

## The Life of an Object

---

See also: Chapters 8, 9

### Revisiting SCell

The modified SCell class is now capable of handling both strings and doubles:

```
class SCell
{
    public:
        SCell(double initialValue);
        SCell(string initialValue);

        void setValue(double inValue);
        void setString(std::string inString);
        double getValue();
        std::string getString();

    protected:
        double mValue;
        std::string mString;
};
```

Notice that SCell now has two constructors. Here are their definitions:

```
SCell::SCell(double initialValue)
{
    setValue(initialValue);
}

SCell::SCell(string initialValue)
{
    setString(initialValue);
}
```

Since SCell has defined one or more constructors, the compiler-generated no-argument (default) constructor is no longer provided. Note the following two attempts to create an SCell:

```
SCell myCell; // BUG! No default constructor!
SCell myOtherCell(6.5); // Works fine!
```

```
SCell myThirdCell("test"); // This works too!
```

If we want users to be able to create an SCell without providing an initial value, we need to add a no-arg ctor:

```
class SCell
{
    public:
        SCell(); // no-arg (default) constructor
    [etc.]
}
```

The implementation of the default constructor would simply set the values to reasonable defaults:

```
SCell::SCell()
{
    mValue = 0;
    mString = "";
}
```

### Initializer Lists

The initializer list is a way for classes to set values for data members as they are constructed. This is fundamentally different from setting them inside the body of the constructor, even though it appears to be merely an alternate syntax:

```
SCell::SCell() : mValue(0), mString("")
{
}
```

### Dynamic Memory Allocation with Spreadsheet

The Spreadsheet class can have an arbitrary number of SCells placed on a grid whose size is provided at construction time:

```
class Spreadsheet
{
    public:
        Spreadsheet(int inWidth, int inHeight);

    protected:
        int mWidth, mHeight;
        SCell** mCells;
};
```

```

Spreadsheet::Spreadsheet(int inWidth, int inHeight) :
    mWidth(inWidth), mHeight(inHeight)
{
    mCells = new SCell* [mWidth];
    for (int i = 0; i < mWidth; i++) {
        mCells[i] = new SCell[mHeight];
    }
}

```

## Copy Constructors

Copy constructors are used behind the scenes in C++ all the time. When you pass an object into a function or method by value (not a pointer or reference), a copy is made using the copy constructor.

The *compiler generated copy constructor* is provided if you don't define one yourself (even if you have other non-copy constructors). It works by calling the copy constructor of all of your data members to make the copy. For members that are basic types (int, pointers, etc.) it simply copies the values.

Here is how to declare and define a copy constructor for Spreadsheet:

```

class Spreadsheet
{
    public:
        Spreadsheet(int inWidth, int inHeight);
        Spreadsheet(const Spreadsheet& src);    // Copy Constructor

    [etc...]
}

```

```

Spreadsheet::Spreadsheet(const Spreadsheet& src)
{
    int i, j;

    mWidth = src.mWidth;
    mHeight = src.mHeight;

    mCells = new SCell* [mWidth];
    for (i = 0; i < mWidth; i++) {
        mCells[i] = new SCell[mHeight];
    }
}

```

```

    for (i = 0; i < mWidth; i++) {
        for (j = 0; j < mHeight; j++) {
            mCells[i][j] = src.mCells[i][j];
        }
    }
}

```

When you dynamically allocate memory in your classes, it is essential to write a copy constructor so that copied objects aren't sharing memory. The alternative is to disable copying by making the copy constructor private. However, this means you can't pass the object by value.

### Destructors

Destructors provide objects with an opportunity to clean up when they are destroyed. When your object is destroyed, the destructors on your individual data members will be called automatically, in reverse order of creation.

