

Pimp Your Classes

See also: The middle part of Chapter 9 (194-208), Chapter 12

Pretty much any Object-Oriented Language lets you create data members and methods, but only C++ gives you a catalog of keywords you can apply to your classes to create special features. This handout summarizes some of the favorites, both useful and obscure.

The static Keyword

`static`, much like our friend `const`, can be used in several different ways in C++. One of the uses is *static data members*, which are data members that exist not on a per-object basis, but a per-class basis. Below is a static data member in the Spreadsheet class:

```
class Spreadsheet
{
public:
    // Omitted for brevity
protected:
    bool inRange(int val, int upper);
    void copyFrom(const Spreadsheet& src);

    int mWidth, mHeight;
    SpreadsheetCell** mCells;

    static int sCounter;
};
```

Static data members need to be explicitly constructed. This generally goes in the `.cpp` file:

```
// Spreadsheet.cpp
int Spreadsheet::sCounter = 0;
```

The static can be accessed within a method or constructor, just like any other data member. The only difference is that there is only one `sCounter`, no matter which object is looking at it:

```
Spreadsheet::Spreadsheet(const Spreadsheet& src)
{
    sCounter++;
    copyFrom(src);
}
```

Static data members can also be accessed from outside the class (if they're public) by using the scope resolution operator (::). No instance of the class is needed because a static isn't bound to a particular object, it's bound to the class.

Methods can be static as well. A method that doesn't apply to a particular instance of a class (for example, a utility method like a conversion routine) can be declared static:

```
class SpreadsheetCell
{
    public:
        // Omitted for brevity

    protected:
        static string doubleToString(double val);
        static double stringToDouble(const string& str);

        // Omitted for brevity
};
```

When a method is static, it cannot access non-static data members. Static methods aren't specific to an object, so it would be impossible to determine which object's data members to access. The following line won't compile:

```
// SpreadsheetCell.cpp
string SpreadsheetCell::doubleToString(double val)
{
    mValue = 7; // BUG! A static method can't access non-static data!
}
```

If they are public, static methods can be accessed by other classes, again using the scope resolution operator:

```
// main.cpp
string s = SpreadsheetCell::doubleToString(14);
```

const – Aren't we done with this yet?

Not quite! Imagine the possibilities if we combine static and const! We can put a constant right in a class and make it externally accessible without an instance of the object.

```
class Spreadsheet
{
    public:
        // Omitted for brevity

        static const int kMaxHeight = 100;
        static const int kMaxWidth = 100;
};
```

```

    protected:
        // Omitted for brevity
};

```

Note that unlike other data members, C++ lets you initialize const data members inside the class definition if they are of basic types. Otherwise, you have to do it externally, just like initializing any other static:

```

// Spreadsheet.cpp
const int Spreadsheet::kMaxHeight = 100;
const int Spreadsheet::kMaxWidth = 100;

```

Remember that methods can also be const, indicating that they do not change the underlying object:

```

double getValue() const; // getters are often const

```

mutable – const's nemesis

Sometimes, you'll write a method that is *logically* const, but actually needs to change the class. This modification has no effect on any user-visible data, but is technically a change, so the compiler won't let you declare the method const. For example, if we wanted to keep track of the number of times `getValue()` was called, we couldn't do it if `getValue()` is const:

```

double SpreadsheetCell::getValue() const {
    mNumberOfTimesAccessed++; // BUG! Can't change data member in const method
}

```

By making `mNumberOfTimesAccessed` *mutable*, we can tell the compiler, "It's okay. This data member can be changed by a const method":

```

class SpreadsheetCell
{
public:
    // Omitted for brevity

protected:
    mutable int mNumberOfTimesAccessed;

    // Omitted for brevity
};

```

Edge Cases with References

We've already covered most of the ways that references are used (such as const references – those are important!), but here are a few other tidbits about refs:

- References cannot be bound to a non-variable (`int& ref = 2` won't compile) unless the reference itself is `const` (`const int& ref = 2` will compile)
- References act like pointers, but they aren't pointers (`int& xref = &x` won't compile). Don't be confused by the `&` sign – it doesn't mean address of!
- You *can* have a pointer to a reference (`int*& myPtrRef = intPtr` is okay) but you *cannot* have a pointer to a reference (`int&*& myRefPtr = &myRef` won't compile) or a reference to a reference (`int&& myRefRef = &myRef` won't compile)
- References can be used as data members, but they must be set in the initializer list.

friend – The feature that only pretends to be your friend

Sometimes you feel like you want to get around the whole public versus protected thing and just grant an individual class, method, or function the right to call a method. The `friend` keyword lets you do this, but you should use it sparingly – it's often a sign that your class structure has broken down. Here is an example of a friend class. The `SpreadsheetCell` class is allowing the `Spreadsheet` class to use any of its protected and private methods and members:

```
class SpreadsheetCell
{
    public:
        friend class Spreadsheet;

        // Remainder of the class omitted for brevity
};
```

Method Overloading

In C++, you can have two versions of the same method as long as they differ in their argument lists. For example, `setValue()` and `setStringValue()` could *both* be called `set()`:

```
class SpreadsheetCell
{
    public:
        ...

        void set(double inValue);
        void set(const string& inString);
        double getValue() const;
        string getString() const;

        // Remainder of the class omitted for brevity
};
```

Note that method overloading does *not* apply to return types. You cannot have two methods with the same name that only differ in their return type:

```
class SpreadsheetCell
{
    public:
        ...

        double get() const;
        string get() const; // BUG! There is already a get() method

        // Remainder of the class omitted for brevity
};
```