COPYING OBJECTS

- Sometimes, you want to make a copy of an object
  - I.e., you have created a book, and you want to make 100 copies of that book for a book store

- What if one of the data members is a pointer?

```cpp
class Book {
    public:
        Book(char * t) { // constructor
            title = new char[strlen(t) + 1];
            strcpy(title, t);
        }
        void printTitle() { // print book title
            cout << title << endl;
        }
    private:
        char * title;
};
```
COPYING OBJECTS

What will happen if we just assign one object to another one?

```java
Book original("To Kill a Mockingbird");
Book copy = original; // what does this do?
copy.setTitle("The Hobbit"); // whose data changes?
```

Memberwise assignment: Every member will be copied

But... `title` is a pointer – `address` is copied, not data!

A shallow copy is created – both objects store the address of the same memory location
**Shallow VS Deep Copy & Pointers**

<table>
<thead>
<tr>
<th>Shallow Copy</th>
<th>Deep Copy</th>
</tr>
</thead>
<tbody>
<tr>
<td>➢ Copy the address of one variable into another variable</td>
<td>➢ Make a copy of the data and assign this data to a new variable</td>
</tr>
<tr>
<td>➢ Two variables refer to the same data stored in the same location in memory</td>
<td>➢ Two variables refer to two different locations in memory where the same data is stored</td>
</tr>
</tbody>
</table>

**Diagram:**
- **Shallow Copy**:
  - X refers to data
  - Y also refers to data

- **Deep Copy**:
  - X refers to data
  - Y also refers to data

© Pearson Education Limited 2015
COPY CONSTRUCTORS

A copy constructor is

- A special type of constructor used when a newly created object is initialized to the data of another object of same class

```cpp
Book original(“The Hobbit”);
Book copy(original); // this calls a copy constr
// parameter: another object
Book anotherCopy = original; // same
```

- C++ creates a default copy constructor if you don’t provide one – it simply does member-wise assignment
- Only works well when there’s no dynamic memory
PROGRAMMER-DEFINED COPY CONSTRUCTOR

- Allows us to create a **deep copy**:

```c++
Book(const Book &obj); // in Book.h

Book::Book(const Book &obj) // in Book.cpp
{
    title = new char[strlen(obj.title) + 1];
    strcpy(title, obj.title);
}
```

- Now the copy of the book will have its own title
- Copy constructor takes a **reference parameter** to an object of the class - make it `const` to make sure the original object is not modified
Each object now points to separate dynamic memory:

```java
Book original("The Hobbit");
Book copy = original;
copy.setTitle("Lord of the Rings");
```
PROGRAMMER-DEFINED COPY CONSTRUCTOR

- Since copy constructor has a reference to the object it is copying from,

  \[
  \text{SomeClass}::\text{SomeClass}(\text{SomeClass} &\text{obj})
  \]

  it can **modify that object**. It shouldn’t, though.

- To prevent this from happening, make the object parameter **const**:

  \[
  \text{SomeClass}::\text{SomeClass}(\textbf{const} \ \text{SomeClass} &\text{obj})
  \]
COPYING OBJECTS

- Copy constructors are used when a new object is created from an existing object:

  ```
  Book original("The Hobbit");
  Book copy = original; // invoke copy constr.
  ```

- What if both objects already exist, and we want to make one equal to the other one?

  ```
  Book tolkien1("The Hobbit");
  Book tolkien2("The Lord of the Rings");
  tolkien1 = tolkien2; // what will happen?
  ```
In C++, you can decide how operators work in the context of your class.

It is called operator overloading.

To make assignment meaningful for our class, we shall overload the := operator:

```cpp
void operator=(const SomeClass &rval);
```

Operator is called via object on left side.
**Overloaded Operators**

- **So how exactly does it work?**
  - The *name of the function for the overloaded operator* is `operator` followed by the operator symbol, *e.g.*, `operator=`
  - **Declare** overloaded operator function in the class (.h file)
  - **Implement** overloaded operator function with the rest of the member functions (.cpp file)

- Operator can be invoked as a member function:
  ```
  object1.operator=(object2);
  ```

- It can also be used in more conventional manner:
  ```
  object1 = object2;  // means the same,
  // looks better
  ```
IMPLEMENTING AN OVERLOADED OPERATOR

- Implement `operator=` for book – create a **deep copy**:

```cpp
void Book::operator=(const Book &obj)
{
    delete [] title; // del. previously allocated mem.
    title = new char[strlen(obj.title) + 1];
    strcpy(title, obj.title);
}
```

- Now the following is valid:

```cpp
Book tolkien1("The Hobbit");
Book tolkien2("The Lord of the Rings");
tolkien1 = tolkien2; // deletes "The Hobbit",
// replaces it by "The Lord of the Rings"
```
**Chaining Overloaded Operators**

- Primitive types allow to chain operators:
  - ```
  int x, y, z;
  y = z = x = 3;
  ```
- Can we do it with objects?
  - ```
  object1 = object2 = object3;
  ```
- Interpret the above as a nested function call:
  - ```
  object1.operator = (object2.operator = (object3));
  ```

`operator=` must return a value for this to work
CHAINING OVERLOADED OPERATORS

Interpret the nested function call:

- `object1.operator=(object2.operator=(object3));`

First look at: `object2.operator=(object3)`

Now consider: `object1.operator=(???)`;

Applied to an object

Another object sent as parameter

What must we put in place of `???' for this to work?
CHAINING OVERLOADED OPERATORS

- Re-write operator= for book:

```
Book& Book::operator=(const Book &obj)
{
    delete [] title;  // del. previously allocated mem.
    title = new char[strlen(obj.title) + 1];
    strcpy(title, obj.title);
    return *this;
}
```

- The function returns a reference to the current object
- Whichever object calls this function, it will return a reference to itself
**THE THIS POINTER**

- **this**: predefined pointer available to all class’s non-static member functions

- Always points to the instance (object) of the class whose function is being called

- Can be used to “un-hide” variable names:

```cpp
Book::Book(char * title) // re-use variable name
{
    this->title = new char[strlen(title) + 1];
    strcpy(this->title, title);
}
```
Can we overload other operators, such as +, -, <<?

Yes, we can! (Next lecture)

We can change the meaning of an operator, but we cannot change the number of operands of the operator.

- i.e., operator= will always take exactly one input.

Only certain operators can be overloaded. Cannot overload the following operators:

- ?: . .* :: sizeof

Questions?