

# Copying Data

Steven J. Zeil

November 13, 2013

## Contents

<b>1</b>	<b>Destructors</b>	<b>2</b>
<b>2</b>	<b>Copy Constructors</b>	<b>9</b>
2.1	Where Do We Use a Copy Constructor? . . . . .	9
2.2	Compiler-Generated Copy Constructors . . . . .	10
2.3	Example: Bid: Compiler-Generated Copy Constructor . . . . .	10
2.4	Example: BidCollection: Compiler-Generated Copy Constructor . . . . .	11
2.5	Writing a BidCollection Copy Constructor . . . . .	17
2.6	Shallow & Deep Copying . . . . .	21
<b>3</b>	<b>Assignment</b>	<b>22</b>
3.1	Compiler-Generated Assignment Ops . . . . .	22
3.2	Implementing Assignment . . . . .	26
<b>4</b>	<b>The Rule of the Big 3</b>	<b>28</b>

### Copying Data Structures

Copying data is one of the most common operations in C++.

- Every time we write an assignment

```
x = y;
```

we are copying data.

- As our data gets more complicated, so does copying.

.....

### But, First...

A slight diversion...

.....

## 1 Destructors

### Destructors

Destructors are used to clean up objects that are no longer in use.

- A destructor for a class *C* is a function named *~C*
  - This function takes no parameters
- The most common action within a destructor is deleting pointers

.....

### Destructors are never Called Explicitly

The compiler generates all calls to destructors implicitly

- when a variable goes out of scope,
- or when we delete a pointer.

.....

### Implicit Destructor Call 1

If we write

```
void foo (int d, int c)
{
    Money m (d,c);
    for (int i = 0; i < 100; ++i)
    {
        Money mon2 = m;
        :
    }
}
```

the compiler generates

```
void foo (int d, int c)
{
    Money m (d,c);
    for (int i = 0; i < 100; ++i)
    {
        Money mon2 = m;
        :
        mon2.~Money();
    }
    m.~Money();
}
```

.....

### Implicit Destructor Call 2

If we write

```
void foo (int d, int c)
{
    Money* m = new Money (d,c);
    :
    delete m;
}
```

the compiler generates

```
void foo (int d, int c)
{
    Money* m = new Money (d,c);
    :
    m->~Money();
    free (m);
}
```

.....

### A Typical Destructor

We use destructors to clean up data “owned” by our structured types:

```
class BidCollection {  
  
    int MaxSize;  
    int size;  
    Bid* elements; // array of bids  
  
public:  
    BidCollection (int MaxBids = 1000);  
  
    ~BidCollection ();  
    :  
};
```

.....

### Implementing the Destructor

The destructor would be implemented as

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

.....

### Compiler-Provided Destructors

**If you don't provide a destructor for a class, the compiler generates one for you automatically.**

- The automatically generated destructor simply invokes the destructors for any data member objects.
- If none of the members have programmer-supplied destructors, does nothing.

.....

### A Compiler-Generated Destructor

- We have not declared or implemented a destructor for our *Time* class,

```
#ifndef TIMES_H
#define TIMES_H

#include <iostream>

/**
 * Times in this program are represented by three integers: H, M, & S, representing
 * the hours, minutes, and seconds, respectively.
 */

class Time {
public:
    // Create time objects
    Time(); // midnight
    Time (int h, int m, int s);

    // Access to attributes
    int getHours() const;
    int getMinutes() const;
    int getSeconds() const;

    // Calculations with time
    void add (Time delta);
    Time difference (Time fromTime);

    /**
     * Read a time (in the format HH:MM:SS) after skipping any
     * prior whitespace.
     */
    void read (std::istream& in);

    /**
     * Print a time in the format HH:MM:SS (two digits each)
     */
};
```

```
void print (std::ostream& out) const;

// Comparison operators
bool operator== (const Time&) const;
bool operator< (const Time&) const;

private:
    // From here on is hidden
    int secondsSinceMidnight;
};

inline
std::ostream& operator<< (std::ostream& out, const Time& t)
{
    t.print(out);
    return out;
}

inline
bool operator!= (const Time& t1, const Time& t2)
{
    return !(t1 == t2);
}

inline
bool operator> (const Time& t1, const Time& t2)
{
    return t2 < t1;
}

inline
bool operator<= (const Time& t1, const Time& t2)
{
    return !(t1 > t2);
}
```

```
inline
bool operator>= (const Time& t1, const Time& t2)
{
    return !(t1 < t2);
}

#endif // TIMES_H
```

– That's OK

- We have not declared a destructor for our *Bid* class

```
#ifndef BIDS_H
#define BIDS_H

#include <iostream>
#include <string>
#include "time.h"
#include "money.h"

//
// Bids Received During Auction
//

class Bid {

    std::string bidderName;
    Money amount;
    std::string itemName;
    Time bidPlacedAt;

public:

    Bid (std::string bidder, double amt,
         std::string item, Time placedAt);

    Bid();
```

