

Implementing Templates and Smart Pointers

See also: Chapter 11, Chapter 16 (449 – 454), Chapter 25 (736-741)

Creating Templates and Implementing Smart Pointers

CS193D 2/27/06

Remember This Example?

Instead of a GameBoard that stores GamePieces on a 2-D grid, let's have a Grid that stores *anything* in a 2-D grid.

```
// Grid.h
template <typename T>
class Grid
{
    [etc.]
}
```

The template keyword tells the compiler that the class is parameterized by type. A new version of the class can be created for any typename T.

```

template <typename T>
class Grid {
    public:
        Grid(int inWidth, int inHeight);
        Grid(const Grid<T>& src);
        ~Grid();
        Grid<T>& operator=(const Grid<T>& rhs);

        void setElementAt(int x, int y, const T& inElem);
        T& getElementAt(int x, int y);
        const T& getElementAt(int x, int y) const;
        int getHeight() const { return mHeight; }
        int getWidth() const { return mWidth; }
        static const int kDefaultWidth = 10;
        static const int kDefaultHeight = 10;

    protected:
        void copyFrom(const Grid<T>& src);
        T** mCells;
        int mWidth, mHeight;
};

```

Implementing Templated Methods

The template specifier goes before each implementation:

```
template <typename T>
Grid<T>::Grid(int width, int height) : mWidth(width),
                                     mHeight(height)
{
    mCells = new T* [mWidth];
    for (int i = 0; i < mWidth; i++) {
        mCells[i] = new T[mHeight];
    }
}
[etc.]
```

```
template <typename T>
void Grid<T>::setElementAt(int x, int y, const T& elem)
{
    mCells[x][y] = inElem;
}
```

Revisiting Selective Instantiation

```
class NoCopies {
    private:
        NoCopies& operator=(const NoCopies& src);
};

template <typename T>
class Foo {
    public:
        dontCopy(T input) { cout << "not copying!" << endl; }
        tryToCopy(T input) { mCopy = input; }

    protected:
        T mCopy;
};

Foo<NoCopies> myFoo;
NoCopies myNoCopies;
myFoo.dontCopy(myNoCopies); // This will compile
```

Where to Place Your Template Code

The compiler needs to "see" the raw code when using a template.

Option 1: Put all the code in the .h file(s)

Option 2: Have the .h file include the .cpp file:

```
// Grid.h

template <typename T>
class Grid {
    [etc]
};

// Weird, but acceptable
#include "Grid.cpp"
```

Multiple Template Parameters

You can have non-type template parameters which are basically just placeholders for values:

```
template <typename T, int WIDTH, int HEIGHT>
class Grid
{
    public:
        [etc]
        int getHeight() const { return HEIGHT; }
        [etc]
    protected:
        T mCells[WIDTH][HEIGHT]; // No more dynamic memory!
};
```

The advantage of template parameters over constructor arguments is that they can be used at compile-time.

The Downside of Non-Typename Template Parameters

- 1) Since they're compile-time values, they must be literals or const:

```
int j = getInteger();  
Grid<int, 17, j> myGrid; // BUG! j is not a constant
```

- 2) They actually create a new type for every combination used.

```
Grid<int, 3, 4> myGrid;  
Grid<int, 2, 2> yourGrid;  
  
myGrid = yourGrid; // BUG! They're different types!
```

Templatized Methods

You can assign an int to a double, but you can't assign a Grid<int> to a Grid<double>... unless you templatize operator= on something other than T.

```
template<typename T>
class Grid {
    public:
        Grid(const Grid<T>& src); // Keep the original!
        template <typename E>
        Grid(const Grid<E>& src);

        Grid<T>& operator=(const Grid<T>& rhs); // Keep!
        template <typename E>
        Grid<T>& operator=(const Grid<E>& rhs);
};
```

Templatized Method Implementation

```
template <typename T>
template <typename E>
Grid<T>::Grid(const Grid<E>& src)
{
    mWidth = src.getWidth();
    mHeight = src.getHeight();

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

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

About Method Templates

- Related to selective instantiation – it works for `Grid<int>` and `Grid<double>` because an `int` can be assigned to a `double`.
- Compiler support for method templates is sometimes weak.
- The original version of `op=` or copy ctor is still required! Otherwise, the compiler will generate the built-in one.
- Even non-type parameters can be used for method templates:

```
// allow copying grid of a different type & size!  
template <typename E, int WIDTH2, int HEIGHT2>  
Grid(const Grid<E, WIDTH2, HEIGHT2>& src);
```

Template Class Specialization

```
// Special Grid for C-style strings
template <>
class Grid<char*>
{
    [Entire implementation for char*s]
};

// Implementation
char* Grid<char*>::getElementAt(int x, int y) const {
    char* ret = new char[strlen(mCells[x][y]) + 1];
    strcpy(ret, mCells[x][y]);
    return ret;
}
```

This class is *completely* separate from our templated Grid class. It just reuses the name.

Subclassing and Templates

You can subclass the template itself (or rather, each instantiation), as another template:

```
template <typename T>
class GameBoard : public Grid<T> {
    public:
        void move(int srcX, int srcY, int dstX, int dstY);
};
```

You can also subclass a particular instantiation:

```
class ChessBoard : public Grid<ChessPiece> {
};
```

Use inheritance to add functionality or for polymorphism. Use specialization when the implementation differs for a particular type.

Function Templates

Templates aren't limited to classes and methods (much as you might wish they were). You can templatize a function:

```
template<typename T>
int Find(T& value, T* arr, int size) {
    for (int i = 0; i < size; i++) {
        if (arr[i] == value) {
            return i;
        }
    }
    return -1;
}
```

The compiler can even deduce the template type:

```
result = Find<int>(i, myArray, 10); // This works
result = Find(i, myArray, 10); // So does this!
```

More Fun with Templates

Multiple type parameters:

```
template <typename T, typename Container>
class Grid { ... };
```

Default template type parameters:

```
template <typename T, typename Container = vector<T> >
```

Template template parameters: (not a typo!)

```
template <typename T, template <typename E>
class Container = vector >
[... ]
    Container<T>* mCells;
```

Zero-Initialization

How do you refer to the empty value for a type T?

```
template <typename T>
Grid<T>::Grid(int width, int height) : mWidth(width),
                                     mHeight(height) {
    mCells = new T* [mWidth];
    for (int i = 0; i < mWidth; i++) {
        mCells[i] = new T[mHeight];
        for (int j = 0; j < mHeight; j++) {
            mCells[i][j] = T(); // zero-initialization
        }
    }
}
```

T() will use 0 (zero) for non-objects and the default constructor for objects.

If You're Interested in Learning More about Templates...

- Try partial specialization (specialize for *some* of the params):

```
template<int, WIDTH, HEIGHT>  
class Grid<char*, WIDTH, HEIGHT>
```

- Try specializing for just pointer types:

```
template<typename T>  
class Grid<T*>
```

- Look into recursive templates (Grid of Grids)

```
RecursiveGrid<int, 7> sevenMinuteAbs;
```

- Visit your local library (or the textbook)

Smart Pointers

The heap is a dangerous place.

- Unclear who owns memory
- Risk of memory leaks
- Risk of double deletion

The stack is not.

- Objects cleaned up when they go out of scope
- Ownership limited to current scope
- No explicit allocation, or deleting

How can we leverage the benefits of the stack without giving up the flexibility of the heap?

- 1) Avoid dynamic memory. Use the STL whenever possible, make liberal use of copies and assignments, pass by reference when necessary.
- 2) Use dynamic memory, but surround each renegade pointer with a more predictable stack-based object.

A *smart pointer* is a stack-based object that manages an underlying pointer. The smart pointer itself takes care of the tricky memory tasks associated with the heap-based memory.

The STL's auto_ptr

```
class Simple {
    public:
        Simple() { mPtr = new int(); }
        ~Simple() { delete mIntPtr; }
    protected:
        int* mIntPtr;
};

void leaky() {
    Simple* mySimplePtr = new Simple();
    // BUG! mySimplePtr is never deleted
}

void notLeaky() {
    auto_ptr<Simple> mySimplePtr(new Simple());
    // OK! The auto_ptr will clean up
}
```

The transparency of auto_ptr

Once you have an auto_ptr, you can forget about it:

```
auto_ptr<Simple> mySimplePtr(new Simple());
```

```
mySimplePtr->someMethod();
```

```
Simple anotherSimple = *mySimplePtr;
```

Smart pointers protect against memory leaks. When the stack-based smart pointer is destructed, it deletes the associated memory.

One Drawback of auto_ptr

The STL `auto_ptr` does not perform *reference counting*. If you have two `auto_ptr`s controlling the same pointer, you'll end up with double deletion:

```
{
    Simple* mySimplePtr;

    auto_ptr<Simple> ptr1(mySimplePtr);
    auto_ptr<Simple> ptr2(mySimplePtr);

    // BUG! Double deletion of mySimplePtr!
}
```

As pointers are assigned, copied, stored in objects, etc., the risk of double deletion increases.

Starting Our Own Smart Pointer Class

Start with a simple templated class that manages a pointer:

```
template <typename T>
class Pointer
{
    public:
        explicit Pointer(T* inPtr);
        ~Pointer();

        operator void*() const { return mPtr; }

    protected:
        T* mPtr;

    private: // prevent assignment and copying
        Pointer(const Pointer<T>& src);
        Pointer<T>& operator=(const Pointer<T>& rhs);
};
```

Implementation of Pointer Management

```
template <typename T>
Pointer<T>::Pointer(T* inPtr)
{
    mPtr = inPtr;
}
```

```
template <typename T>
Pointer<T>::~~Pointer();
{
    delete mPtr;
}
```

We're simply writing a stack-based class that takes care of heap-based memory.

Adding Pointer Semantics

```
template <typename T>
class Pointer {
    public:
        [etc]
        T& operator* ();
        const T& operator* () const;
    [etc]
};
```

```
template <typename T>
T& Pointer<T>::operator* () {
    return *mPtr;
}
```

```
template <typename T>
const T& operator* () const {
    return *mPtr;
}
```

What About operator-> ?

Someone could use -> to refer to a member *or* a method. How can you handle that?

Turns out that C++ translates `foo->set(5)` into:

```
(foo.operator->())->set(5); // it does -> for you!
```

So our implementation is:

```
template <typename T>
T* Pointer<T>::operator->()
{
    return mPtr;
}
```

```
// TODO: implement both const and non-const versions
```

Using Our Smart Pointer

We've just implemented the basic functionality of `auto_ptr`:

```
// Create
Pointer<Simple> myPtr(new Simple());

// Call method with ->
myPtr->someMethod();

// Access data member with ->
cout << "myPtr's foo is " << myPtr->foo << endl;

// Copy underlying object with *
Simple otherSimple = *myPtr;
```

Adding Reference Counting

First, let's allow copying and assignment since we know we'll always clean up properly. Also keep a map for the reference counting.

```
template <typename T>
class Pointer {
    public:
        [etc]
        Pointer(const Pointer<T>& src);
        Pointer<T>& operator=(const Pointer<T>& rhs);

    protected:
        T* mPtr;
        static std::map<T*, int> sRefCountMap;

        void finalizePointer();
        void initPointer(T* inPtr);
};
```

Ref Counting Implementation

```
Pointer<T>::Pointer(T* inPtr) {  
    initPointer(inPtr);  
}
```

```
Pointer<T>::Pointer(const Pointer<T>& src) {  
    initPointer(src.mPtr);  
}
```

```
Pointer<T>& Pointer<T>::operator=(const Pointer<T>& rhs) {  
    if (this == &rhs) {  
        return *this;  
    }  
    finalizePointer();  
    initPointer(rhs.mPtr);  
    return *this;  
}
```

Note: template <typename T> has been omitted from these examples

```

Pointer<T>::~~Pointer() {
    finalizePointer();
}

void Pointer<T>::initPointer(T* inPtr) {
    mPtr = inPtr;
    if (sRefCountMap.find(mPtr) == sRefCountMap.end()) {
        sRefCountMap[mPtr] = 1;
    } else {
        sRefCountMap[mPtr]++;
    }
}

void Pointer<T>::finalizePointer() {
    sRefCountMap[mPtr]--;
    if (sRefCountMap[mPtr] == 0) {
        sRefCountMap.erase(mPtr);
        delete mPtr;
    }
}

```

Testing the New Functionality

```
class Simple {
    public:
        Simple() { sNumAllocations++; }
        ~Simple() { sNumDeletions++; }

        static int sNumAllocations;
        static int sNumDeletions;
};

int Simple::sNumAllocations = 0;
int Simple::sNumDeletions = 0;
```

```
void testPointer()
{
    Simple* mySimple = new Simple();

    {
        Pointer<Simple> ptr1(mySimple);
        Pointer<Simple> ptr2(mySimple);
    }

    if (Simple::sNumAllocations != Simple::sNumDeletions) {
        failed();
    } else {
        passed();
    }
}
```