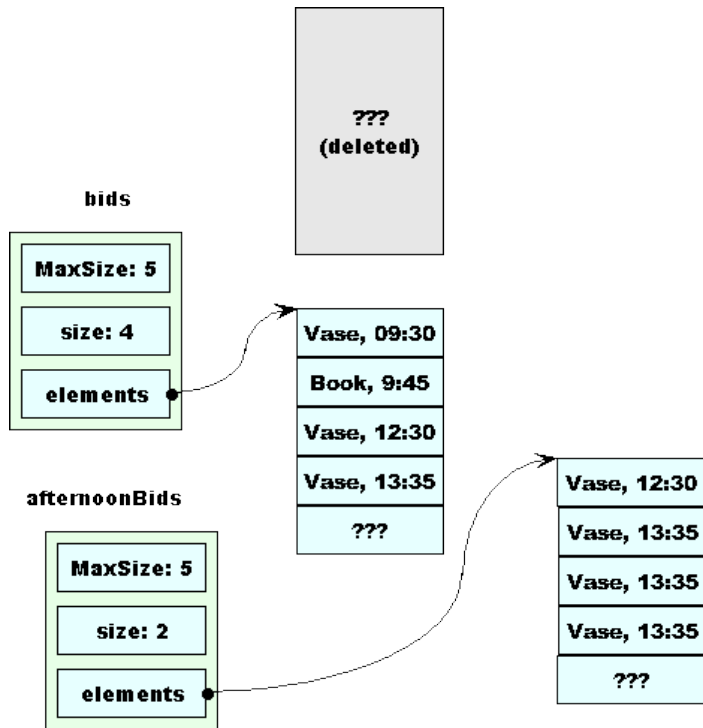

The return statement makes a copy of bc, which is stored in afterNoonBids.
 removeLate removes the remaining morning bid from bc.

```

BidCollection removeLate (BidCollection bc, Time
{
    :
    return bc;
}
:
BidCollection afterNoonBids =
    removeLate (bids, Time(12,0,0));
    
```



Much Nicer



The destructor for `BidCollection` is called on `bc`

- Both remaining collections are OK.

2.7 Shallow & Deep Copying

If We Never Write Our Own

If our data members do not have explicit copy constructors (and their data members do not have explicit copy constructors, and ...) then the compiler-provided copy constructor amounts to a bit-by-bit copy.

.....

Shallow vs Deep Copy

Copy operations are distinguished by how they treat pointers:

- In a *shallow copy*, all pointers are copied.
 - Leads to shared data on the heap.
- In a *deep copy*, objects pointed to are copied, then the new pointer set to the address of the copied object.
 - Copied objects keep exclusive access to the things they point to.

.....

Shallow copy is wrong when...

- Your ADT has pointers among its data members, and
- You don't want to share the objects being pointed to.

.....

Compiler-generated copy constructors are wrong when...

- Your ADT has pointers among its data members, and
- You don't want to share the objects being pointed to.

.....

3 Assignment

Assignment

Copy constructors are not the only way we make copies of data.

- Even more common is the use of assignment:

```
time2 = time1;
```

- Assignment is so common that it can be hard to imagine programming without it
 - though in rare conditions we will do so
 - E.g., you cannot assign `istream`s and `ostream`s
 - * You cannot copy them by constructor either
- So the compiler will try to be helpful again

.....

3.1 Compiler-Generated Assignment Ops

Compiler-Generated Assignment Ops

If you don't provide your own assignment operator for a class, the compiler generates one automatically.

- assigns each data member in turn.
- If none of the members have programmer-supplied assignment ops, then this is a *shallow copy*

.....

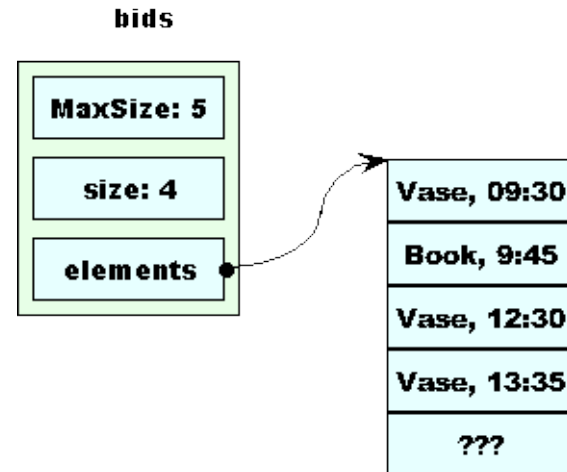
Example: BidCollection: Guess what happens

