MAT 685: C++ for Mathematicians

Containers

John Perry

University of Southern Mississippi

Spring 2017

Outline

Outline

Containers

Container: data structure that "contains" multiple instances of some type

Containers

Container: data structure that "contains" multiple instances of some type

Example (Arrays!)

```
int A[25];
```

`A` "contains" 25 `int`'s.

Outline

STL

"Standard Template Library:" substantial variety of containers

- **templates** are... well, "templates" for code
  - code reuse w/different types
  - implementation in *header* file
  - instantiate particular types by declaration
  - compiler "fills in" methods corresponding to type
- **iterators** simplify code reuse and data access
- compiler *must know* types

Rest assured for now

Goal is to *describe* and *use* templates. Creating comes later!

Example: gcd template

Listing 1: `gcd_template.hpp` (extract, 1/2)

```
template <typename T> // T stands in for a type
T gcd(T a, T b) {
  if (a < 0) a = -a;
  if (b < 0) b = -b;

  if ( (a == 0) and (b == 0) ) {
    cerr << "WARNING: gcd called with two zeros.\n";
    return 0;
  }

  if (b == 0) return a;
  if (a == 0) return b;

  T c = a % b; // declare c of type T

  return gcd(b, c);
}
```

Example: gcd template

Listing 2: `gcd_template.hpp` (extract, 2/2)

```
template <typename T>
T gcd(T a, T b, T & x, T & y) {

  // passed over the error checks

  T c = a % b; // many vars of type T
  T q = a / b;

  T d = gcd(b, c, x, y); // here, too
  T new_x = y;
  T new_y = x - q*y;
  x = new_x;
  y = new_y;
  if (a_neg) x = -x;
  if (b_neg) y = -y;
  return d;
}

#endif
```

Example: gcd template

Listing 3: `test_xgcd_template.cpp`

```
#include <iostream>
using std::cin; using std::cout; using std::endl;
#include "gcd_template.hpp"

// instantiate for T = int64_t
template<>
int64_t gcd<int64_t>(int64_t, int64_t);

int main() {
  int64_t a, b, x, y;
  cout << "Enter two numbers: " << endl;
  cin >> a >> b;
  cout << "gcd(" << a << "," << b << ") = ";
  cout << gcd(a,b,x,y) << " = ";
  cout << a << '*' << x;
  cout << " + " << b << '*' << y << endl;
}
```

Compile, execute

```
$ g++ -o test_xgcd_recursive \
        test_xgcd_template.cpp
$ ./test_xgcd_recursive
Enter two numbers:
123456789 987654321
gcd(123456789,987654321) = 9 =
123456789*-8 + 987654321*1
```

Outline

Our focus

We focus on a few. More in book, online.

- list-like: array, vector, list
  - "in a row"
  - sortable
  - multiple copies of element allowed

Our focus

We focus on a few. More in book, online.

- list-like: array, vector, list
  - "in a row"
  - sortable
  - multiple copies of element allowed
- set-like: set, unordered_set
  - "in a bag"
  - non-sortable (w/caveats)
  - one copy of element only

Our focus

We focus on a few. More in book, online.

- list-like: array, vector, list
  - "in a row"
  - sortable
  - multiple copies of element allowed
- set-like: set, unordered_set
  - "in a bag"
  - non-sortable (w/caveats)
  - one copy of element only
- map-like: map, unordered_map
  - "in relation"
  - non-sortable (doesn't make sense, really)
  - one copy of element only

Which is better?

Depends on situation

- Fast access, *any* position?
  - fixed size? array
  - needs to expand? vector
- Easily expandable?
  - ordered? list
  - unordered? set

Outline

Outline

STL pairs and tuples

pair of elements (`#include <utility>`)

```
pair<int, int> P(3,-5);
```

STL pairs and tuples

pair of elements (`#include <utility>`)

```
pair<int, int> P(3,-5);
```

tuple of elements (`#include <tuple>`)

```
tuple<pair<int, int>, double, double> T(P, 1.0, 3.0);
```

STL pairs and tuples

pair of elements (`#include <utility>`)

```
pair<int, int> P(3,-5);
```

tuple of elements (`#include <tuple>`)

```
tuple<pair<int, int>, double, double> T(P, 1.0, 3.0);
```

Notice *we can nest templates*

STL array

```
#include <array>
```

not *quite* same as "native" array

```
int NA[25];           // native array
array<int, 25> SA; // STL array
NA[0];
SA[0];
```

STL array

```
#include <array>
```

not *quite* same as "native" array

```
int NA[25];          // native array
array<int, 25> SA; // STL array
NA[0];
SA[0];
```

It gets better! STL array:

- knows own size

```
SA.size(); // should return 25
```

- easily filled

```
SA.fill(3); // fills SA w/3
```

Major b/w native, STL array

Must know size at compile time.

