The Standard Template Library

- *Standard* : Comes with all C++ implementations
  - May not be available in tiniest systems (e.g. Arduino)

- *Template* : heavily dependent on templates
  - Containers that can contain any type
  - Algorithms that can operate on any type

- *Library* : a piece of re-useable code
  - You include a header for the declarations
  - Some definitions will get linked in as separate source

Friends from the STL

We've been using some things extensively:

```
vector<T> :
string
cin, cout
```

Others we've seen occasionally

```
sort
min, max
list
```

pair<A,B> : a new friend

```
template<typename T1, typename T2>
struct pair
{
    T1 first;
    T2 second;
};
```

- pair represents the idea of a pair of values
- you can use pretty much any types for T1 and T2

pair<A,B> : a new friend

```
template<typename T1, typename T2>
struct pair
{
    T1 first;
    T2 second;
};

int main()
{
    pair<int,int> ab;
}
```

- pair represents the idea of a pair of values
- you can use pretty much any types for T1 and T2

pair<A,B> : a new friend

```
template<typename T1, typename T2>
struct pair
{
    int first;
    int second;
};

int main()
{
    pair<int,int> ab;
}
```

- pair represents the idea of a pair of values
- you can use pretty much any types for T1 and T2

pair<A,B> : a new friend

```
template<typename T1, typename T2>
struct pair
{
    T1 first;
    T2 second;
};

int main()
{
    pair<int,int> ab;
    pair<string,float> cd;
}
```

- pair represents the idea of a pair of values
- you can use pretty much any types for T1 and T2
- you can have many types of pairs in one program

pair<A,B> : a new friend

```
template<typename T1, typename T2>
struct pair
{
    T1 first;
    T2 second;
};

int main()
{
    pair<string,float> cd{"blah",4.5};
}
```

- can construct just like a normal struct

pair<A,B> : a new friend

```
template<typename T1, typename T2>
struct pair
{
    T1 first;
    T2 second;
};

int main()
{
    pair<string,float> cd{"blah",4.5};
    cd.first = "blurb";
    cd.second += 2.3;
}
```

- can construct just like a normal struct
- access member variables just like a normal struct

Some things in the STL are simple

- People often need to create pairs
  - It often isn't worth creating a new type
  - We get some free stuff with pair: e.g. comparison operators

- It is sometimes useful for designing APIs
  - Using `pair<T1,T2>` we know the type is not that important
  - Often used for returning two values from a function

- Not everything in the STL is complicated
  - `min, max, swap, identity`
  - It avoids repetition, and aids understanding

`vector<T>` : an old friend

We know and understand vector well

- How to use it
- How to implement it
- Some of the costs associated with it

In terms of functionality we could say:

vector<T> : maps indices in [0,n)
to values of type T

We can read and write the value at any index

We can change the size n

vector<T>: strengths + weaknesses

Strengths:

- *Speed* : access to any index is extremely fast
- *Efficiency*: we only store the values; indexes cost nothing

Weaknesses:

- *Contiguous indices*: must allocate values for [0,n-1) to store at n
- *Fixed index type*: only natural numbers can be used as indices

The STL provides a richer set of container types

list<T> : linked list

map<K,V> : mapping or dictionary from keys to values

set<K> : finite set of values

Motivating example: histograms

Our requirements:

1. Read a stream of values from `cin`
2. Track the number of times each value appears
3. Print a histogram of values to `cout`

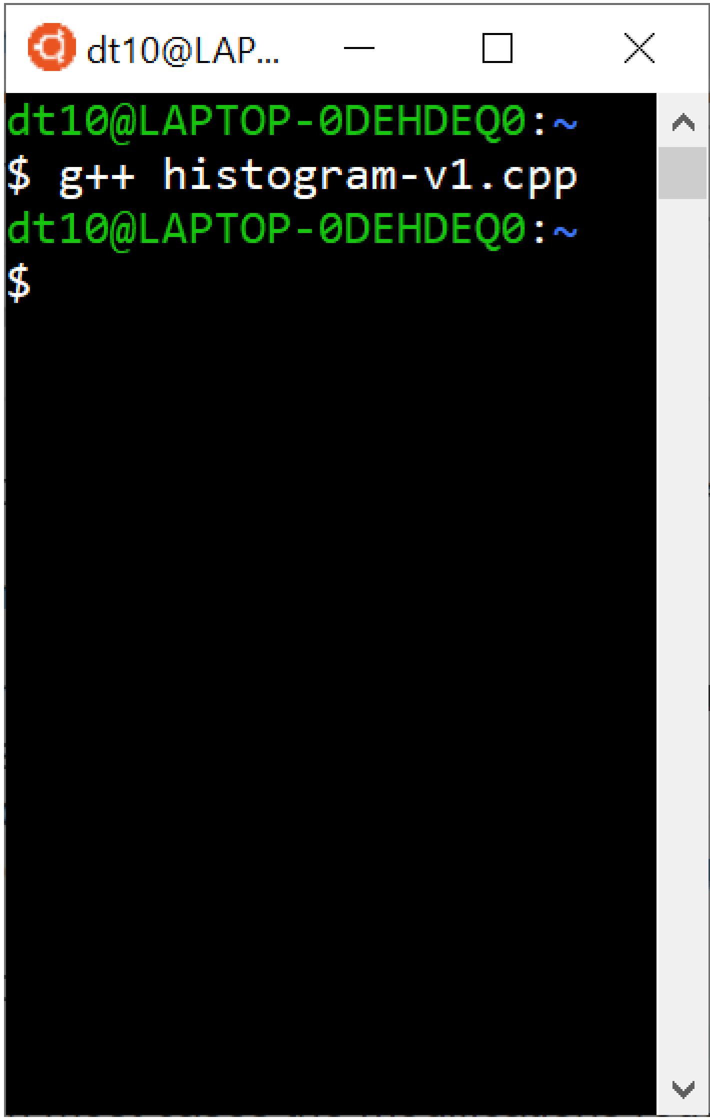
A basic operation that appears in lots of data-science and statistical work-flows

Motivating example: histograms

```
int main()
{
  // Count of the number of times each input is seen
  vector<int> histogram;

  int x;
  while( cin >> x ) {
    // Extend histogram range if necessary
    if( x >= histogram.size() ){
        histogram.resize( x+1 );
    }
    // Increment the location in the histogram
    histogram[x] += 1;
  }

  // Print the histogram counts out
  for(int i=0;i<histogram.size();i++){
    cout<<i<<","<<histogram[i]<<endl;
  }
}
```

Motivating example: histograms

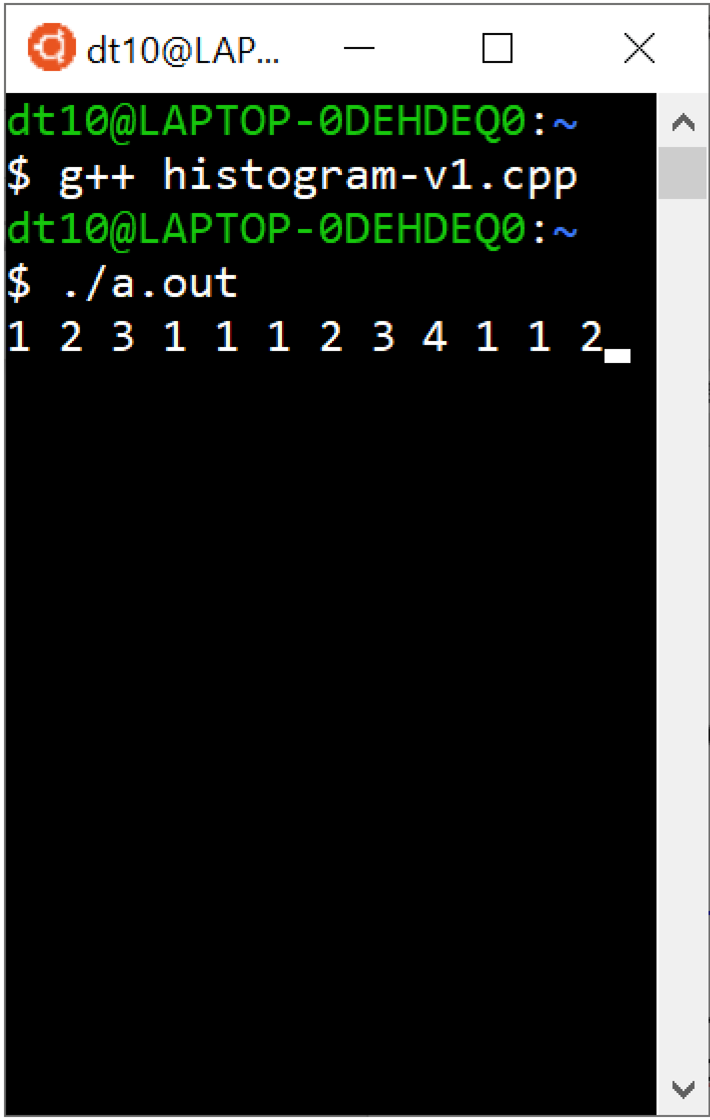
```
int main()
{

  vector<int> histogram;

  int x;
  while( cin >> x) {

    if( x >= histogram.size() ){
        histogram.resize( x+1 );
    }

    histogram[x] += 1;
  }


  for(int i=0;i<histogram.size();i++){
    cout<<i<<","<<histogram[i]<<endl;
  }
}
```



```
dt10@LAPTOP-0DEHDEQ0:~
$ g++ histogram-v1.cpp
dt10@LAPTOP-0DEHDEQ0:~
$
```