## Copying Data

---

```
// Access to attribute
std::string getBidder() const {return bidderName;}
double getAmount() const {return amount;}
std::string getItem() const {return itemName;}
Time getTimePlacedAt() const {return bidPlacedAt;}

// Print a bid
void print (std::ostream& out) const;

// Comparison operators
bool operator== (const Bid&) const;
bool operator< (const Bid&) const;

};

inline
std::ostream& operator<< (std::ostream& out, const Bid& b)
{
    b.print(out);
    return out;
}

#endif
```

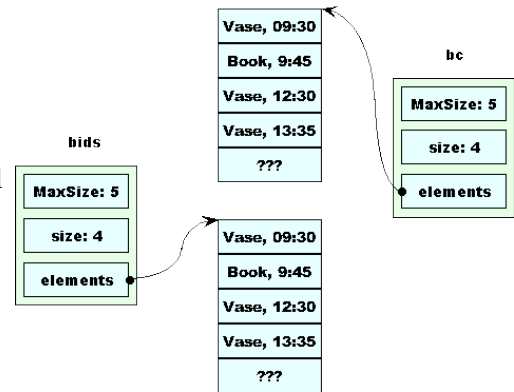
- Again that's OK
  - \* Despite the fact that the strings may need cleanup.

.....

## Trusting the Compiler-generated Destructor



What would happen if we trusted the compiler-generated destructor for *BidCollection*?



.....

**Compiler-generated destructors are wrong when...**

The compiler-generated destructor will leak memory when

- Your ADT has pointers among its data members, and
- Your ADT objects do not share those pointers with other objects.

.....

**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;
```

## Copying Data

---

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

## Copying Data

---

```
struct Bid {
    std::string bidderName;
    Money 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;
    Money 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 ();

    /**

```

## Copying Data

---

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

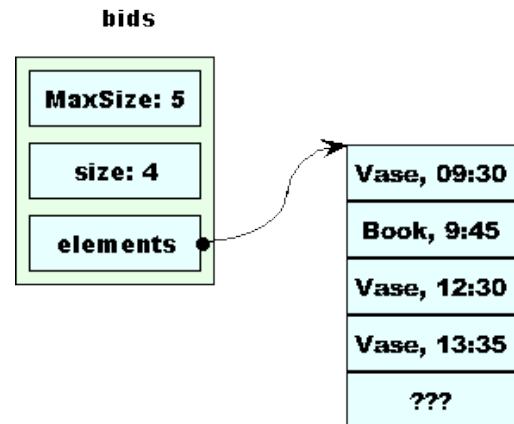
## Copying Data

```

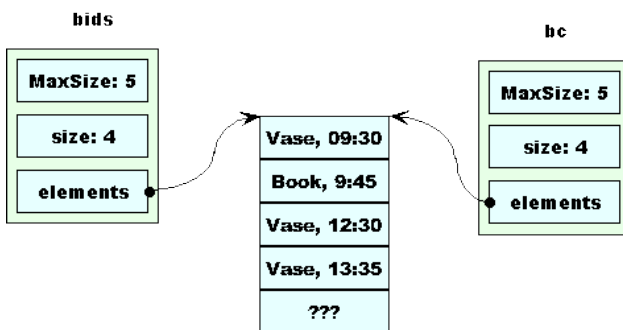
:
BidCollection afterNoonBids =
  removeEarly (bids, Time(12,0,0));

```

Assume we start with this in bids.



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



```

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

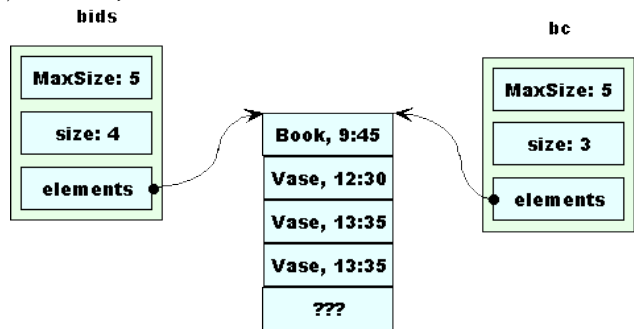
```

## Copying Data

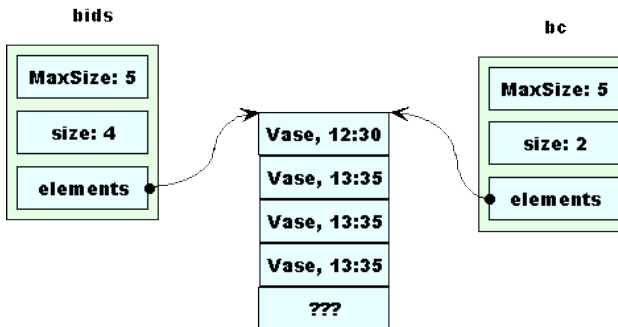
removeEarly removes the first morning bid from bc.