STL vector

`#include <vector>`

"resizable" array

- resizing preserves old entries when possible
- "inserting" not advisable

```
vector<int> V(25, 3); // "vector" V holds 25 3's
V[0];                 // should return 3
V.resize(35, 2);      // expands V to hold 35 els
V[0];                 // still (!) 3
V[30];                // 2 (new els set to 2)
V[30] = -2;           // sets V[30] to -2
```

STL vector

`#include <vector>`

"resizable" array

- resizing preserves old entries when possible
- "inserting" not advisable

```
vector<int> V(25, 3); // "vector" V holds 25 3's
V[0];                 // should return 3
V.resize(35, 2);      // expands V to hold 35 els
V[0];                 // still (!) 3
V[30];                // 2 (new els set to 2)
V[30] = -2;           // sets V[30] to -2
```

vector "size" v. "capacity"

- **size**: number of elements *actually assigned*
- **capacity**: number of spots for elements
- no automatic resize on bracket access!
  - need to resize when size, capacity equal
  - reading or writing beyond end causes trouble

STL list

`#include <list>`

think of a pile of notecards

- size hard to determine
- easy to access front, back
- cannot immediately access anything else

```
list<int> L;        // ()
L.push_front(3); // (3)
L.push_front(4); // (4,3)
L.push_back(2);  // (4,3,2)
L.front();          // 4
L.back();           // 2
```

STL list

`#include <list>`

think of a pile of notecards

- size hard to determine
- easy to access front, back
- cannot immediately access anything else

```
list<int> L;        // ()
L.push_front(3); // (3)
L.push_front(4); // (4,3)
L.push_back(2);  // (4,3,2)
L.front();          // 4
L.back();           // 2
```

Easy to insert or remove, too! Requires iteration; see later.

Pushing, emplacing

Pushing

- Adds element to front or back
- lists and vectors: `push_back()`
  - vector automatically resized if necessary
- lists only: `push_front()`

Pushing, emplacing

Pushing

- Adds element to front or back
- lists and vectors: `push_back()`
  - vector automatically resized if necessary
- lists only: `push_front()`

Emplacing

- good for objects of a *class*
- constructs a new object and pushes to front or back
- lists and vectors: `emplace_back()`
  - vector automatically resized if necessary
- lists only: `emplace_front()`

Outline

STL set

`#include <set>`

much like a mathematical set; for things that can be ordered

- easy *and fast* to search
- can pass through "in order"
- easy and *relatively fast* to insert, remove
- cannot speak of "front," "back," or "*i*th" element

```
set<int> S;     // {}
S.insert(4);    // { 4 }
S.insert(3);    // { 3, 4 }
S.erase(3);     // { 4 }
S.count(3);     // returns 0
S.count(4);     // returns 1
```

STL set

`#include <set>`

much like a mathematical set; for things that can be ordered

- easy *and fast* to search
- can pass through "in order"
- easy and *relatively fast* to insert, remove
- cannot speak of "front," "back," or "*i*th" element

```
set<int> S;     // {}
S.insert(4);    // { 4 }
S.insert(3);    // { 3, 4 }
S.erase(3);     // { 4 }
S.count(3);     // returns 0
S.count(4);     // returns 1
```

Can retrieve specific elements. Requires iteration; see later.