Motivating example: histograms

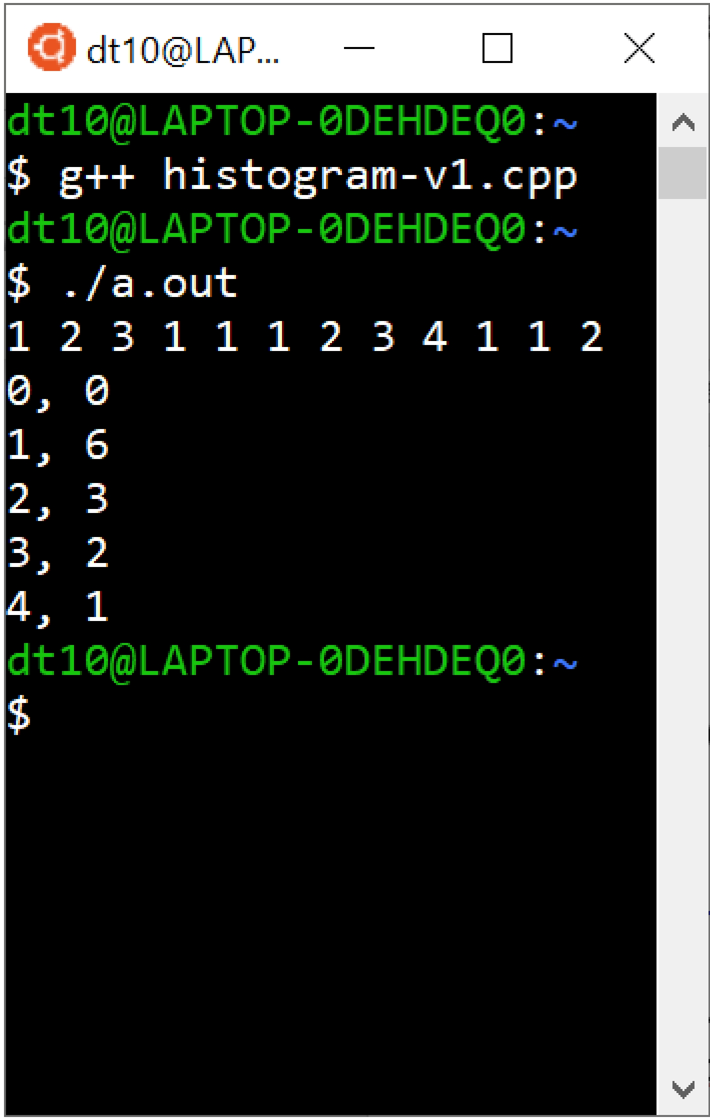
```
int main()
{

  vector<int> histogram;

  int x;
  while( cin >> x) {

    if( x >= histogram.size() ){
        histogram.resize( x+1 );
    }

    histogram[x] += 1;
  }


  for(int i=0;i<histogram.size();i++){
    cout<<i<<","<<histogram[i]<<endl;
  }
}
```



```
dt10@LAPTOP-0DEHDEQ0:~
$ g++ histogram-v1.cpp
dt10@LAPTOP-0DEHDEQ0:~
$ ./a.out
1 2 3 1 1 1 2 3 4 1 1 2
```

Motivating example: histograms

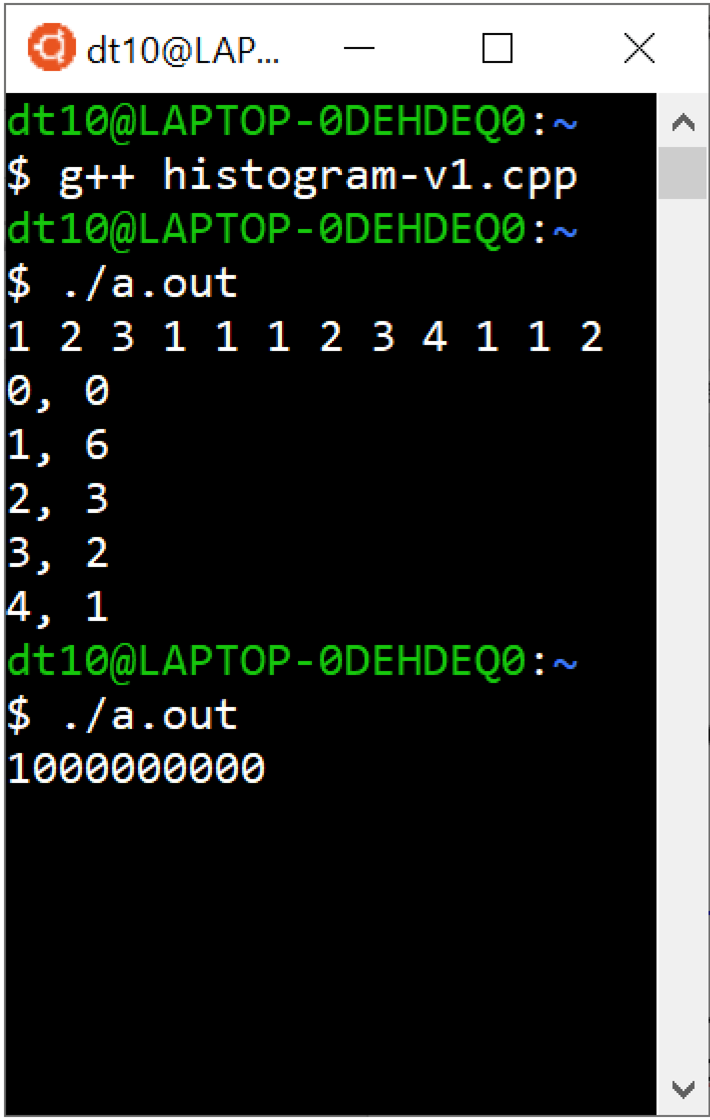
```
int main()
{

  vector<int> histogram;

  int x;
  while( cin >> x) {

    if( x >= histogram.size() ){
        histogram.resize( x+1 );
    }

    histogram[x] += 1;
  }


  for(int i=0;i<histogram.size();i++){
    cout<<i<<","<<histogram[i]<<endl;
  }
}
```



```
dt10@LAPTOP-0DEHDEQ0:~
$ g++ histogram-v1.cpp
dt10@LAPTOP-0DEHDEQ0:~
$ ./a.out
1 2 3 1 1 1 2 3 4 1 1 2
0, 0
1, 6
2, 3
3, 2
4, 1
dt10@LAPTOP-0DEHDEQ0:~
$
```