```

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



Then removeEarly removes the remaining morning bid from bc.



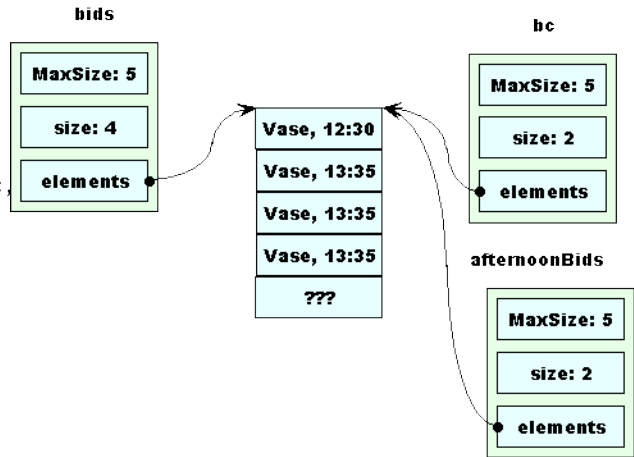
```

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

## Copying Data

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

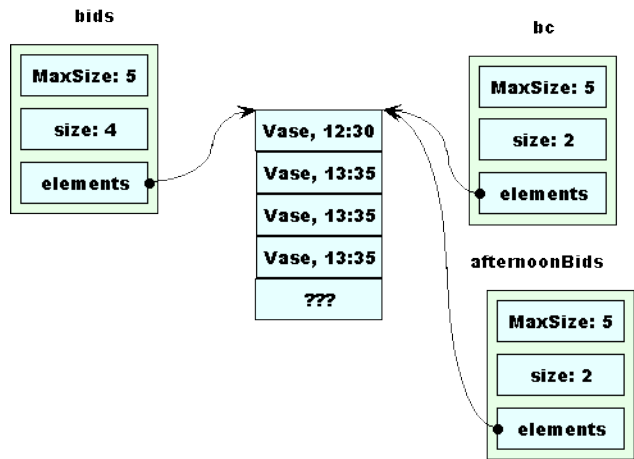
```
BidCollection removeEarly (BidCollection bc,
{
    :
    return bc;
}
:
BidCollection afternoonBids =
    removeEarly (bids, Time(12,0,0));
```



## 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 removeEarly,

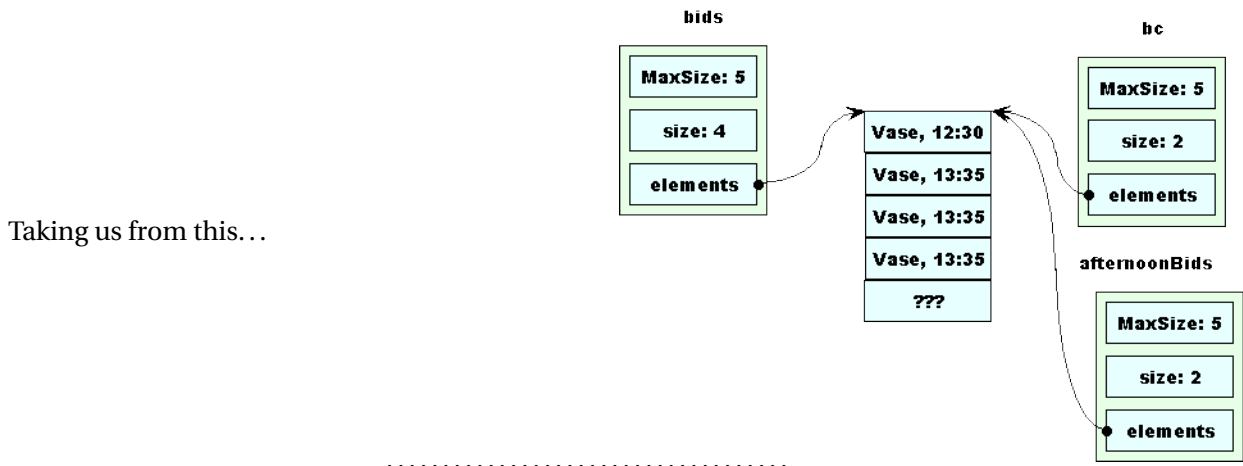
```
BidCollection removeEarly (BidCollection bc, Time t)