STL unordered set

`#include <set>` — *not a typo*

much like a mathematical set; for things that can be "binned"

- American sense of "binned," not UK's
- easy *and fast* to search
- easy *and relatively fast* to insert, remove
- cannot pass through in any sort of order
- cannot speak of "front," "back," or "*i*th" element

```
unordered_set<int> US; // { }
US.insert(4);          // { 4 }
US.insert(3);          // { 4, 3 }
US.erase(4);           // { 3 }
US.count(3);           // returns 1
US.count(4);           // returns 0
```

Outline

STL map

`#include <map>`

relates *key* and *value*, much like a dictionary

- subset of $K \times V$, where keys in K, values in V
- keys must have a type that can be ordered
- bracket access via key to value; i.e., `M[key]` $\rightarrow$ `value`
- can be used to remember previously computed values (**cache**)

```
map<int, int> Fibonacci;
Fibonacci[1] = 1;
Fibonacci[2] = 1;
for (unsigned i = 3; i < 10; ++i)
  Fibonacci[i] = Fibonacci[i-1]
                 + Fibonacci[i-2];
Fibonacci[9]; // returns 34
```

STL map

`#include <map>`

relates *key* and *value*, much like a dictionary

- subset of $K \times V$, where keys in K, values in V
- keys must have a type that can be ordered
- bracket access via key to value; i.e., `M[key]` $\rightarrow$ `value`
- can be used to remember previously computed values (**cache**)

```
map<int, int> Fibonacci;
Fibonacci[1] = 1;
Fibonacci[2] = 1;
for (unsigned i = 3; i < 10; ++i)
  Fibonacci[i] = Fibonacci[i-1]
                 + Fibonacci[i-2];
Fibonacci[9]; // returns 34
```

Looks just like an array or vector but no concerns about size.

STL unordered map

`#include <map>` – *not a typo*

relates *key* and *value*, much like a dictionary

- subset of $K \times V$, where keys in K, values in V
- keys must have a type that can be "binned," see above
- bracket access via key to value; i.e., `M[key]` $\rightarrow$ `value`
- can be used to remember previously computed values (**cache**)

Outline

Passing arguments

Remember that passing an argument:

by value argument *copies* the data

by reference argument *is* the data

Passing arguments

Remember that passing an argument:

by value argument *copies* the data

by reference argument *is* the data

Question, then:

If we pass container by value, does it copy every element?

Passing arguments

Remember that passing an argument:

by value argument *copies* the data

by reference argument *is* the data

Question, then:

If we pass container by value, does it copy every element?

Answer:

Logically, yes. To modify container data, pass by reference.

Proof by example

```
#include <vector>
using std::vector;

#include <iostream>
using std::cout; using std::endl;

void change_vector_val(vector<int> V) { V[0] = 1; }

void change_vector_ref(vector<int> & V) { V[0] = 1; }

int main() {
  vector V(1, 0);          // vector w/1 zero
  cout << V[0] << endl;
  change_vector_val(V); // call by value
  cout << V[0] << endl;
  change_vector_ref(V); // call by reference
  cout << V[0] << endl;
}
```

Compile, execute

```
$ g++ -o container_by_value \
  container_by_value.cpp
$ ./container_by_value
0
0
1
```

Homework

pp. 150–155 #8.1, 8.7 (use a map), 8.12

Outline

Outline

Idea of iteration

Pass over elements of set in convenient fashion

- "from beginning to end" rather than "from `[0]` to `[size-1]`"
- works w/all containers, even unordered
  - can't speak of 3rd element of set, after all

Two ways of iteration

- use of iterator
  - "points" to element in container
  - forward iterators move right (`++i`)
  - reverse iterators move left (`--i`)
  - most STL iterators can do both

Two ways of iteration

- use of iterator
  - "points" to element in container
  - forward iterators move right (`++i`)
  - reverse iterators move left (`--i`)
  - most STL iterators can do both
- "range-based for loop" (since C++11)
  - `for ( Type c : C ) { /* do stuff w/c */ }`
  - each pass of loop: `c` a different element in `C`
  - cannot ask to move left or right
  - works only when `C`:
    - knows its size (STL arrays yes, "native" arrays no)
    - has forward iterator (can move right)

Outline

Example: prime sieve, revisited

For sieve and list of primes, use *vectors* instead of *arrays*

- no need to allocate w/`new`, deallocate w/`delete []`
- iterate over primes easily and conveniently
- also switch to `uint64_t` instead of `long`

Interface

Listing 4: `sieve_vectorized.hpp`

```
#ifndef __SIEVE_H_
#define __SIEVE_H_

#include <cstdint>

/**
Sieve of Eratosthenes: generate table of primes.
@param n find primes <= n
@param primes array of primes
@return number of primes found
*/

uint64_t sieve(uint64_t n, vector<uint64_t> & primes);

#endif
```