Motivating example: histograms

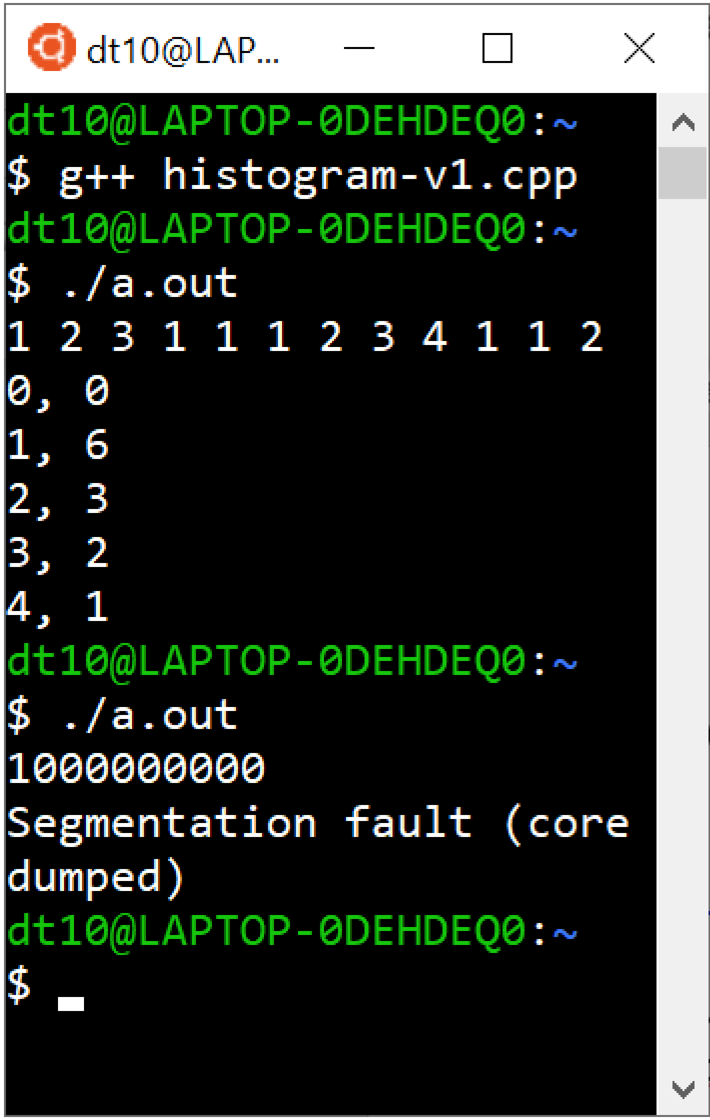
```
int main()
{

  vector<int> histogram;

  int x;
  while( cin >> x) {

    if( x >= histogram.size() ){
        histogram.resize( x+1 );
    }

    histogram[x] += 1;
  }


  for(int i=0;i<histogram.size();i++){
    cout<<i<<","<<histogram[i]<<endl;
  }
}
```



```
dt10@LAPTOP-0DEHDEQ0:~
$ g++ histogram-v1.cpp
dt10@LAPTOP-0DEHDEQ0:~
$ ./a.out
1 2 3 1 1 1 2 3 4 1 1 2
0, 0
1, 6
2, 3
3, 2
4, 1
dt10@LAPTOP-0DEHDEQ0:~
$ ./a.out
1000000000
```

Motivating example: histograms

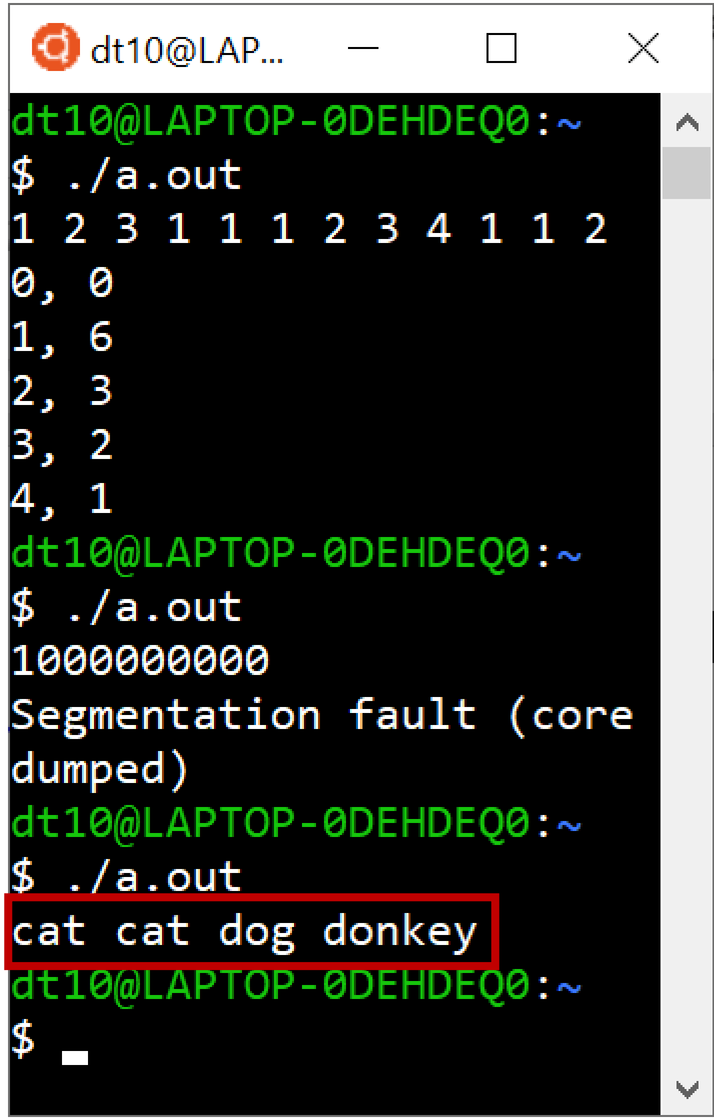
```
int main()
{

  vector<int> histogram;

  int x;
  while( cin >> x) {

    if( x >= histogram.size() ){
        histogram.resize( x+1 );
    }

    histogram[x] += 1;
  }


  for(int i=0;i<histogram.size();i++){
    cout<<i<<","<<histogram[i]<<endl;
  }
}
```



```
dt10@LAPTOP-0DEHDEQ0:~
$ g++ histogram-v1.cpp
dt10@LAPTOP-0DEHDEQ0:~
$ ./a.out
1 2 3 1 1 1 2 3 4 1 1 2
0, 0
1, 6
2, 3
3, 2
4, 1
dt10@LAPTOP-0DEHDEQ0:~
$ ./a.out
1000000000
Segmentation fault (core
dumped)
dt10@LAPTOP-0DEHDEQ0:~
$ 
```