{
    for (int i = 0; i < x.size;)
    {
        if (bc.elements[i].bidPlacedAt < t)
            removeElement (elements, size, i);
        else
            ++i;
    }
}
```

## Copying Data

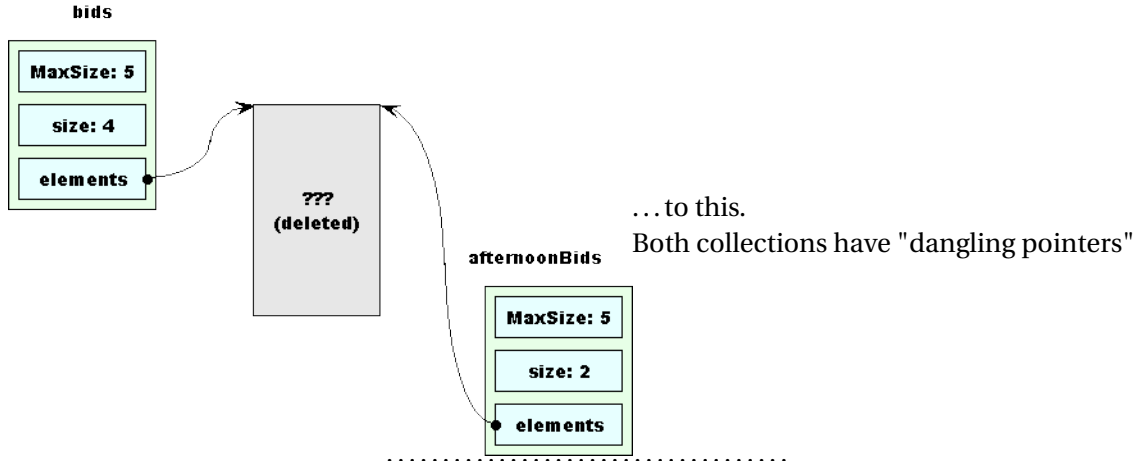
```
return bc;  
}
```

the destructor for BidCollection is called on bc

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



## What a Mess!



## Avoiding this Problem

We could



## Copying Data

---

- 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

.....

## 2.5 Writing a BidCollection Copy Constructor

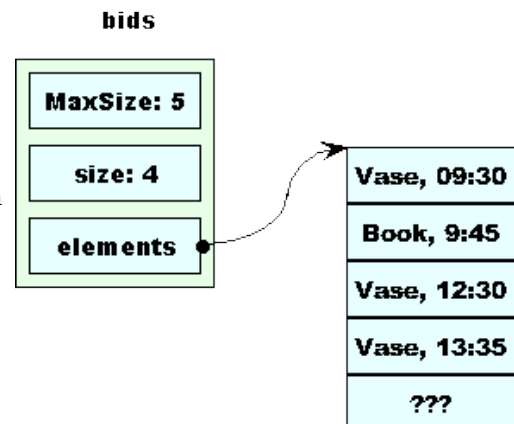
### Writing a BidCollection Copy Constructor