Implementation

Listing 5: `sieve_vectorized.cpp` (1/3)

```
#include <cmath>
using std::sqrt;

#include <vector>
using std::vector;

#include "sieve.hpp"

uint64_t sieve(uint64_t n, vector<uint64_t> & primes)
{
  uint64_t m = uint64_t(sqrt(n));
  // resize primes later, rather than now

  uint64_t num_primes = 0;
  vector<bool> theSieve(n, true);
```

First implementation

Listing 6: `sieve_vectorized.cpp` (2/3)

```
uint64_t i;
for (i = 2; i < m + 1; ++i) {
  if (theSieve[i] == true) {
    ++num_primes;
    uint64_t a = 2*i;
    while (a < n) {
      theSieve[a] = false;
      a += i;
    }
  }
}
for (/* */; i < n; ++i) {
  if (theSieve[i] == true)
    ++num_primes;
}
```

First implementation

Listing 7: `sieve_vectorized.cpp` (3/3)

```
  // resize vector, copy all primes in
  primes.resize(num_primes);
  uint64_t j = 0;
  for (i = 2 ; i < n ; ++i) {
    if (theSieve[i] == true) {
      primes[j] = i;
      ++j;
    }
  }
  return num_primes;
}
```

Test program

Listing 8: `test_sieve_vectorized.cpp` (1/2)

```
#include <iostream>
using std::cin; using std::cout;
using std::endl;

#include <vector>
using std::vector;

#include "sieve_vectorized.hpp"

int main() {
vector<uint64_t> primes;
uint64_t n;

cout << "This program finds all primes ";
cout << "less than your choice of number.\n";
cout << "Please choose a number --> ";
cin >> n;
```

Test program

Listing 9: `test_sieve_vectorized.cpp` (2/2)

```
uint64_t np = sieve(n, primes);
cout << "There are " << np << " primes:\n";
for (uint64_t i : primes)
cout << i << ", ";
cout << endl;
}
```

Compile, execute

Compilation note

`clang` on my computer requires `-std=c++11`

```
$ g++ -c sieve_vectorized.cpp
$ g++ -std=c++11 -o test_sieve_vectorized \
  sieve_vectorized.o test_sieve_vectorized.cpp
$ ./test_sieve_vectorized
This program finds all primes less than your choice of
number.
Please choose a number --> 1000
There are 168 primes:
2, 3, 5, 7, 11, 13, 17, 19, ... 977, 983, 991, 997,
```

Drawback to this approach

Run through list twice

- first time: `for` loops in lines 17, 27
- second time: `for` loop in line 34

Drawback to this approach

Run through list twice

- first time: `for` loops in lines 17, 27
- second time: `for` loop in line 34

Not too bad, but:

- Don't want to waste time in computation, *and...*
- doubling time not as bad as exponentiating it, *but...*
- perhaps we can do better?

Store during first two loops

Store prime whenever we discover it

- Pros: pass through list only once
- Cons: code more complicated (internally)
  - don't know number of primes beforehand
  - `push_back()` needs to resize

Second implementation

Listing 10: `sieve_vectorized_resizing.cpp` (1/2)

```
#include <cmath>
using std::sqrt;

#include <vector>
using std::vector;

#include "sieve.hpp"

uint64_t sieve(uint64_t n, vector<uint64_t> & primes) {
  uint64_t m = uint64_t(sqrt(n));
  // resize primes later, rather than now

  uint64_t num_primes = 0;
  vector<bool> theSieve(n, true);

  uint64_t i;
```

Second implementation

Listing 11: `sieve_vectorized_resizing.cpp` (2/2)

```
  for (i = 2; i < m + 1; ++i) {
    if (theSieve[i] == true) {
      primes.push_back(i);
      ++num_primes;
      uint64_t a = 2*i;
      while (a < n) {
        theSieve[a] = false;
        a += i;
      }
    }
  }
  for (/* */; i < n; ++i) {
    if (theSieve[i] == true) {
      primes.push_back(i);
      ++num_primes;
    }
  }
  return num_primes;
}
```

Compile, execute

Compilation note

`clang` on my computer requires `-std=c++11`

```
$ g++ -c sieve_vectorized_resizing.cpp
$ g++ -std=c++11 -o test_sieve_vectorized_resizing \
  sieve_vectorized_resizing.o test_sieve_vectorized.cpp
$ ./test_sieve_vectorized
This program finds all primes less than your choice of
number.
Please choose a number --> 1000
There are 168 primes:
2, 3, 5, 7, 11, 13, 17, 19, ... 977, 983, 991, 997,
```

Timings

Second version (resizing) takes ~25% less time: why?

