CS 294-73
Software Engineering for Scientific Computing

Lecture 7: Introduction to STL Containers
Function and Operator Overloading

- We are using \(+, *, -, /, \ldots\) with many different types of arguments, different meanings in different contexts.
  - Familiar in all programming languages: \(a*b\) is understood for \(a, b\) integers, floats, …

- C++ takes this to the limit consistent with static strong typing.
  - Operators: binary infix \((x, *, \ldots)\), prefix / postfix \((++, --)\), operator precedence is tied to the operator, but otherwise we can define them anyway we want.
  - Function names can also be overloaded
  - Uniquely determined by types of arguments, return values, classes.

- One more level of overloading can be obtained by using namespaces.
Examples of Overloading

- `ostream& operator<<(ostream&, const T&)`
  cout << t << ... ;

- `void BoxData<T,N>::operator*=(const T&);`  
  void BoxData<T,N>::operator*=(const RectMDArray<T,N>);`

- `const T& BoxData<T,N>::operator()(const Point&, int) const;`  
  T& BoxData<T,N>::operator()(const Point&, int);

- Member function names. In fact, you want to use the same member function names for analogous functionality across multiple classes (see later with iterators).
Standard Template Library.

Predefined classes: aggregates that are templated on the type being held.
Example of a namespace. The names of these classes are `std::className`.
We use the command
using namespace std;
in global scope to tell compiler to look for functions of the form `std::className`. Some authorities view this as bad form.
http://www.cplusplus.com/
NB: C++11 standard.
Three Examples

Container templates in the STL.

- C arrays as first-class objects (array),
- dynamic arrays (vector),
- Many others, which we will discuss later.

Shared pointers.

To use these, you need to include the appropriate header file, e.g.

```cpp
#include <array>
#include <vector>
#include <memory>
```
Array\langle T, N \rangle, \text{ pair}\langle T_1, T_2 \rangle

- Why not int foo[3], rather than Array\langle int, 3 \rangle foo?
  - array\langle int, 3 \rangle is a type—objects of this type can be returned, assigned, etc.
  - array\langle int, 3 \rangle tupleFunction(...) // perfectly ok.

- pair: lots of circumstances where you need to hand around pairs of objects of different classes.
  - pair\langle T_1, T_2 \rangle pr = make_pair(t1, t2);
  - pr.first
  - pr.second
vector<int> foo;
for (int k = 0; k < 10; k++)
{
    foo.push_back(k);
}
for (auto it=foo.begin(); it != foo.end(); ++it)
{
    cout << *it << endl;
}
vector<T>

Several new things:
• Classes declared inside of classes. What things can be declared inside of a class A?
  - Functions void A::bar(...)  
  - Data a.m_foo (one per object); A::s_bar (static, one per class).  
  - Classes: A::Aprime;

• vector<T>::iterator is a class member of vector<T>. Abstracts the idea of location in a linearly-ordered set.
  - it = vec.begin(); Calls a member function of vector<T> that returns an object of class vector<T>::iterator, initialized to initial location in vec.
  - it.end() == true if you have reached the end of vec.
  - ++it, --it increments, decrements the location by one.
  - *it returns a reference to the contents at the current location in vec.
  - You could have gotten the same functionality by an ordinary loop and indexing, but only for vector, not for the other containers.
vector<T>

- auto
  - (vector<T>::iterator it = vec.begin(); ... is a lot of keystrokes.
  - auto <varname> = ...; can be used instead of a type declaration if the type can be inferred unambiguously from the right-hand side at compile time. In this case, vector<T>::begin() has not been overloaded, i.e. there is only one member function with that name and no arguments, and its return type is vector<T>::iterator.
  - auto can be used for many other things than this. For readability and self-documentation, it is probably best not to overuse it (Compilers can find meaningful interpretations of what may be typographical errors).
Adding, deleting, accessing elements of vector

unsigned int size();

push_back(const T&);

pop_back(const T&);

T& back();

T& front();

operator[](int);

Vector<T>::iterator begin()
• Looks like a 1D array: can index any element by an integer less than size().
• Can add, delete elements at the end of an array.
• Fast access: data stored in contiguous locations in memory (just as if you had used new. In fact, you can access the underlying contiguous storage as an ordinary 1D array.)
vector<int> foo;
for (int k = 0; k < 10; k++)
    { foo.push_back(k); }
for (auto it=foo.begin();it != foo.end(); ++it)
    { cout << *it << endl; }

Slightly different from our use of iterator in Box – for vector<T> *it returns a T&.
How do remove an element from a vector?

