Destructor

You can provide a destructor function that runs at the moment the object is removed from memory

- If object was created with “new”, then destructor runs when you call “delete”
- If the object was created by declaring a variable (automatic storage), then the destructor runs when the variable goes out of scope
- It runs **reliably**, unlike the finalize method from Java.

Useful for:

- Calling delete on data-members that were allocated with “new”
- Closing streams, etc.

Syntax is similar to a default constructor, but starts with the tilde (~) symbol:

- Default Constructor: Foo()
- Destructor: ~Foo()

- Destructors never take parameters

Example: NestedObjects

Lifecycle for nested objects:

Creation:

1. Constructor(s) for inner object(s) run
   - Member initializers allow customized construction
   - Without member initializers, default constructor(s) are invoked
2. Constructor for outer object runs

Removal:

1. Destructor for outer object runs
2. Destructor(s) for inner object(s) run

Friend

Normally private class members are only accessible within the class itself.

However… Individual access to private members can be granted via the **friend** keyword.

These could go in the header file for our class:

- `friend Foo;` // class Foo now can access our private members
- `friend void f()` // function f now can access our private members

Useful in certain cases of operator overloading.

Use this very sparingly!

Operator Overloading

Operators in C++ are functions that we can overload at will.

The only exceptions are:

- . (dot operator)
- .* (dot-star operator – what the heck is that???)
- ?:
- sizeof

Even when we overload an operator, we can't change syntax:

- Number and placement of operands
- Associativity
- Rules of precedence

Operator Overloading with a Member function

If the left-operand will be an instance of our class, then we may overload an operator with a member function:

We would like to do this:

```
IntWrapper x(3), y(4);
IntWrapper z = x + y;
```

Here is our overload of the + operator:

```
IntWrapper IntWrapper::operator+(IntWrapper const &rhs)
{
   int newValue = this->value + rhs.value;
   return IntWrapper(newValue);
}
```

Operator Overloading with a Non-Class Function

We want to do this:

```
IntWrapper x(3);
int y = 7 + x;
```

Since the left-operand of the operator is an int, we cannot use a member function. So we try this **non-member** function:

```
int operator+(int lhs, IntWrapper const &rhs)
{
   return lhs + rhs.value;
}
```

Problem: `value` is a private

Possible solutions:

- Use a public getter, e.g. `getValue()`
- Declare `operator+` as a friend of the IntWrapper class

Example: SimpleOperatorOverloading

- Overloading as member-function
- Overloading as non-class function
- Use of “friend”

Shallow vs. Deep Copy

If you do not write your own copy constructor or assignment operator, they are provided by the compiler.

The Bad News: The ones provided only do a shallow copy.

The Good News: In C++, usually a shallow copy is the same thing as a deep copy.

Shallow/Deep copies: Java vs. C++

```
class CatOwner
{
    string name;
    Cat pet;
}
```

Java:

Shallow copy and Deep copy are **different**

C++:

Shallow copy and Deep copy are **the same**

When Shallow is not Enough

```
class CatOwner
{
    string *name;
    Cat *pet;
}
```

When members are dynamically allocated, the built-in copy constructor and operator= are not going to be enough.

Example: BrokenClass

- **Copy constructor is not working**
- **Operator= is not working**

Remember that these two members are called implicitly by the system in many circumstances.

This is serious!

We must fix these two broken members.

- **By the way, we also have memory leaks.**