- automatic resizing not a big penalty?
- less to copy when resizing v. passing through the loop twice?

$$\pi(n) \sim n/\ln(n) \ll n$$

Outline

Obtaining iterators

- `begin()`, `end()`: iterators at beginning, end
- `rbegin()`, `rend()`: iterators at "reverse beginning", "reverse end"
- search commands return `iterator`
  - sometimes return `const_iterator`; see below

Obtaining iterators

- `begin()`, `end()`: iterators at beginning, end
- `rbegin()`, `rend()`: iterators at "reverse beginning", "reverse end"
- search commands return `iterator`
  - sometimes return `const_iterator`; see below
- type templated *and scoped*
  - `vector<int>::iterator vi = V.begin();`
  - `vector<int>::const_iterator vi = V.begin();`

Obtaining iterators

- `begin()`, `end()`: iterators at beginning, end
- `rbegin()`, `rend()`: iterators at "reverse beginning", "reverse end"
- search commands return `iterator`
  - sometimes return `const_iterator`; see below
- type templated *and scoped*
  - `vector<int>::iterator vi = V.begin();`
  - `vector<int>::const_iterator vi = V.begin();`
- cons
  - annoying to type
  - detracts from readability
  - makes it harder to change container if necessary
- improve w/`auto`
  - `auto vi = V.begin();`

Using iterators

C-legacy operators: `list<unsigned>::iterator Ci`

- forward increment (++), backward decrement (– –)
  ```
  ++Ci; // move forward
  ```
- access data by **dereference** (*)
  ```
  unsigned this_entry = *Ci;
  ```
- access methods/fields by **indexing** (->)
  ```
  Ci->... // better example below
  ```

Iterator w/`const` argument

Sometimes we might want to pass a vector and promise not to change anything.

```
void do_something_with(
     const vector<int> & V
);
```

Iterator w/`const` argument

Sometimes we might want to pass a vector and promise not to change anything.

```
void do_something_with(
    const vector<int> & V
);
```

- want to iterate over `V`'s elements, *but...*
- `iterator` allows element modification, *and...*
- `begin(),end()` give us `iterator`

Iterator w/`const` argument

Sometimes we might want to pass a vector and promise not to change anything.

```
void do_something_with(
    const vector<int> & V
);
```

- want to iterate over `V`'s elements, *but...*
- `iterator` allows element modification, *and...*
- `begin(),end()` give us `iterator`

Violates "`const`-correctness"!

Solution: `const_iterator`

C++ `const_iterator`!

- does not allow element modification
- obtain w/`cbegin()`, `cend()`
  - new with C++11
- automatically provided w/`auto`

Example

```
#include <vector>
using std::vector;

#include <iostream>
using std::cout; using std::endl;

int sum_of(const vector<int> & V) {
  int result = 0;
  for ( // usually avoid the following mess w/auto
       vector<int>::const_iterator vi = V.cbegin();
       vi != V.cend();
       ++vi
  ) {
    result += *vi;
  }
  return result;
}

int main() {
  vector<int> V = { -3, 1, 0, 2 };
  cout << sum_of(V) << endl;
}
```

Outline

Example: sparse matrix

Ordinarily, matrix is two-dimensional array:

- nested arrays/vectors
- outer array viewed as “rows”
- inner array viewed as “columns”

Example: sparse matrix

Ordinarily, matrix is two-dimensional array:

- nested arrays/vectors
- outer array viewed as “rows”
- inner array viewed as “columns”

In a sparse matrix, most entries are *zero*, so full matrix:

- wastes space
- many algorithms: wastes time

Example: matrix multiplication

$$M = \begin{pmatrix} 1 & 1 & & & \\ & 1 & 1 & & \\ & & 1 & 1 & \\ & & & 1 & 1 \\ & & & & 1 \end{pmatrix} \qquad N = \begin{pmatrix} 1 & 2 & & & \\ & 2 & 3 & & \\ & & 3 & 3 & \\ & & & 2 & 2 \\ & & & & 1 \end{pmatrix}$$

- naive multiplication: 5^3 multiplications
  - most by 0, complete waste of time!
- only 20 non-zero multiplications: can we approach this?