- Can do this at the end easily (pop_back), but in general:
  - find the element you wish to remove
  - make a whole new vector 1 smaller than the original
  - copy all but the excluded object to the new vector

- But we have already been doing something almost as awful with the push_back function of vector:
  - grow vector length by one
  - copy all elements to the new vector with length+=1
  - copy the new element on the end
  - (in reality vector is doing a version of doubling it’s size when it runs of of room and keeps track of it’s “real” size and it’s size() )

- Vectors are good at:
  - Accessing individual elements by their position index (constant time).
  - Iterating over all the elements (linear time).
  - Add and remove elements from its end (constant amortized time).
Question: how do we debug memory errors?

• There are tools (valgrind / memcheck) – may discuss them later.
• The best way to is to design your code so that you don’t (can’t) make them.
Question: how do we debug memory errors?

- There are tools (valgrind / memcheck) – may discuss them later.
- The best way to is to design your code so that you don’t (can’t) make them.
  - Use STL containers that manage memory for you (vector<T>).
  - Disciplined memory management: new only in constructors, delete only in destructors.
  - STL shared_ptr (and other memory management STL tools).
Memory errors

```cpp
{
    T* foo = new T[10];
}

Memory leak: out of scope, so can’t delete memory foo pointed to.

```cpp```cpp
{
    T* foo = new T[10];
    ...
    foo[20] = 1.0;
}

Out of bounds access – corrupts data at memory location foo+20.
Wrapping pointered data in a class...

BoxData<T,N>::BoxData()
{
};
BoxData<T,N>::BoxData(const Box& a_bx)
{
define(a_bx);
}
BoxData<T,N>::define(const Box& a_bx)
{
    m_data = new T[a_bx.size()]*N};
BoxData<T,N>::~BoxData()
{
delete [] m_data;};
...eliminates a class of memory leaks.

```c
{
    BoxData<int> A(bx);
    ...
}
```

When A goes out of scope, destructor is called, and memory is reclaimed.

However, what we did was not quite right, because we don’t use strong construction in BoxData. The following is a memory leak.

```c
{
    BoxData<int> A(bx);
    ...
    A.define(bx2)
}
```
Wrapping pointered data with declare / define.

BoxData<T,N>::BoxData()
{m_isDefined = false;m_data = NULL;};
BoxData<T,N>::BoxData(const Box& a_bx)
{define(a_bx)};
BoxData<T,N>::define(const Box& a_bx)
{if (m_isDefined) {delete [] m_data;} m_isDefined = true;
m_data = = new T[a_bx.size()]N];;
BoxData<T,N>::~BoxData()
{if (m_isDefined) {delete [] m_data;}};
Disallowing shallow copies.

What does assignment do?

```
{  
  BoxData<int> A,B;  
  ...  
  A = B;  // copies data members.  
}

When you go out of scope, A and B each call delete on m_data. ???
```
But sometimes you want shallow copies...

BoxLayout
{
...
std::vector<Box> m_boxes;
}

BoxLayoutData<T>
{
...
BoxLayout m_bl;
}

You may declare many BLD variables for a single BoxLayout. Making copies becomes a significant memory overhead, plus you will want to know when two BLDs have the same BoxLayout, e.g. for copying (“Shared metadata”).
std::shared_ptr<T>

std::shared_ptr<Counter> foo;
foo = shared_ptr<Counter>(new Counter);
{
    std::shared_ptr<Counter> foo2 = foo;
    foo->incrementCounter();
    foo2->incrementCounter();
}
    cout << foo->getCounterValue() << endl;
// a->b means (*a).b .
std::shared_ptr<T>

```cpp
std::shared_ptr<Counter> foo;
{
    std::shared_ptr<Counter> foo2;
    foo2 = shared_ptr<Counter>(new Counter);
    foo = foo2;
    foo->incrementCounter();
    foo2->incrementCounter();
}
    cout << foo->getCounterValue() << endl;
```
What is shared_ptr doing?