```
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];
}
```

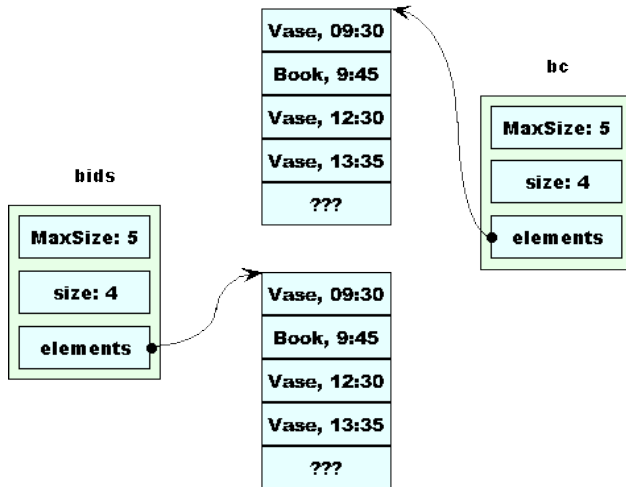
.....

### Once More, With Feeling!

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



## Copying Data



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

```

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

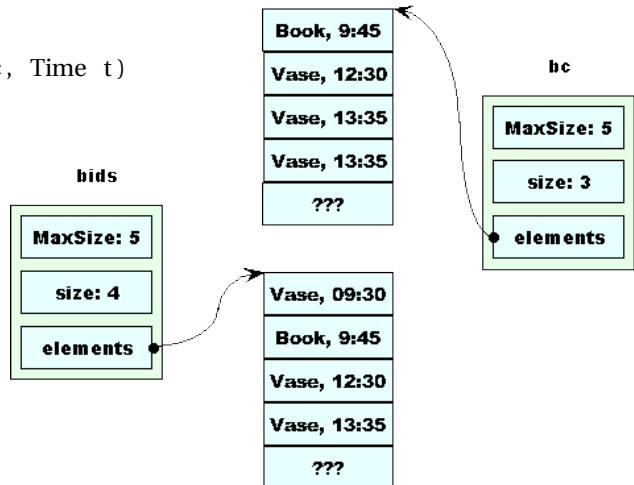
```

`removeEarly` removes the first morning bid from `bc`.

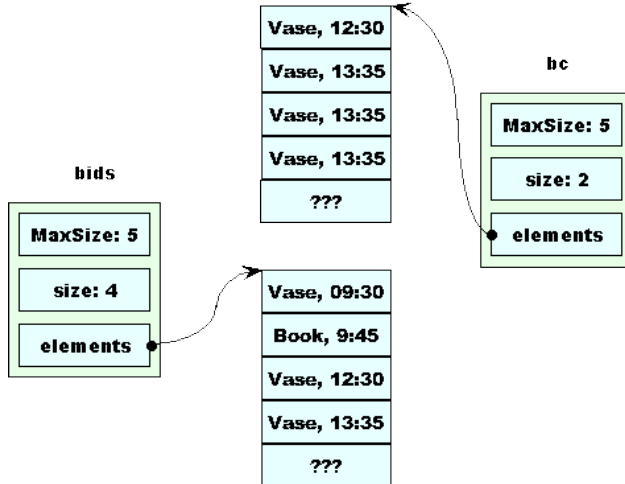
```

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

```



## Copying Data



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

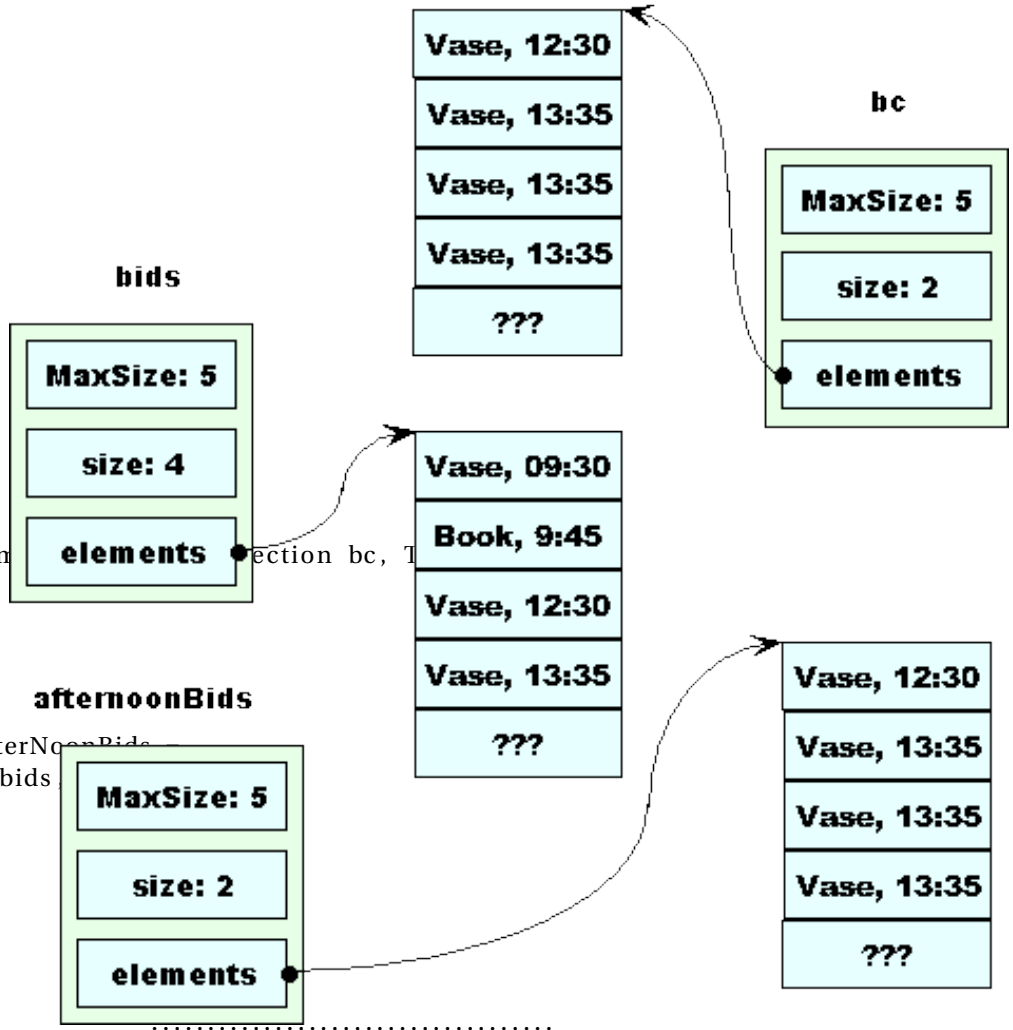
```
BidCollection removeEarly (BidCollection bc, Time t)
{
    for (int i = 0; i < bc.size; i++)
    {
        if (bc.elements[i].bidPlacedAt < t)
            removeElement (bc.elements, i);
        else
            ++i;
    }
    return bc;
}