Motivating example: histograms

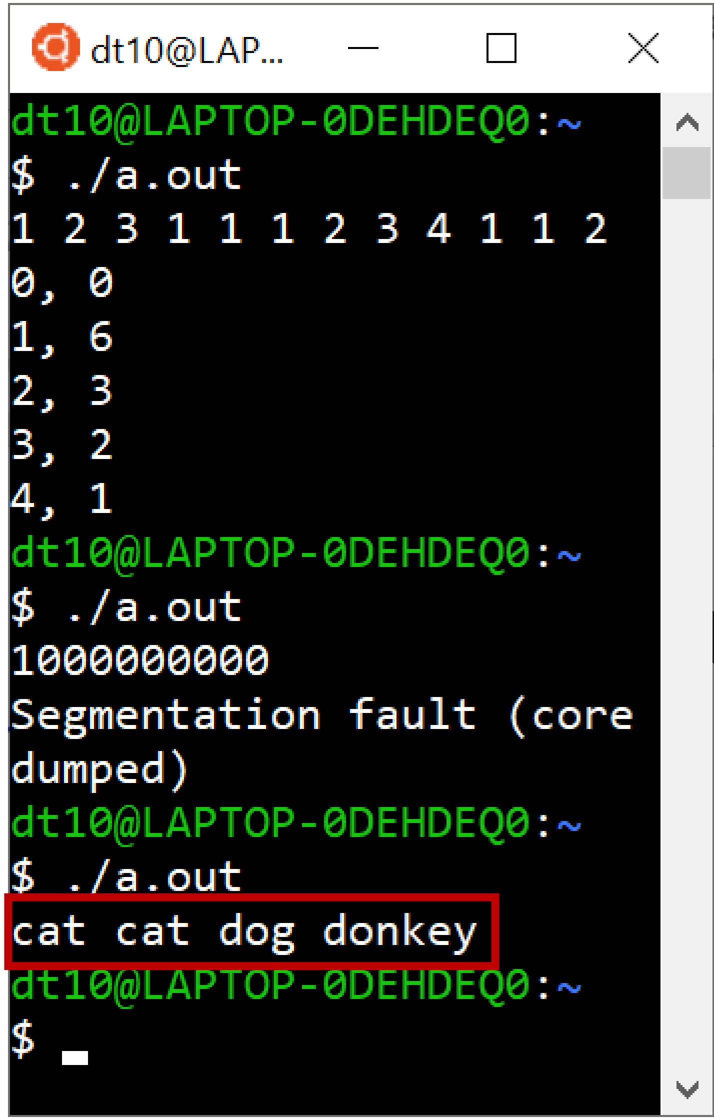
```
int main()
{

  vector<int> histogram;

  int x;
  while( cin >> x) {

    if( value >= histogram.size() ){
        histogram.resize( x+1 );
    }

    histogram[x] += 1;
  }


  for(int i=0;i<histogram.size();i++){
    cout<<i<<","<<histogram[i]<<endl;
  }
}
```



```
dt10@LAPTOP-0DEHDEQ0:~
$ ./a.out
1 2 3 1 1 1 2 3 4 1 1 2
0, 0
1, 6
2, 3
3, 2
4, 1
dt10@LAPTOP-0DEHDEQ0:~
$ ./a.out
1000000000
Segmentation fault (core
dumped)
dt10@LAPTOP-0DEHDEQ0:~
$ ./a.out
cat cat dog donkey
dt10@LAPTOP-0DEHDEQ0:~
$
```

Motivating example: histograms

```
int main()
{

  vector<int> histogram;

  string x;
  while( cin >> x) {

    if( x >= histogram.size() ){
        histogram.resize( x+1 );
    }

    histogram[x] += 1;
  }


  for(int i=0;i<histogram.size();i++){
    cout<<i<<","<<histogram[i]<<endl;
  }
}
```



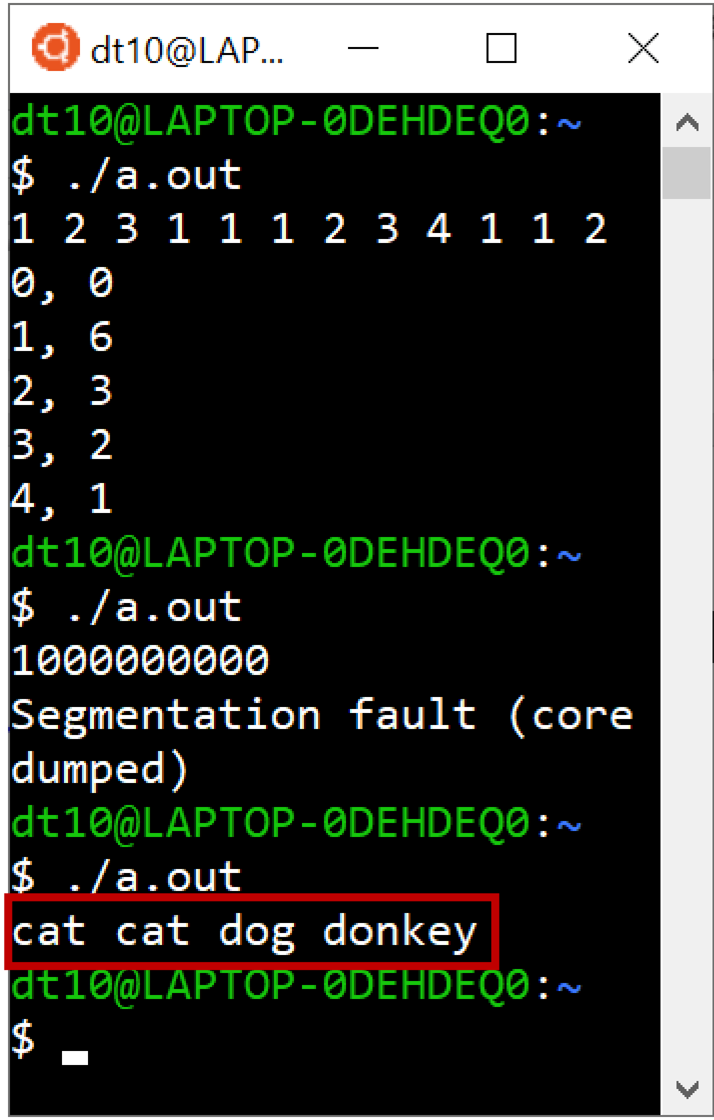
```
dt10@LAPTOP-0DEHDEQ0:~
$ ./a.out
1 2 3 1 1 1 2 3 4 1 1 2
0, 0
1, 6
2, 3
3, 2
4, 1
dt10@LAPTOP-0DEHDEQ0:~
$ ./a.out
1000000000
Segmentation fault (core
dumped)
dt10@LAPTOP-0DEHDEQ0:~
$ ./a.out
cat cat dog donkey
dt10@LAPTOP-0DEHDEQ0:~
$ 
```

Motivating example: histograms

```
int main()
{
```

What type allows us to index by strings?

```
  vector<int> histogram;

  string x;
  while( cin >> x) {

    if( x >= histogram.size() ){
        histogram.resize( x+1 );
    }

    histogram[x] += 1;
  }


  for(int i=0;i<histogram.size();i++){
    cout<<i<<","<<histogram[i]<<endl;
  }
}
```



```
dt10@LAPTOP-0DEHDEQ0:~
$ ./a.out
1 2 3 1 1 1 2 3 4 1 1 2
0, 0
1, 6
2, 3
3, 2
4, 1
dt10@LAPTOP-0DEHDEQ0:~
$ ./a.out
1000000000
Segmentation fault (core
dumped)
dt10@LAPTOP-0DEHDEQ0:~
$ ./a.out
cat cat dog donkey
dt10@LAPTOP-0DEHDEQ0:~
$
```