Every newly created object has a counter.
Shallow copies increment the counter by one.
Deleted copies by going out of scope decrement the counter by one.
When counter reaches zero, delete is called.
(Poor person’s garbage collection).

There’s also a version of this corresponding to allocating with \texttt{new T[N]}, but we won’t be using it anytime soon.
BoxLayout
{
...
std::shared_ptr<std::vector<Box> > m_boxes;}
BoxLayoutData<T>
{
...
BoxLayout m_bl;}

BoxLayoutData<T>::BoxLayoutData(BoxLayout a_bl)
{
m_bl = a_bl; ...};

BoxLayoutData<BoxData> A;
{
    BoxLayout myBl = ...;
    BoxLayoutData<BoxData> B(myBl);
    A.define(myBl);
}
What about out-of-bounds access?

If you are wrapping your memory-managed data in a class, you will know its size at the time you define it. So put in bounds checking as an assertion so that it is checked when in debug mode.

BoxData<T,N>::operator()(Point a_p)
{
    assert(m_box.contains(a_p));
    ...
}

Back to Poisson’s equation.
Some Vector Calculus

Gradient operator: \( \nabla \Psi = \left( \frac{\partial \Psi}{\partial x}, \frac{\partial \Psi}{\partial y} \right) \)

Divergence operator: \( \nabla \cdot (F_x, F_y) = \frac{\partial F_x}{\partial x} + \frac{\partial F_y}{\partial y} \)

Laplacian: \( \Delta \phi = \nabla \cdot (\nabla \phi) = \frac{\partial^2 \Psi}{\partial x^2} + \frac{\partial^2 \Psi}{\partial y^2} \)

- Green’s theorem (aka integration by parts)

\[- \int_{\Omega} \Psi(x)(\nabla \cdot (\nabla \phi))(x)dx = \int_{\Omega} \nabla \Psi \cdot \nabla \phi dx + \int_{\partial \Omega} \Psi(x)(\nabla \phi)(x)dS \]

- If \( \Psi \equiv 0 \) on \( \partial \Omega \), then

\[- \int_{\Omega} \Psi(x)(\nabla \cdot (\nabla \phi))(x)dx = \int \nabla \Psi \cdot \nabla \phi dx \]
We want to solve Poisson’s equation (note the sign convention)

\[-\Delta \phi = f \text{ on } \Omega\]
\[\phi = 0 \text{ on } \partial \Omega\]

We want find a weak solution, i.e.

\[\int_{\Omega} (-\Delta \phi)(\boldsymbol{x}) \Psi(\boldsymbol{x}) d\boldsymbol{x} = \int_{\Omega} f(\boldsymbol{x}) \Psi(\boldsymbol{x}) d\boldsymbol{x} \text{ on } \Omega\]

For all continuous piecewise smooth test functions

\[
\cdot \Psi(\boldsymbol{x}) \text{ with } \Psi = 0 \text{ on } \partial \Omega
\]

Applying Green’s Theorem, this is the same as

\[\int_{\Omega} \nabla \phi \cdot \nabla \Psi d\boldsymbol{x} = \int_{\Omega} f(\boldsymbol{x}) \Psi(\boldsymbol{x}) d\boldsymbol{x} , \Psi \in V\]
Finite element discretization

Step 1: we discretize our domain as a union of triangles.

Step 2: We replace \( V \) by \( V^h \), a finite-dimensional space of test functions. For this exercise, we will use linear combinations of continuous, piecewise linear functions, indexed by interior nodes, linear on each triangle containing the node. A basis for this space is given by \( \{ \Psi^h_n(x) : n \in N_I \} \).

Step 3: We also approximate the solution as a linear combination of the the elements in \( V^h \).

