Copy Constructors
and
Overloaded Assignment
When do we make copies of an object?

1) When passing them to a function by value
2) When returning them from a function by value
3) When creating a new object that is initialized with a copy of an existing object
4) When assigning objects \((x = y)\)

1. Items 1, 2 and 3 are handled by the copy constructor.
2. Item 4 is handled by overloading the assignment \((=)\) operator.
What is a copy constructor?

- It’s a constructor – it’s used to construct new objects.
- It does so by making a copy of an existing object.
- We can do so explicitly.

```cpp
// construct t1
Time t1 (12, 34, 56);

// construct t2 by copying t1
Time t2 (t1);

// construct t3 by copying t1
Time t3 = t1;
```

The compiler may make copies when it needs them.
Copy constructor syntax

The function prototype for every copy constructor is of the form:

```
ClassName::ClassName (const ClassName &);
```

Why is it necessary to for this parameter to be passed by reference? Why can’t it be passed by value?
Why haven’t we seen this before?

• The compiler provides a default copy constructor which up until now has been sufficient.

• The default copy constructor simply copies each of the data members from the existing object into the new object.

• This is not sufficient if one or more of the data members points to dynamically allocated memory.
Dynamic Memory Within a Class

- Sometimes a data member of a class points to dynamically allocated memory. When this occurs, we have to answer the following questions:

1. When will the dynamic memory be allocated?
2. When will the dynamic memory be deallocated?
3. What else is affected?
A potential security problem

• In more complex projects, you will often have pointers as private class members. It is critical that a copy constructor allocate new memory for each pointer in the new copy
  – The default copy constructor will just create a new pointer for the copy and give it the same value as the original pointer. So the copy and original will both point to the same data. This is a gross semantic error: changing the copy will change the original! How can this create a serious security problem?
A potential security problem

• Also, consider the “accessor” functions for your class, which exist simply to report the value of unchangeable private data members.
  – Why is it a security problem if your function returns a pointer to the data instead of a copy of the data? This can be a hard error to catch in languages like Java, which blur the distinctions of which variables are pointers and which are not.

• These are security problems, and debugging nightmares. Be careful to avoid them.
A Simple Array Class

One class that is often defined in C++ applications and libraries is a “smart” array. It has features that the built-in array doesn’t have such as automatic initialization and automatically checking indices to prevent a core dump. Other features are also possible.

We’ll use a such an array class to illustrate the impact of dynamic memory allocation within a class.
SmartArray

class SmartArray {
    public:
        SmartArray ( int size = 100 );
    // other members

    private:
        int m_size;
        int *m_theData;
};
Using SmartArray

Some SmartArray objects:

SmartArray a1 (50);  // 50 ints
SmartArray a2 (200);  // 200 ints
SmartArray a3;     // 100 ints by default
When Does the Memory Get Allocated?

The obvious answer to this question is, “In the constructor.”

```
SmartArray::SmartArray (int size)
{
    m_size = size;
    m_theData = new int [ m_size];
    for (int j = 0; j < m_size; j++)
        m_theData [ j ] = 0;
}
```
A Picture of Memory

Given the instantiation

SmartArray a1(50);

we get this picture of memory:
When Does the Memory Get Deallocated?

The intuitive answer is, “In the destructor.” The compiler provides us with a default destructor that deallocates the private data members. But this is not sufficient. If we relied on the default destructor, we’d create a memory leak because the memory pointed to by `m_theData` would not be freed.

```cpp
SmartArray::~SmartArray ( )
{
    delete [ ] m_theData;
}
```
What Else Is Affected?

• This time the answer is not so obvious.

• Consider the problems we want to avoid with dynamic memory:
  – dynamically allocated memory to which multiple things point (a probable logic error)
  – dynamically allocated memory to which nothing points (a memory leak)
  – a pointer that points “nowhere” (dangling pointer)

• All of these situations may arise when we wish to make a copy of a SmartArray object.
Effect of Default Copy Constructor (shallow copy)

Because the default copy constructor only copied the contents of `m_theData` in `a1` into `m_theData` in `a2`, both point to the array allocated in `a1`.

(Possible dangling pointer or other logic problem!)
The picture of memory we want (deep copy)

m_size
m_theData

a1:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>47</th>
<th>48</th>
<th>49</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>-1</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

a2:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>47</th>
<th>48</th>
<th>49</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>-1</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
SmartArray Copy Constructor

SmartArray::
SmartArray (const SmartArray& array)
{
    m_size = array.m_size;
    m_theData = new int [ m_size ];
    for (int j = 0; j < m_size; j++)
        m_theData[j] = array.m_theData [j];
}
When Is the Copy Constructor Invoked?

Silently by the compiler when we
- Pass by value:
  void someFunction(SmartArray array);
- Return by value:
  SmartArray someFunction(parameters)
  {
    SmartArray temp;
    // code manipulating "temp"
    return (temp);
  }
When Is the Copy Constructor Invoked? (cont’d)

- Explicitly by us upon construction
  
  ```cpp
  SmartArray a1;

  // constructing a2 as a copy of a1
  SmartArray a2 = a1;
  OR
  SmartArray a2(a1);
  ```
What’s an assignment operator?