BidCollection afterNoonBids =
    removeEarly (bids, Time(12,0,0));
```

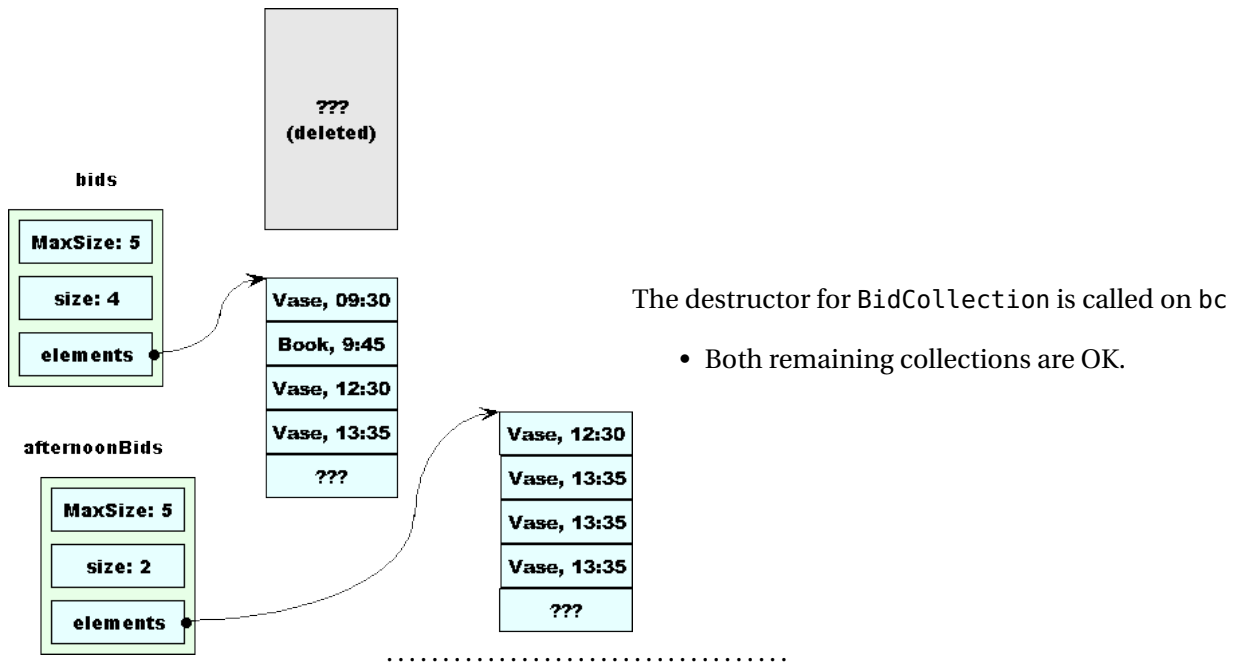
The return statement makes a copy of bc, which is stored in afterNoonBids. removeEarly removes the remaining morning bid from bc.

```

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



Much Nicer



The destructor for BidCollection is called on bc

- Both remaining collections are OK.

## 2.6 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*

.....

## Copying Data

### Example: BidCollection: Guess what happens

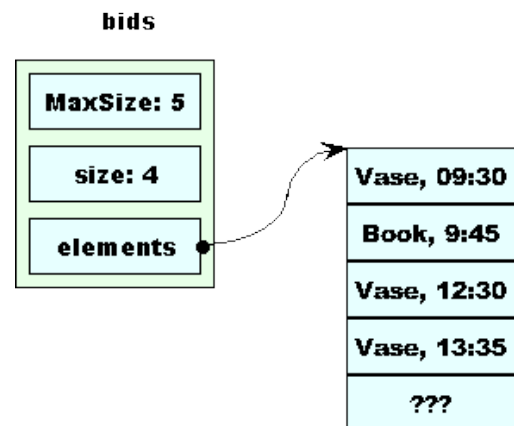
Our BidCollection class has no assignment operator, so the code below uses the compiler-generated version.

