

# **Copying Conversions Friends**

- Version 1: Dr. Ofir Pele
- Version 2: Dr. Erel Segal-Halevi

# Copy Constructor *(folder 1)*

- Called whenever an object of type T is copied.
- Copy Constructor to class T gets as argument **const T&** *(Why?)*
- You should consider for each class whether it needs **deep copy** or **shallow copy**.

# Rule of Three

- A rule of thumb:
  - When you need to make a deep copy of an object, you need to define all of these:
    1. Copy constructor
    2. Destructor
    3. Operator =
  - Or in other words:

when you need one, you need all.

# A skeleton for deep copy

**// Copy constructor**

```
A (const A& other) : init {  
    copy_other(other);  
}
```

**// Operator =**

```
A& operator=(const A& other) {  
    if (this!=&other) { // preventing problems in a=a  
        clear(); init // or recycle  
        copy_other(other);  
    } return *this; } // allows a= b= c= ...
```

**// Destructor**

```
~A() {  
    clear();  
}
```

# IntBuffer example (*folder 2*)

# Conversions of types is done in two cases:

1. Explicit casting (we'll learn more about it in next lessons)
2. When a function gets **X** type while it was expecting to get **Y** type, and there is a casting from **X** to **Y**:

```
void foo(Y y)
```

```
...
```

```
X x;
```

```
foo(x); // a conversion from X to Y is done
```

# Conversions danger: unexpected behavior

```
Buffer(size_t length) // ctor
```

...

```
void foo(const Buffer& v) // function
```

...

```
foo(3); // Equivalent to: foo(Buffer(3))
```

```
// Did the user really wanted this?
```

The Buffer and the size\_t objects are not logically the same objects!

# Conversion example (folder 4)



# User defined conversion (folders 5,6)

```
class Fraction {  
    ...  
    // double --> Fraction conversion  
    Fraction (const double& d) {  
        ...  
    }  
    ...  
    // Fraction --> double conversion  
    operator double() const {  
        ...  
    }  
}
```

**Operator Suffix** (folder 8)

**Operator Comma** (folder 9)

**friend**

# friend functions

Friend function in a class:

- Not a method of the class
- Have access to the class's private and protected data members
- Defined inside the class scope

Used properly does not break encapsulation

# friend functions example:

## Complex revisited

# friend classes

- A class can allow other classes to access its private data members
- *QUESTION: Is the friendship link one-sided or two-sided? I.e:*
  - *Suppose class A is a friend of class B.*
  - *Does it mean that class B is a friend of A?*

# friend classes - example

```
class IntTree {
```

```
...
```

```
friend class IntTreeliterator;
```

```
};
```

// Treeliterator can access Tree's data members

```
IntTreeliterator& IntTreeliterator::operator++() {
```

```
...
```

```
return *this;
```

```
}
```