Interior Nodes = \( N_I \)
Elements \( e = 0, \ldots E - 1 \)

\[
\Psi^h_n(x_{n'}) = \delta_{nn'}, \quad n' \in N
\]
We apply the weak form of the equations to the finite-dimensional subspace $V^h$

$$\phi(x) \approx \phi^h(x) = \sum_{n \in N_I} a_n \Psi^h_n(x)$$

$$\int_\Omega \nabla \phi^h \cdot \nabla \Psi^h_n dx = \int_\Omega f(x) \Psi^h_n(x) dx, \quad n \in N_I$$

$$\sum_{n' \in N_I} a_{n'} \int_\Omega \nabla \Psi^h_{n'} \cdot \nabla \Psi^h_n dx = \int_\Omega f(x) \Psi^h_n(x) dx$$

$$(La)_n = \sum_{n' \in N_I} L_{n,n'} a_{n'} = b_n$$

$$L_{n,n'} = \int_\Omega \nabla \Psi^h_{n'} \cdot \nabla \Psi^h_n dx, \quad b_n = \int_\Omega f(x) \Psi^h_n(x) dx$$
Matrix Assembly

Pseudocode: Interior Nodes = \mathcal{N}_I, Elements e = 0, \ldots E - 1

Initialize \( L = 0 \)

for \( e = 0 \ldots E - 1 \)
  for \( (x_n, x_{n'}) \in K_e : (n, n') \in \mathcal{N}_I \)
    \[ L_{n,n'} = \int_{K_e} \nabla \Psi_{n'}^h \cdot \nabla \Psi_n^h dx \]
  endfor
endfor

- \( L \) is a matrix with mostly zero entries. But it is nice: symmetric, positive-definite, M-matrix.
- \( \nabla \Psi_n^h \) is a constant vector, easily computed.
- We’re building a matrix dimensioned by nodes by iterating over elements and building it up incrementally.
Getting the right-hand side

Quadrature for $b$: midpoint rule on each element.

$$\int_{\Omega} \Psi^h_n f \, d\mathbf{x} = \sum_{K_e} \int_{K_e} \Psi^h_n f \, d\mathbf{x}$$

$$\int_{K_e} \Psi^h_n f \, d\mathbf{x} \approx \text{Area}(K_e) f(\mathbf{x}_{e\text{ centroid}}) \Psi^h_n(\mathbf{x}_{e\text{ centroid}})$$

Initialize $b = 0$

for $e = 0 \ldots E - 1$

for $\mathbf{x}_n \in K_e : n \in \mathcal{N}_I$

$$b_n := \text{Area}(K_e) f(\mathbf{x}_{e\text{ centroid}}) \Psi^h_n(\mathbf{x}_{e\text{ centroid}})$$

endfor

endfor

Fig 1.9 The basis function $\varphi_j$.

$$\mathbf{x}_{e\text{ centroid}} = \frac{1}{\text{Area}(K_e)} \int_{K_e} \mathbf{x} \, d\mathbf{x}$$

More element magic.
Matrix Properties

Our matrix has the following properties:

• Symmetric, positive-definite: \( L = L^T \), \( v \cdot (Lv) > 0 \) if \( v \neq 0 \)

• Positive along diagonal. \( L_{k,k} \geq -\sum_{k'} L_{k,k'} \)

• Rows sum to a non-negative number:

• For triangles sufficiently close to equilateral, the nonzero off-diagonal elements are non-negative, i.e. \( \nabla \Psi_n^h \cdot \Psi_{n'}^h \leq 0 \) on \( K_e \).
Point Jacobi Iteration

Motivation: to solve $La = b$, we compute it as a steady-state solution to an ODE.

$$\frac{da}{dt} = -La + b$$

If all of the eigenvalues of $L$ are positive, then

$$La_\infty = b , \ a_\infty = \lim_{t\to\infty} a(t)$$

Point Jacobi: use forward Euler to solve ODE.

$$a^{l+1} = a^l + \lambda(b - La^l) , \ l = 0, 1, \ldots ; \ a^0 = 0 \ \ \lambda > 0$$

Stop when the residual has been reduced by a suitable amount.

$$||b - La^l|| \leq \epsilon||b||$$
Choosing a Relaxation Parameter

This leads to the following choice for our relaxation parameter.

\[ \lambda = \alpha \frac{1}{\max_k L_{k,k}}, \quad \alpha < 1 \]

If your grid is strongly-varying, may want to use a local relaxation parameter (you will not be doing this in the present assignment).

\[ a_{k}^{l+1} = a_{k}^{l} + \lambda_k (b - L a_{k}^{l})_k \]
\[ \lambda_k = \alpha \frac{1}{L_{k,k}} \]