- The assignment operator is the function `operator=`
- It’s called when we assign one existing object to another existing object

```cpp
Time t1 (12, 34, 56);
Time t2;
t2 = t1;  // object assignment
```
Why haven’t we heard of this before? (this may sound familiar)

• The compiler provides a default assignment operator, which up until now has been sufficient.
• The default assignment operator simply copies each of the data members from the existing object on the right hand side into the existing object on the left hand side.
• This is not sufficient if one or more of the data members points to dynamically allocated memory
Assigning SmartArray Objects

Consider the following code:

```cpp
SmartArray a1 ( 50 );
SmartArray a2 ( 50 );

// some code to manipulate a1
// some code to manipulate a2
// now assign a1 to a2
a2 = a1;
```
Assignment Operator

The statement

\[ a2 = a1; \]

calls the assignment operator ( operator= ) for the SmartArray class.

- If we don’t provide operator=, the compiler uses the default behavior
- Like the default copy constructor, the default assignment operator does a member-by-member (shallow) assignment.
Default Assignment Code

Conceptually, the default assignment operator for SmartArray contains the following code

\[
\begin{align*}
m\_size &= \text{rhs}.m\_size; \\
m\_theData &= \text{rhs}.m\_theData;
\end{align*}
\]
Prior To Assignment

We have a picture something like this:

```
Prior To Assignment

We have a picture something like this:

   a1:
     m_size  50
     m_theData  0 1 2 47 48 49
               4 0 12 ..

   a2:
     m_size  50
     m_theData  0 1 2 47 48 49
               0 5 42 ..
```

CMSC 202, Version 3/02

26
After Default Assignment

Because the default assignment operator only copied the contents of `m_theData` in `a1` into `m_theData` in `a2`, both point to the array allocated in `a1`. 
In Fact, It’s Worse

What happened to the memory that a2 used to point to? We’ve also caused a memory leak.

\[
\begin{array}{ccc}
0 & 1 & 2 \\
0 & 5 & 42 \\
\end{array}
\quad \ldots \quad
\begin{array}{ccc}
47 & 48 & 49 \\
6 & 58 & 0 \\
\end{array}
\]
We want this picture without a memory leak

- a1:
  - m_size: 50
  - m_theData:
    - 0: 4
    - 1: 0
    - 2: 12
    - ...: 47 48 49
    - ...: 40 -3 0

- a2:
  - m_size: 50
  - m_theData:
    - 0: 4
    - 1: 0
    - 2: 12
    - ...: 47 48 49
    - ...: 40 -3 0
SmartArray::operator=
(A First Attempt)

void SmartArray::operator= (const SmartArray& rhs)
{
    // free the memory for the current array
    delete [ ] m_theData;

    // now make a deep copy of the rhs
    m_size = rhs.m_size;
    m_theData = new int [ m_size ];
    for (int j = 0; j < m_size; j++)
        m_theData[ j ] = rhs.m_theData [ j ];
}
We’re Not Done Yet

Recall that it’s desirable for our objects to emulate the built-in types. In particular, we can do the following with built-in types:

```c
int bob, mary, sally;
bob = mary = sally; // statement 1
bob = bob;         // statement 2
(bob = mary) = sally; // statement 3
```

So our objects should also support these statements.
Analysis of These Statements

1. Statement 1 is a common thing to do – it’s called **cascading assignment**. To accomplish this, `operator=` must return a reference to a `SmartArray`.

2. Statement 2 is meaningless, but allowable by the language. To support this without causing a problem, `operator=` must check for this case (this is called **self-assignment**).

3. Statement 3 is odd, but valid. To support this, `operator=` must return a **non-const** reference to a `SmartArray`. (Debatable -- your text does not do this.)
SmartArray&    // non-const reference
SmartArray::operator= (const SmartArray& rhs) 
{
    if (this != &rhs)     // not  bob = bob
    {
        // free the memory for the current array
        delete [ ] m_theData;

        // make a copy of the rhs
        m_size = rhs.m_size;
        m_theData = new int [m_size ];
        for (int j = 0; j < m_size; j++)
            m_theData[ j ] = rhs.m_theData[ j ];
    }
    return (*this);     // for cascading assignment
}
Exercises For the Student

1) For the following statements, determine if the method called is a “regular” constructor, copy constructor, or operator=.

   a. SmartArray a1;
   b. SmartArray a2( a1 );
   c. SmartArray a3 = a2;
   d. SmartArray a4(100);
   e. a1 = a4;
Exercises For the Student (con’t)

2. Suppose operator= for SmartArray did not contain the statement if (this != &rhs); i.e., we allowed self-assignment. Draw the picture of memory that results from the following statements:

    SmartArray a1( 3 );
    a1 = a1;