map<K,V> : a new friend

`vector<T>` : a mapping from the integers [0,n)
to values of type T

We can read and write the value at any index

We can change the size n

`map<K,V>` : a mapping from keys of type K
to values of type V

We can insert a value at a new key

We can read and write the value at any key

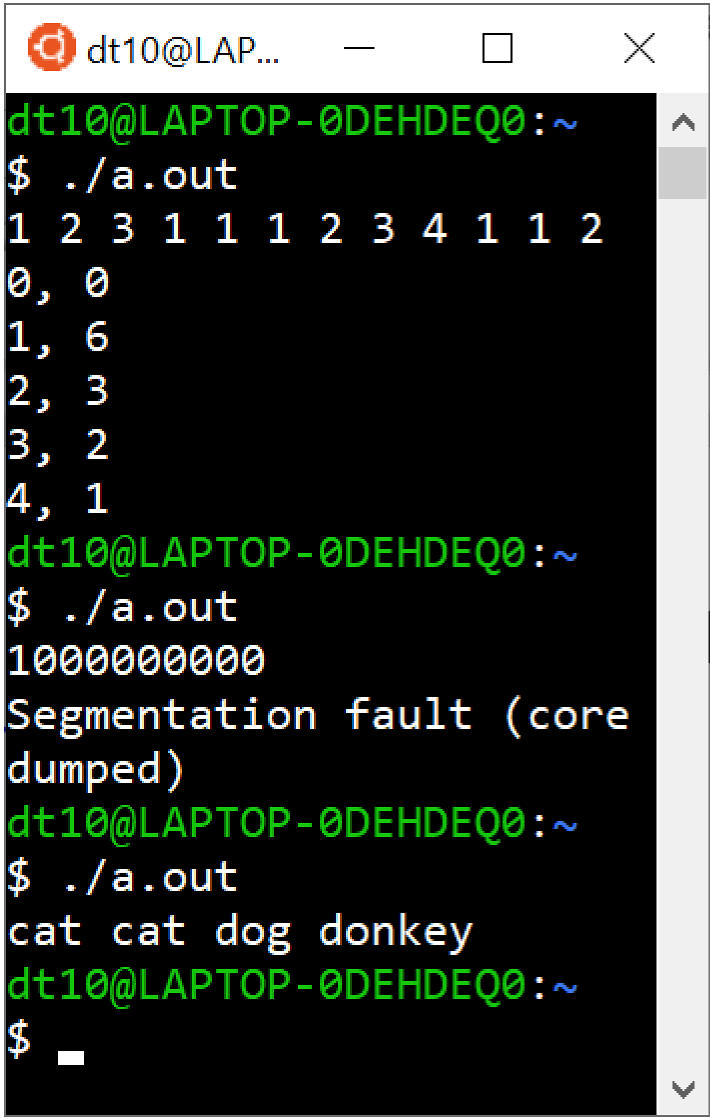
We can delete the value at a given key

Motivating example: histograms

```
int main()
{
```

A histogram is a mapping of strings to integers

```
  map<string,int> histogram;

  string x;
  while( cin >> x) {
    histogram[x] += 1;
  }


  for(int i=0;i<histogram.size();i++){
    cout<<i<<","<<histogram[i]<<endl;
  }
}
```



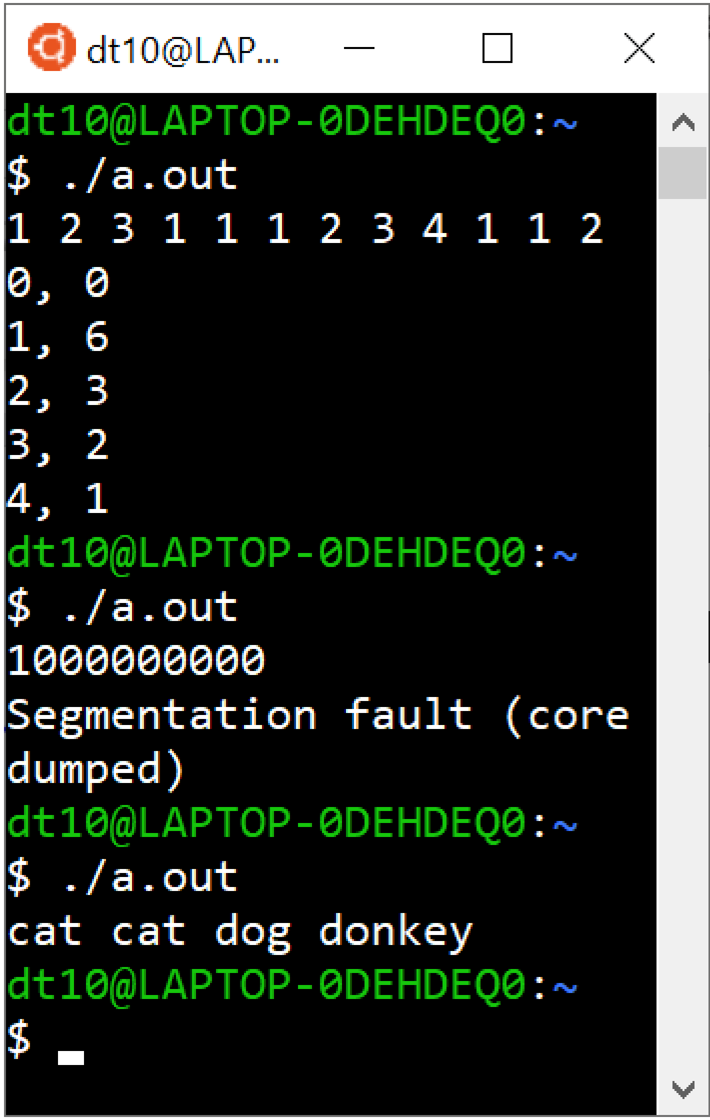
```
dt10@LAPTOP-0DEHDEQ0:~
$ ./a.out
1 2 3 1 1 1 2 3 4 1 1 2
0, 0
1, 6
2, 3
3, 2
4, 1
dt10@LAPTOP-0DEHDEQ0:~
$ ./a.out
1000000000
Segmentation fault (core
dumped)
dt10@LAPTOP-0DEHDEQ0:~
$ ./a.out
cat cat dog donkey
dt10@LAPTOP-0DEHDEQ0:~
$ _
```

Motivating example: histograms

```
int main()
{

  map<string,int> histogram;

  string x;
  while( cin >> x) {
    histogram[x] += 1;
  }
```

Increment the mapping associated with x by one

```
  for(int i=0;i<histogram.size();i++){
    cout<<i<<","<<histogram[i]<<endl;
  }
}
```

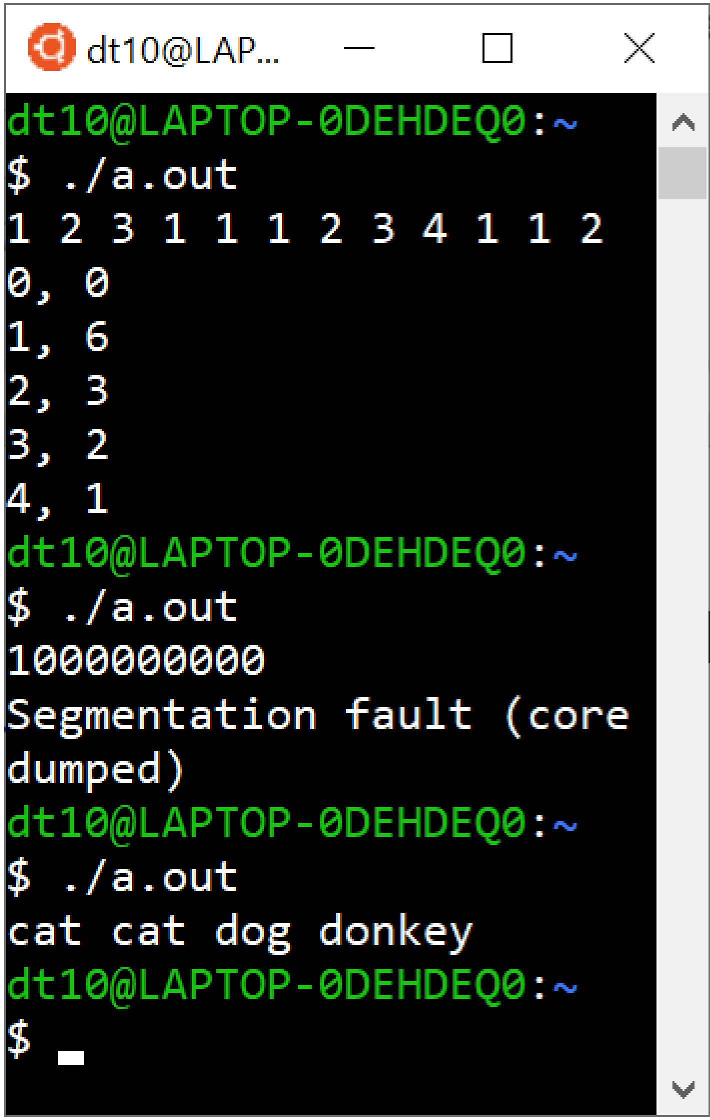


```
dt10@LAPTOP-0DEHDEQ0:~
$ ./a.out
1 2 3 1 1 1 2 3 4 1 1 2
0, 0
1, 6
2, 3
3, 2
4, 1
dt10@LAPTOP-0DEHDEQ0:~
$ ./a.out
1000000000
Segmentation fault (core
dumped)
dt10@LAPTOP-0DEHDEQ0:~
$ ./a.out
cat cat dog donkey
dt10@LAPTOP-0DEHDEQ0:~
$ 
```