The destructor is where you can clean up any heap memory, close file handles, or do any other sort of cleanup:

```

Spreadsheet::~~Spreadsheet()
{
    for (int i = 0; i < mWidth; i++) {
        delete[] mCells[i];
    }

    delete[] mCells;
}

```

### Assignment

In C++, assignment and construction are two separate things. If you have the statement `a=b`, that's assignment. Neither `a` nor `b` are being constructed. However, if you have the statement `Foo a = b`, that's construction and the copy constructor will be used to create `a`.

To prescribe behavior for assignment, you need to implement the `operator=` method:

```

class Spreadsheet
{
public:
    Spreadsheet(int inWidth, int inHeight);

```

```
Spreadsheet(const Spreadsheet& src);
~Spreadsheet();

Spreadsheet& operator=(const Spreadsheet& rhs);
```

[etc..]

The implementation of `operator=` is very similar to the copy constructor and they often (always?) should share code, which can be *refactored* into a separate method. There are three main differences between `operator=` and the copy constructor:

- `operator=` should check for self assignment. If the user is setting foo equal to foo, nothing needs to be done.
- `operator=` may have to clean up in a destructor-like fashion to avoid orphaning memory.
- `operator=` should return a reference to the destination object, so that assignment can be chained (`a = b = c`)

Here is the implementation of `operator=` for Spreadsheet:

```
Spreadsheet& Spreadsheet::operator=(const Spreadsheet& rhs)
{
    int i, j;

    // Check for self-assignment.
    if (this == &rhs) {
        return (*this);
    }

    // Free the old memory. Could share code with destructor.
    for (i = 0; i < mWidth; i++) {
        delete[] mCells[i];
    }

    delete[] mCells;

    // Copy the new memory. Could share code with copy ctor.
    mWidth = rhs.mWidth;
    mHeight = rhs.mHeight;

    mCells = new SCell* [mWidth];
```

```

    for (i = 0; i < mWidth; i++) {
        mCells[i] = new SCell [mHeight];
    }

    for (i = 0; i < mWidth; i++) {
        for (j = 0; j < mHeight; j++) {
            mCells[i][j] = rhs.mCells[i][j];
        }
    }

    return (*this);
}

```

### A Proper C++ Class

It's generally believed that for a C++ class to be complete, it must have a copy constructor, operator=, and a destructor. This is certainly true when you have dynamically allocated memory. For the SCell class, the built-in operator=, copy constructor, and destructor suffice. For Spreadsheet, using the defaults would result in shared memory and memory leaks.

If you are writing a class that *should* have custom versions of these methods but you don't want to write them, you do have a choice. You can explicitly disable copying and assignment by making them private:

```

class Spreadsheet
{
public:
    Spreadsheet(int inWidth, int inHeight);
    ~Spreadsheet();

private:
    Spreadsheet(const Spreadsheet& src);
    Spreadsheet& operator=(const Spreadsheet& rhs);
};

```

Of course, this means that you can't pass a Spreadsheet by value into a function. And there's no way to avoid writing the destructor. If you don't write one, the memory allocated for the grid of SCells will be orphaned when the object is deleted.

## Object Lifecycle Reference

The following table shows the various compiler-generated constructors and when they come into play:

If you define . . .	. . . then the compiler generates . . .	. . . and you can create an object . . .	Example
[no constructors]	A 0-argument constructor A copy constructor	With no arguments. As a copy of another object.	<pre>SCell cell; SCell myCell(cell);</pre>
A 0-argument constructor only	A copy constructor	With no arguments. As a copy of another object.	<pre>SCell cell; SCell myCell(cell);</pre>
A copy constructor only	No constructors	Theoretically, as a copy of another object. Practically, you can't create any objects.	No example.
A single-argument (noncopy constructor) or multiargument constructor only	A copy constructor	With arguments. As a copy of another object.	<pre>SCell cell(6); SCell myCell(cell);</pre>
A 0-argument constructor as well as a single-argument (noncopy constructor) or multiargument constructor	A copy constructor	With no arguments. With arguments. As a copy of another object.	<pre>SCell cell; SCell myCell(5); SCell cellB(cell);</pre>