Suppose we had some application code:

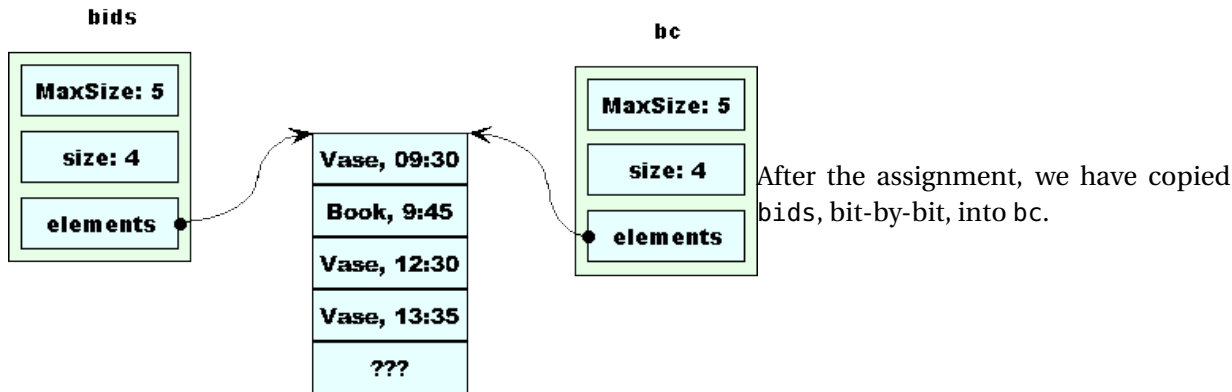
```
BidCollection bids;
:
BidCollection bc;
Time t (12, 0, 0);
bc = bids;
for (int i = 0; i < x.size;)
{
    if (bc.elements[i].bidPlacedAt < t)
        removeElement (elements, size, i);
    else
        ++i;
}

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

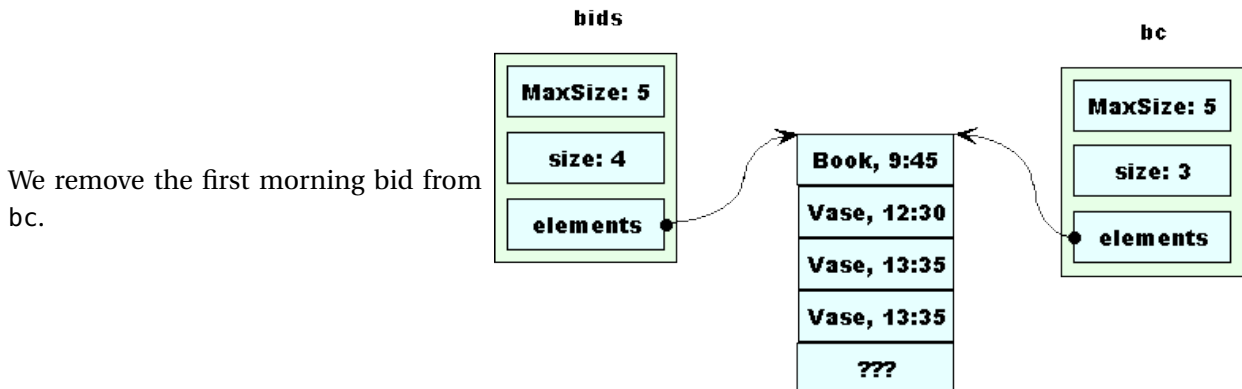
Assume we start with this in bids.



## Copying Data



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

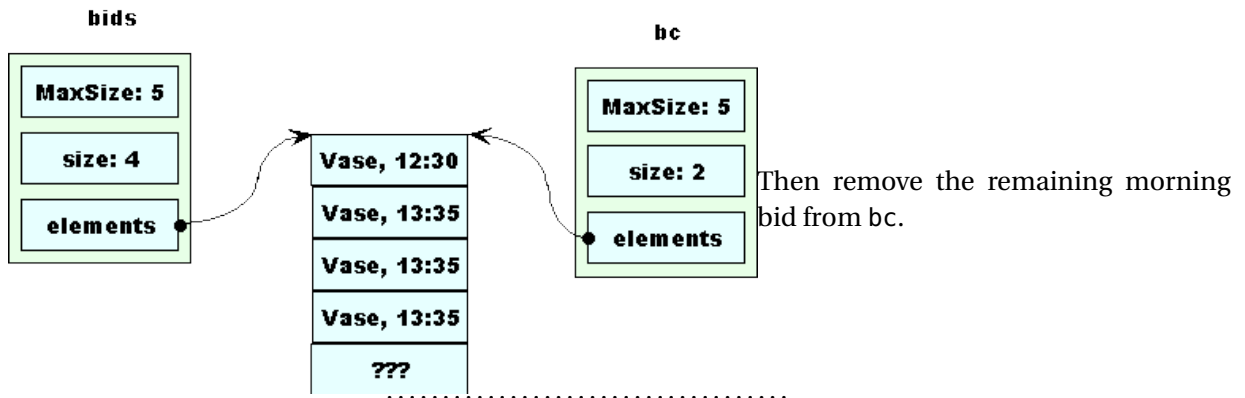


```
BidCollection bc;  
Time t (12, 0, 0);  
bc = bids;  
for (int i = 0; i < x.size;)  
{  
    if (bc.elements[i].bidPlacedAt < t)  
        removeElement (elements, size, i);  
}
```