Sparse representation

Many ways to represent sparse matrix

- "simple" way: list of (col, val) entries

Sparse representation

Many ways to represent sparse matrix

- "simple" way: list of (col, val) entries

$$M = \begin{pmatrix} 1 & 1 & & & \\ & 1 & 1 & & \\ & & 1 & 1 & \\ & & & 1 & 1 \\ & & & & 1 \end{pmatrix} \longrightarrow \begin{pmatrix} (1,1),(2,1) \\ (2,1),(3,1) \\ (3,1),(4,1) \\ (4,1),(5,1) \\ (5,1) \end{pmatrix}$$

Sparse representation

Many ways to represent sparse matrix

- "simple" way: list of (col, val) entries

$$M = \begin{pmatrix} 1 & 1 & & & \\ & 1 & 1 & & \\ & & 1 & 1 & \\ & & & 1 & 1 \\ & & & & 1 \end{pmatrix} \longrightarrow \begin{pmatrix} (1,1),(2,1) \\ (2,1),(3,1) \\ (3,1),(4,1) \\ (4,1),(5,1) \\ (5,1) \end{pmatrix}$$

- entries: $25 \longrightarrow 9 \times 2$, 28% less space

Data structure?

M, N as vectors of maps

- each map represents a row

```
vector< map<unsigned, int> > M(50);
M[0][1] = 1;
M[0][2] = 1;
M[1][2] = 1;
M[1][3] = 1;
```

...and so forth

Implementation (multiply only)

```
void multiply_sparse(
    const vector< map<unsigned, int> > & M,
    const vector< map<unsigned, int> > & N,
    unsigned N_cols,
    vector< map<unsigned, int> > & result
) {
  result.resize(M.size());
  for (unsigned row = 0; row < M.size(); ++row) {
    if (M[row].size() > 0) {
      for (unsigned i = 0; i < N_cols; ++i) {
        int dot_prod = 0;
        for (
            auto el = M[row].begin();
            el != M[row].end();
            ++el
        ) {
          if (N[el->first].count(i) != 0)
            dot_prod += el->second * N[el->first].at(i);
        }
        if (dot_prod != 0)
          result[row].emplace(i, dot_prod);
      }
    }
  }
}
```

Compile, execute, compare

Very sparse 50×50 matrix

```
$ g++ -std=c++11 -o sparse_matrix_demo \
                    sparse_matrix_demo.cpp
$ g++ -std=c++11 -o dense_matrix_demo \
                    dense_matrix_demo.cpp
$ time ./sparse_matrix_demo
...
real 0m4.922s
user 0m4.892s
sys 0m0.019s
$ time ./dense_matrix_demo
...
real 0m17.554s
user 0m17.400s
sys 0m0.061s
```

Compile, execute, compare

Very sparse 50×50 matrix

```
$ g++ -std=c++11 -o sparse_matrix_demo \
                    sparse_matrix_demo.cpp
$ g++ -std=c++11 -o dense_matrix_demo \
                    dense_matrix_demo.cpp
$ time ./sparse_matrix_demo
...
real 0m4.922s
user 0m4.892s
sys 0m0.019s
$ time ./dense_matrix_demo
...
real 0m17.554s
user 0m17.400s
sys 0m0.061s
```

Dense version *3–4 times slower*

Caution: don't try this at home

Pretty poor sparse design, actually

- not so impressive on small matrices!
- sparse matrix computation an entire research field unto itself

Homework

pp. 151–155 #8.8, 8.9

Outline

Outline

Initialization?

Suppose

- we want a container
- we know its initial values

Must we rely on iterators to fill the initial values?

Initialization?

Suppose

- we want a container
- we know its initial values

Must we rely on iterators to fill the initial values?
No: use an initializer list instead!

Initialization?

Suppose

- we want a container
- we know its initial values

Must we rely on iterators to fill the initial values?
No: use an initializer list instead!

Note

This is a C++11 feature. You *must* compile with `-std=c++11`.

How to initialize?