Motivating example: histograms

```
int main()
{

  map<string,int> histogram;

  string x;
  while( cin >> x) {
     int count = histogram[x];
     count = count + 1;
     histogram[x] = count ;
  }


  for(int i=0;i<histogram.size();i++){
    cout<<i<<","<<histogram[i]<<endl;
  }
}
```

Read the current value at x;
if there is no current value, then
insert default constructed value: `int()==0`

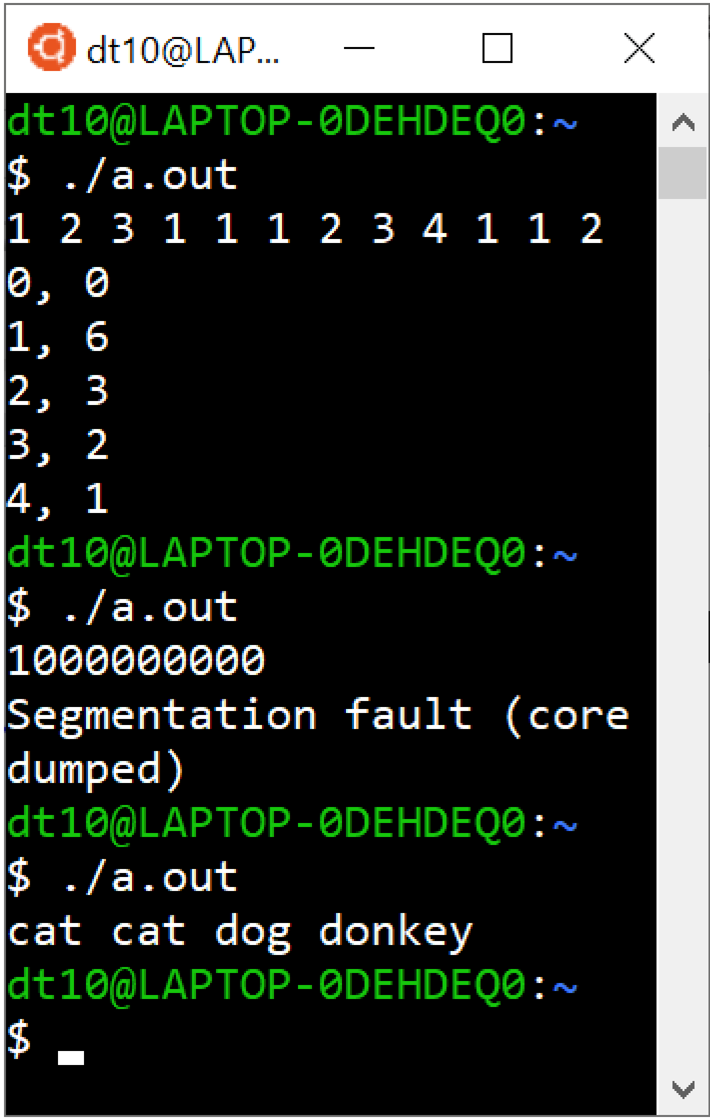


```
dt10@LAPTOP-0DEHDEQ0:~
$ ./a.out
1 2 3 1 1 1 2 3 4 1 1 2
0, 0
1, 6
2, 3
3, 2
4, 1
dt10@LAPTOP-0DEHDEQ0:~
$ ./a.out
1000000000
Segmentation fault (core
dumped)
dt10@LAPTOP-0DEHDEQ0:~
$ ./a.out
cat cat dog donkey
dt10@LAPTOP-0DEHDEQ0:~
$ _
```

Motivating example: histograms

```
int main()
{

  map<string,int> histogram;

  string x;
  while( cin >> x) {
     int count = histogram[x];
     count = count + 1;
     histogram[x] = count ;
  }


  for(int i=0;i<histogram.size();i++){
    cout<<i<<","<<histogram[i]<<endl;
  }
}
```

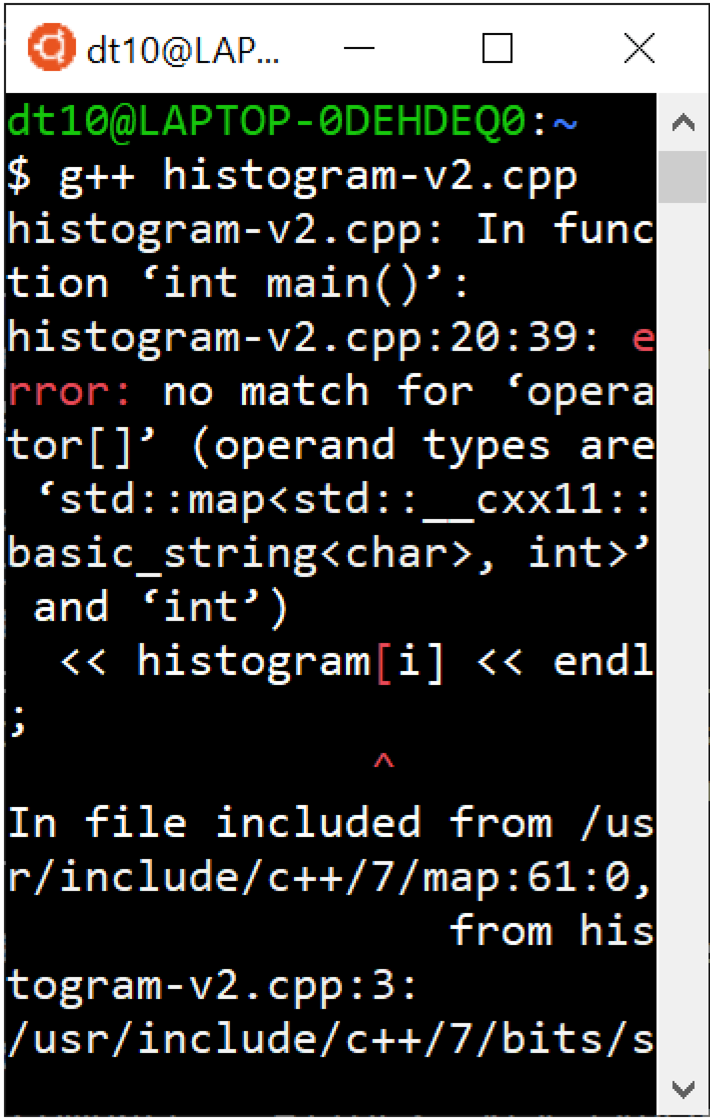
Write a new value for key x



```
dt10@LAPTOP-0DEHDEQ0:~
$ ./a.out
1 2 3 1 1 1 2 3 4 1 1 2
0, 0
1, 6
2, 3
3, 2
4, 1
dt10@LAPTOP-0DEHDEQ0:~
$ ./a.out
1000000000
Segmentation fault (core
dumped)
dt10@LAPTOP-0DEHDEQ0:~
$ ./a.out
cat cat dog donkey
dt10@LAPTOP-0DEHDEQ0:~
$ 
```