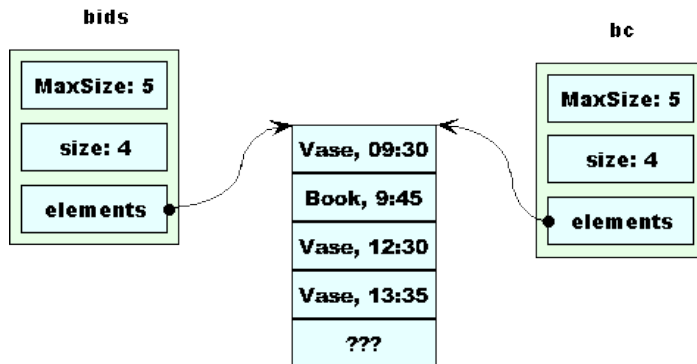
Our BidCollection class has no assignment operator, so the code below uses the compiler-generated version. To see why, suppose we had some application code:

```
BidCollection bids;
:
BidCollection bc;
Time t (12, 0, 0);
bc = bids;
for (int i = 0; i < x.size;)
{
    if (bc.elements[i].bidPlacedAt.noLaterThan(t))
        ++i;
    else
        removeElement (elements, size, i);
}
```

```
BidCollection bc;
Time t (12, 0, 0);
bc = bids;
for (int i = 0; i < x.size;)
{
    if (bc.elements[i].bidPlacedAt.noLaterThan(t))
        ++i;
    else
        removeElement (elements, size, i);
}
```

Assume we start with this in bids.





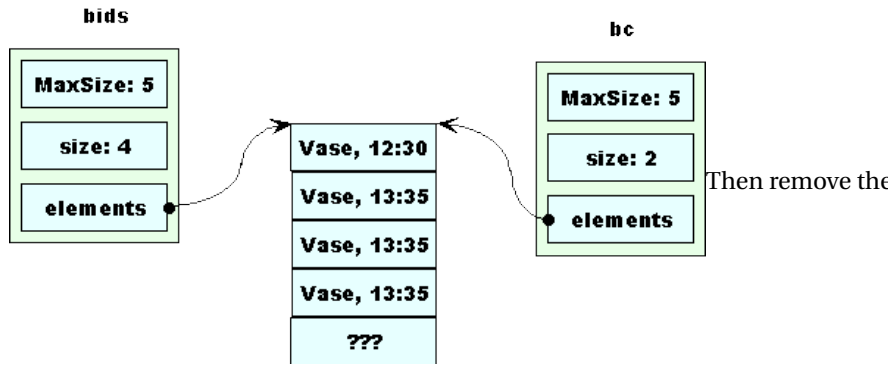
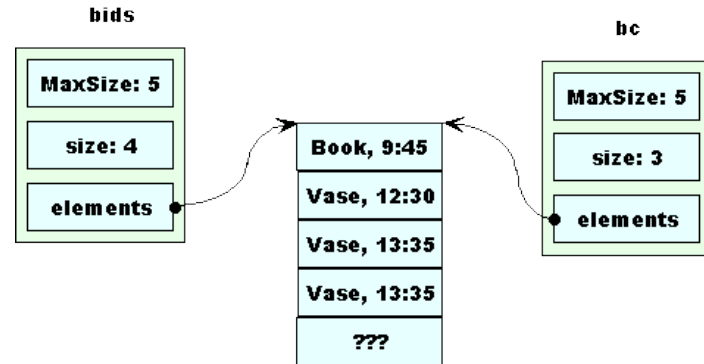
After the assignment, we have copied bids, bit-by-bit, into bc.

```

BidCollection bids;
:
BidCollection bc;
Time t (12, 0, 0);
bc = bids;
for (int i = 0; i < x.size;)
{
    if (bc.elements[i].bidPlacedAt.noLaterThan(t))
        ++i;
    else
        removeElement (elements, size, i);
}

BidCollection bc;
Time t (12, 0, 0);
bc = bids;
for (int i = 0; i < x.size;)
{
    if (bc.elements[i].bidPlacedAt.noLaterThan(t))
        ++i;
    else
        removeElement (elements, size, i);
}
    
```

We remove the first morning bid from bc.

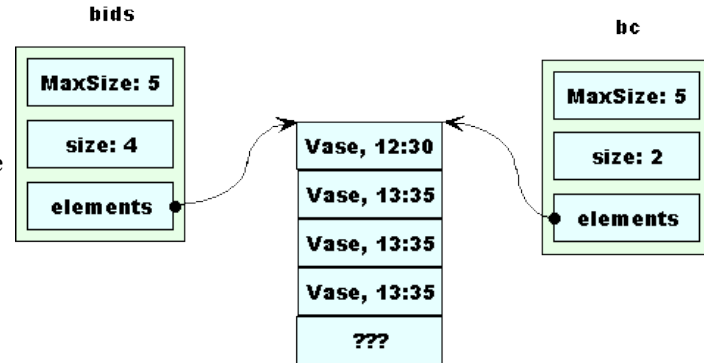


Then remove the remaining morning bid from bc.

Again: bids is corrupted

Note that we have corrupted the original collection, bids

- It thinks it has more (according to size) bids than are left in the array
- The bids that it has are now changed



Avoiding this Problem

We could

- Decide never to assign one BidCollection to another
 - We would have to remember this in all future applications
 - or, write our own assignment that actually works
 - We'll look at how to do this later when we study *operator overloading*
-