When *constructing:*

- *assign* to new element
- enclose sequence of values between braces

How to initialize?

When *constructing:*

- *assign* to new element
- enclose sequence of values between braces

Example

```
#include <set>
using std::set;
#include <iostream>
using std::cout; using std::endl;

int main() {
  set<int> S = { 5, 7, 1, -8, 2, 4 };
  for (auto s : S)
    cout << s << ',';
  cout << endl;
}
```

Compile, execute

```
$ g++ -std=c++11 -o initializer_demo \
                    initializer_demo.cpp
$ ./initializer_demo
-8,1,2,4,5,7,
```

Compile, execute

```
$ g++ -std=c++11 -o initializer_demo \
                    initializer_demo.cpp
$ ./initializer_demo
-8,1,2,4,5,7,
```

Warning

Don't forget `-std=c++11`. W/`clang` you get this error:

```
initializer_demo.cpp:8:12: error:
non-aggregate type 'set<int>' cannot
be initialized with an initializer list
  set<int> S = { 5, 7, 1, -8, 2, 4 };
           ^   ~~~~~~~~~~~~~~~~~~~~~
```

Outline

How to enable

To enable an initialization list for a class:

- include `initializer_list` library
- add constructor that accepts `initializer_list` argument
- iterate over elements in `initializer_list`
  - *either* a range-based for loop,
  - *or* an iterator

Example: univariate polynomials

Univariate polynomials have form

$$a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$$

Example: univariate polynomials

Univariate polynomials have form

$$a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$$

Natural candidate for initializer list!

- elements a sequence of coefficients
- tedious to initialize one coefficient at a time
- annoying to create a list/vector/whatever, then initialize from that

Interface for minimal class

Listing 12: `univariate_polynomial.hpp` (1/3)

```
#include <vector>
using std::vector;

#include <iostream>
using std::ostream;

#include <initializer_list>
using std::initializer_list;

template <typename Ring>
class Univariate_Polynomial {
protected:
  vector<Ring> coeffs;
public:
  /** start list from constant, end at leading coeff */
  Univariate_Polynomial<Ring>(initializer_list<Ring> L);
  /** returns degree of polynomial */
  unsigned degree() const;
  /** returns coefficient of given degree */
  Ring coeff(unsigned) const;
};
```

Interface for minimal class

Listing 13: `univariate_polynomial.hpp` (2/3)

```
template <typename Ring>
Univariate_Polynomial<Ring>::Univariate_Polynomial(
    initializer_list<Ring> L
) {
  coeffs.resize(L.size());
  unsigned i = 0;
  for (auto li = L.begin(); li != L.end(); ++li) {
    coeffs[i] = *li;
    ++i;
  }
}

template <typename Ring>
unsigned Univariate_Polynomial<Ring>::degree() const {
  return coeffs.size();
}

template <typename Ring>
Ring Univariate_Polynomial<Ring>::coeff(unsigned d) const {
  return coeffs[d];
}
```

Listing 14: `univariate_polynomial.cpp` (3/3)

```
template <typename Ring> ostream & operator << (
    ostream & os,
    const Univariate_Polynomial<Ring> & p
) {
  bool first_printed = true;
  for (unsigned d = 0; d < p.degree(); ++d) {
    Ring a = p.coeff(d);
    if (a != 0) {
      if (d == 0) os << a;
      else {
        // first print sign
        if (a < 0) {
          if (not first_printed) os << '-';
          else os << " - ";
          os << -a << '*';
        } else {
          if (first_printed) os << " + ";
          if (a != 1) os << a << '*';
        }
        os << "x^" << d;
      }
      first_printed = true;
    }
  }
  return os;
}
```

Test program

```
#include <iostream>
using std::cout; using std::endl;

#include "upoly_initializers.hpp"

int main() {
  Univariate_Polynomial<int> p
      = { 2, 0, 0, -3, 1, 0, 5 };
  cout << p << endl;
}
```

Compile, execute

```
$ g++ -std=c++11 -o test_init \
            test_upoly_initializers.cpp
$ ./test_init
2 - 3*x^3 + x^4 + 5*x^6
```

Outline

Summary

- Math stuff
  - sparse matrices
- Programming stuff
  - containers
  - iterators
  - constant iterators
  - `auto`
  - `initializer_list`