Motivating example: histograms

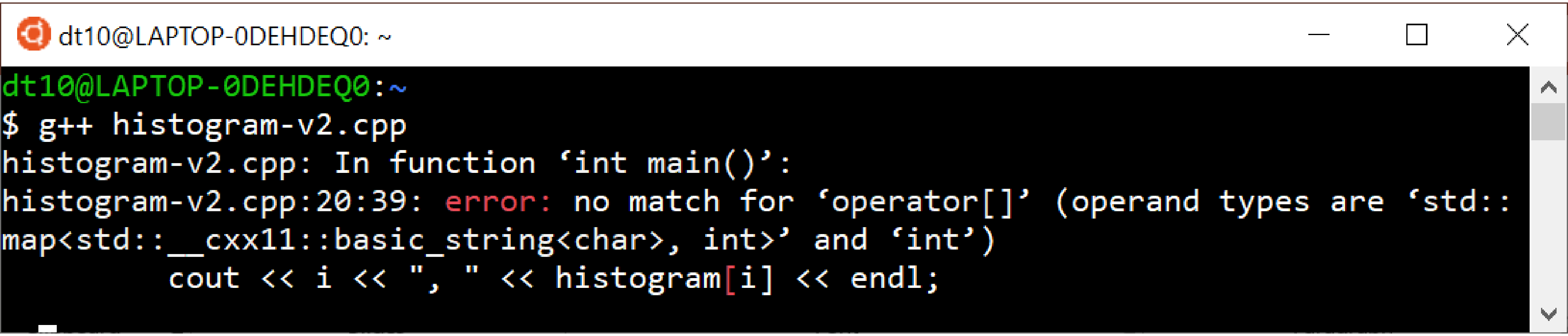
```
int main()
{

  map<string,int> histogram;

  string x;
  while( cin >> x) {
     int count = histogram[x];
     count = count + 1;
     histogram[x] = count ;
  }


  for(int i=0;i<histogram.size();i++){
    cout<<i<<","<<histogram[i]<<endl;
  }
}
```



```
dt10@LAPTOP-0DEHDEQ0:~
$ g++ histogram-v2.cpp
histogram-v2.cpp: In func
tion ‘int main()’:
histogram-v2.cpp:20:39: e
rror: no match for ‘opera
tor[]’ (operand types are
 ‘std::map<std::__cxx11::
basic_string<char>, int>’
 and ‘int’)
  << histogram[i] << endl
;
              ^
In file included from /us
r/include/c++/7/map:61:0,
                 from his
togram-v2.cpp:3:
/usr/include/c++/7/bits/s
```

Motivating example: histograms

```
int main()
{

  map<string,int> histogram;
```



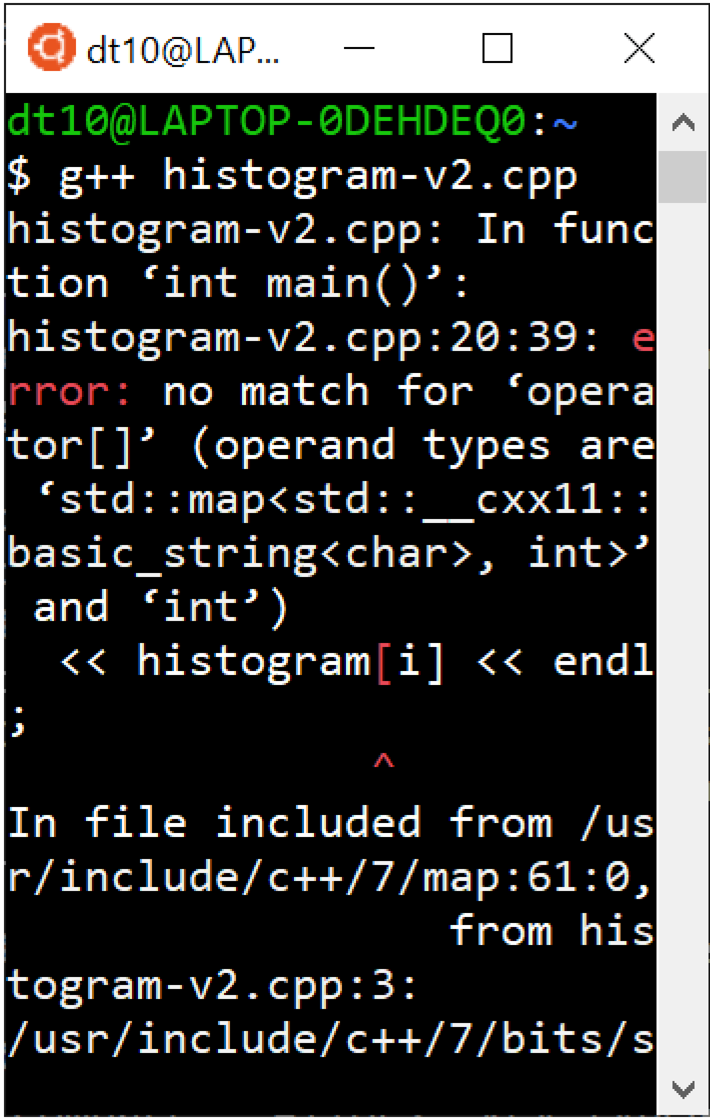
```
dt10@LAPTOP-0DEHDEQ0:~
$ g++ histogram-v2.cpp
histogram-v2.cpp: In function ‘int main()’:
histogram-v2.cpp:20:39: error: no match for ‘operator[]’ (operand types are ‘std::
map<std::__cxx11::basic_string<char>, int>’ and ‘int’)
         cout << i << ", " << histogram[i] << endl;
```

```
  for(int i=0;i<histogram.size();i++){
    cout<<i<<","<<histogram[i]<<endl;
  }
}
```

We can only index using a string
Cannot access all the values in this way

Motivating example: histograms

```
int main()
{

  map<string,int> histogram;

  string x;
  while( cin >> x) {
     int count = histogram[x];
     count = count + 1;
     histogram[x] = count ;
  }


  for(int i=0;i<histogram.size();i++){
    cout<<i<<","<<histogram[i]<<endl;
  }
}
```

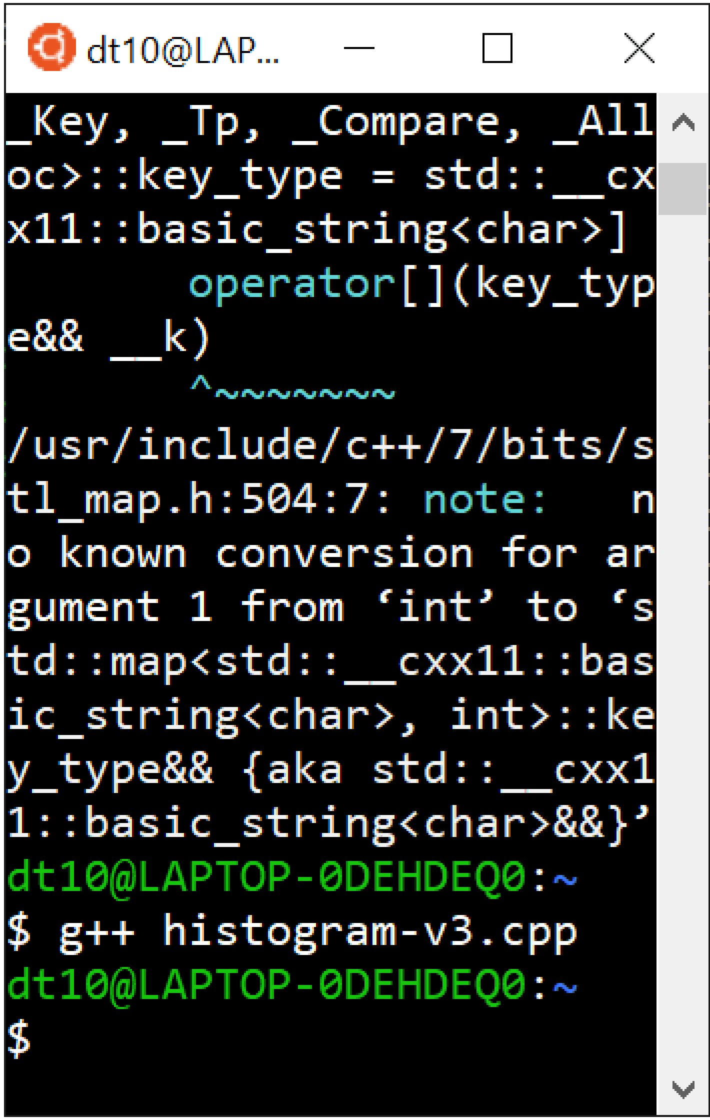


```
dt10@LAPTOP-0DEHDEQ0:~
$ g++ histogram-v2.cpp
histogram-v2.cpp: In func
tion ‘int main()’:
histogram-v2.cpp:20:39: e
rror: no match for ‘opera
tor[]’ (operand types are
 ‘std::map<std::__cxx11::
basic_string<char>, int>’
 and ‘int’)
  << histogram[i] << endl
;
              ^
In file included from /us
r/include/c++/7/map:61:0,
                 from his
togram-v2.cpp:3:
/usr/include/c++/7/bits/s
```