```

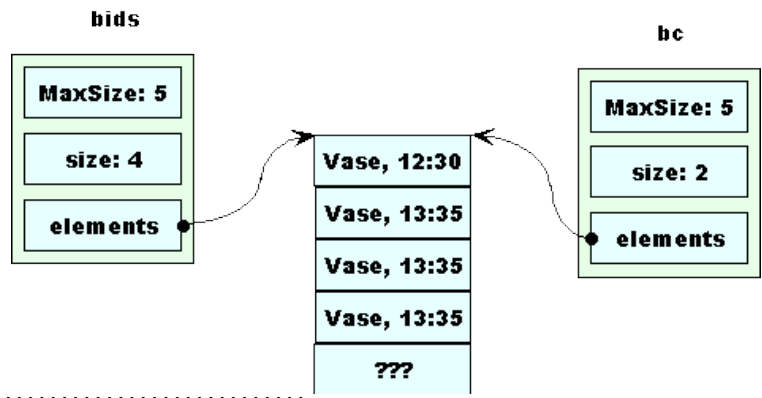
else
    ++i;
}
    
```



**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 have already seen that, for this class, we need to implement our own 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];
}
    
```

For all the same reasons, we will want to implement our own assignment operator.

## 3.2 Implementing Assignment

### Implementing Assignment

```
BidCollection& BidCollection::operator=
    (const BidCollection& bc)
{
    MaxSize = bc.MaxSize;
    size = bc.size;
    delete [] elements;

    elements = new Bid[MaxSize];
    for (int i = 0; i < size; ++i)
        elements[i] = bc.elements[i];

    return *this;
}
```

.....

### Why return \*this?

- By tradition in C++, assignments can be chained:

```
a = b = c = 0;
```

- Works because compiler interprets this as

```
a = (b = (c = 0));
```

- Not unlike <<

.....

### Self-Assignment

What would happen if we did

```
bids = bids;
```

using this assignment operator:

```
BidCollection& BidCollection::operator=
    (const BidCollection& bc)
{
    MaxSize = bc.MaxSize;
    size = bc.size;
    delete [] elements;
```

## Copying Data

---

```
elements = new Bid[MaxSize];
for (int i = 0; i < size; ++i)
    elements[i] = bc.elements[i];

return *this;
}
```

- Would delete all the elements before copying them

.....

### Self-Assignment – Really?

May seem an unlikely occurrence, but can often happen in a disguised form, e.g., during array processing,

```
bidSet[i] = bidSet[j];
```

- What if  $i == j$ ?

.....

### Coping with Self-Assignment

To allow for this possibility, most assignment operator functions check to see if the variable on the left (*this*) is at the same address as the variable on the right:

```
BidCollection& BidCollection::operator= (const BidCollection& bc)
{
    if (this != &bc)
    {
        MaxSize = bc.MaxSize;
        size = bc.size;
        delete [] elements;

        elements = new Bid[MaxSize];
        for (int i = 0; i < size; ++i)
            elements[i] = bc.elements[i];
    }
    return *this;
}
```

.....

## 4 The Rule of the Big 3

### A Question of Trust

- When do we not trust the compiler-provided copy constructor?
- When do we not trust the compiler-provided assignment operator?
- When do we not trust the compiler-provided destructor?

.....

### A Question of Trust

- When do we not trust the compiler-provided copy constructor?
  - When we have pointers among our data members, and
  - We don't want to share the data we are pointing to
- When do we not trust the compiler-provided assignment operator?
  - When we have pointers among our data members, and
  - We don't want to share the data we are pointing to
- When do we not trust the compiler-provided destructor?
  - When we have pointers among our data members, and
  - We don't want to share the data we are pointing to

Notice a pattern?

.....

### The Rule of the Big 3

- The copy constructor, assignment operator, and destructor are known as the *Big 3*.
- The *Rule of the Big 3* states:

*If you provide your own version of any one of the Big 3, you should provide your own version of all three.*

.....

**Providing the Big 3**

- The Bid ADT: `auctionBig3/bids.h` and `auctionBig3/bids.cpp`
- The Bidder ADT: `auctionBig3/bidders.h` and `auctionBig3/bidders.cpp`
- The Item ADT: `auctionBig3/items.h` and `auctionBig3/items.cpp`
- The Time ADT: `auctionBig3/time.h` and `auctionBig3/time.cpp`
- The BidCollection ADT: `auctionBig3/bidcollection.h` and `auctionBig3/bidcollection.cpp`
- The BidderCollection ADT: `auctionBig3/biddercollection.h` and `auctionBig3/biddercollection.cpp`
- The ItemCollection ADT: `auctionBig3/itemcollection.h` and `auctionBig3/itemcollection.cpp`

.....