

Using STL Containers to Implement a Sparse Matrix Class

April 22, 2016

Sparse Matrices

- ▶ A sparse (square) matrix is a matrix that has a number of non-zero coefficients proportional to n rather than n^2
- ▶ It is not convenient to store all entries including zeros
- ▶ It is not convenient to include zero entries when doing algebra with sparse matrices

A Sparse Matrix Class Based on STL Containers

- ▶ Consider the following class:

```
class sparse_matrix : public std::vector<std::map<unsigned int, double> >
```

- ▶ What happens if we do:

```
A = sparse_matrix (4); auto x = A[2][2]
```

- ▶ What happens if we do:

```
A = sparse_matrix (4); A[2][2] = 1.0;
```

- ▶ Cool! This features make

```
class sparse_matrix : public std::vector<std::map<unsigned int, double> >  
a great candidate for the implementation of a (row oriented)  
sparse matrix!
```

Useful things to know about the `std::map` container

- ▶ `mapped_type& std::map::operator[] (const key_type& k);`
A call to this function is equivalent to:
`((*(this->insert(make_pair(k,mapped_type()))).first)).second`
- ▶ A similar member function, `std::map::at`, has the same behavior when an element with the key exists, but throws an exception when it does not.
- ▶ `iterator find (const key_type& k);`
`const_iterator find (const key_type& k) const;`
Searches the container for an element with a key equivalent to `k` and returns an iterator to it if found, otherwise it returns an iterator to `map::end ()`.
- ▶ `size_type count (const key_type& k) const;`
Searches the container for elements with a key equivalent to `k` and returns the number of matches. Because all elements in a map container are unique, the function can only return 1 (if the element is found) or zero (otherwise).

A recap example about inheritance I

```
#include <iostream>
```

```
class A_t
```

```
{
```

```
public:
```

```
void
```

```
A_m () { std::cout << "calling A_t::A_m" << std::endl; };
```

```
virtual
```

```
void
```

```
B_m () { std::cout << "calling A_t::B_m" << std::endl; };
```

```
};
```

```
class B_t :
```

```
public A_t
```

```
{
```

```
public:
```

```
void
```

```
B_m () { std::cout << "calling B_t::B_m" << std::endl; };
```

A recap example about inheritance II

void

```
A_m () { std::cout << "calling _B_t::A_m" << std::endl; };
```

int

```
main (void)
```

```
{
```

```
    B_t b;
```

```
    A_t *a = &b;
```

```
    // b is of class B_t, derived from A_t
```

```
    // therefore it has method A_m definide in B_t
```

```
    // but also the implementation defined in A_m
```

```
    b.A_m ();
```

```
    b.A_t::A_m ();
```

```
    // Also A_t::B_m even though it is virtual can be
```

```
    // accessed via a qualifier
```

```
    b.A_t::B_m ();
```

A recap example about inheritance III

```
// a is a pointer to an object of class A_t  
// if I invoke method A_m the compiler chooses  
// the implementation given in A_t  
a->A_m ();
```

```
// to access the method B_t::A_m I need to use a cast  
static_cast<B_t*> (a) -> A_m ();
```

```
// If I invoke method B_m through  
// the pointer a, instead, the compiler chooses B_t::B_m  
// instead of A_t::B_m because A_t::B_m is virtual  
a->B_m ();
```

```
// to access method A_t::A_m I need to add a qualifier  
a->A_t::A_m ();
```

```
return 0;  
};
```

Exercises

- ▶ Implement a library providing a `sparse_matrix` class inheriting from `std::vector<std::map<unsigned int, double> >`
 - ▶ A basic implementation is already given as a starting point
 - ▶ Write the makefile
 - ▶ Adapt `fem1d` to use this class
- ▶ Create an abstract interface class `general_matrix` and derive both `matrix` and `sparse_matrix` from it
- ▶ Change `fem1d` so that the type of matrix can be selected by the user *at run time*.