Motivating example: histograms

```
int main()
{

  map<string,int> histogram;

  string x;
  while( cin >> x) {
     int count = histogram[x];
     count = count + 1;
     histogram[x] = count ;
  }


  for(int i=0;i<histogram.size();i++){
    cout<<i<<" "<<histogram[i]<<endl;
  }
}
```

Printing : TODO

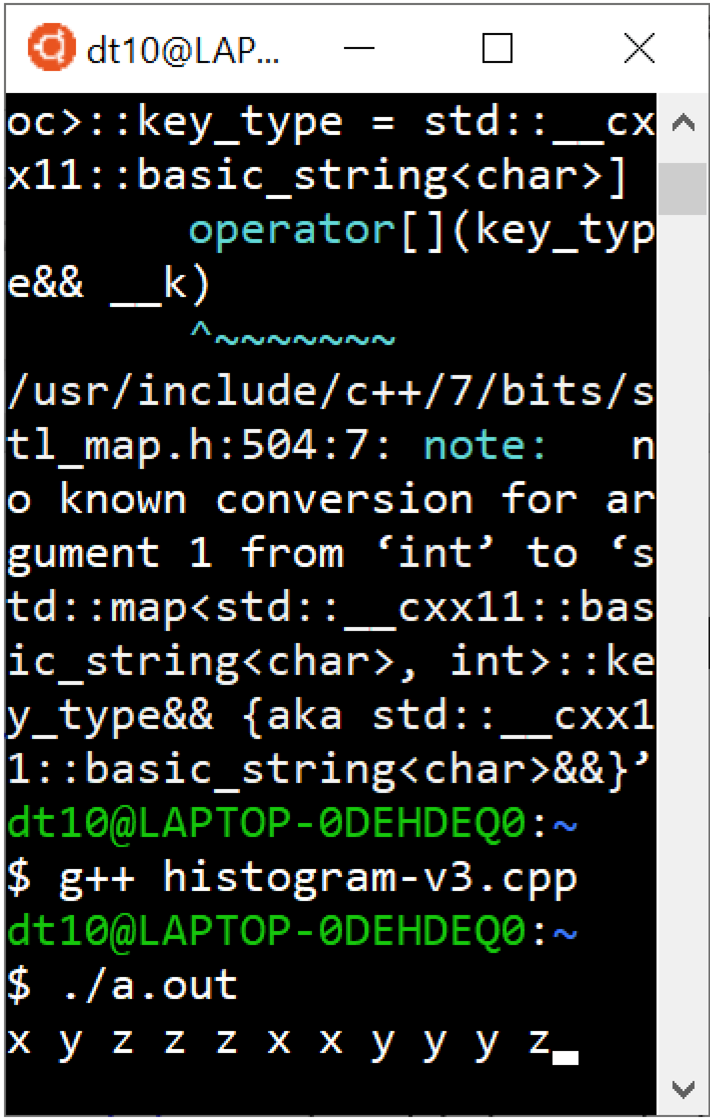


```
_Key, _Tp, _Compare, _All
oc>::key_type = std::__cx
x11::basic_string<char>]
      operator[](key_typ
e&& __k)
      ^~~~~~~~
/usr/include/c++/7/bits/s
tl_map.h:504:7: note:   n
o known conversion for ar
gument 1 from ‘int’ to ‘s
td::map<std::__cxx11::bas
ic_string<char>, int>::ke
y_type&& {aka std::__cxx1
1::basic_string<char>&&}’
dt10@LAPTOP-0DEHDEQ0:~
$ g++ histogram-v3.cpp
dt10@LAPTOP-0DEHDEQ0:~
$
```

Motivating example: histograms

```
int main()
{

  map<string,int> histogram;

  string x;
  while( cin >> x) {
     int count = histogram[x];
     count = count + 1;
     histogram[x] = count ;
  }


  for(int i=0;i<histogram.size();i++){
    cout<<i<<" "<<histogram[i]<<endl;
  }
}
```

Printing : TODO

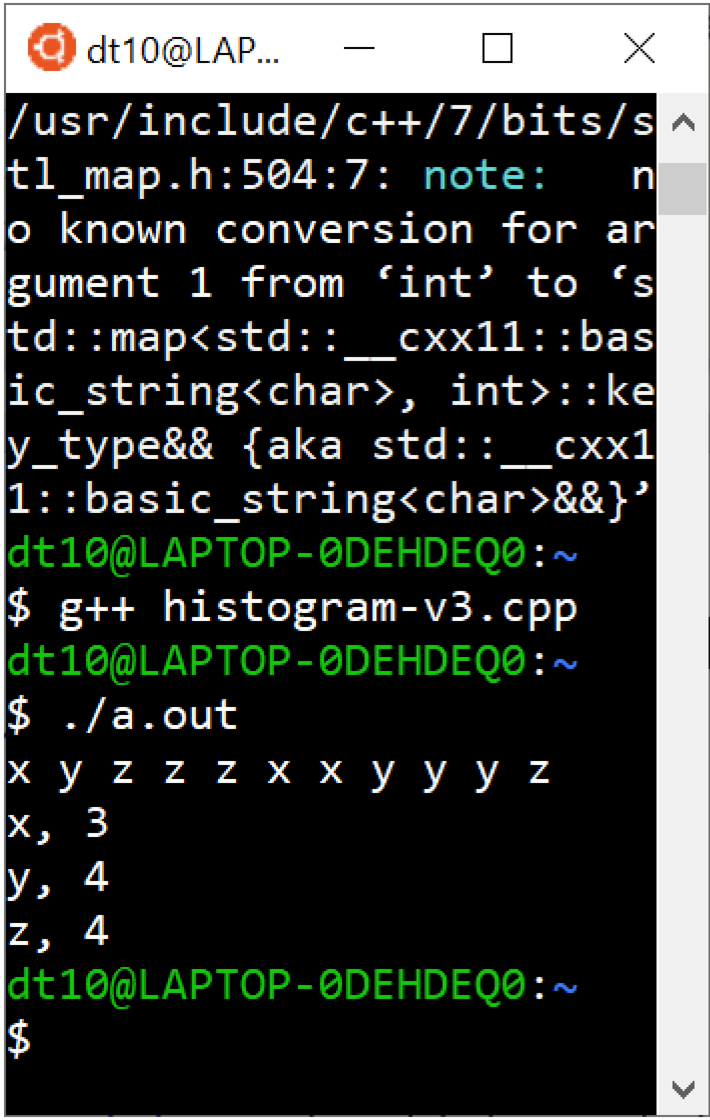


```
oc>::key_type = std::__cx
x11::basic_string<char>]
       operator[](key_typ
e&& __k)
       ^~~~~~~~
/usr/include/c++/7/bits/s
tl_map.h:504:7: note:   n
o known conversion for ar
gument 1 from ‘int’ to ‘s
td::map<std::__cxx11::bas
ic_string<char>, int>::ke
y_type&& {aka std::__cxx1
1::basic_string<char>&&}’
dt10@LAPTOP-0DEHDEQ0:~
$ g++ histogram-v3.cpp
dt10@LAPTOP-0DEHDEQ0:~
$ ./a.out
x y z z z x x y y y z
```

Motivating example: histograms

```
int main()
{

  map<string,int> histogram;

  string x;
  while( cin >> x) {
     int count = histogram[x];
     count = count + 1;
     histogram[x] = count ;
  }


  for(int i=0;i<histogram.size();i++){
    cout<<i<<", "<<histogram[i]<<endl;
  }
}
```

Printing : TODO



```
/usr/include/c++/7/bits/s
tl_map.h:504:7: note:   n
o known conversion for ar
gument 1 from ‘int’ to ‘s
td::map<std::__cxx11::bas
ic_string<char>, int>::ke
y_type&& {aka std::__cxx1
1::basic_string<char>&&}’
dt10@LAPTOP-0DEHDEQ0:~
$ g++ histogram-v3.cpp
dt10@LAPTOP-0DEHDEQ0:~
$ ./a.out
x y z z z x x y y y z
x, 3
y, 4
z, 4
dt10@LAPTOP-0DEHDEQ0:~
$
```

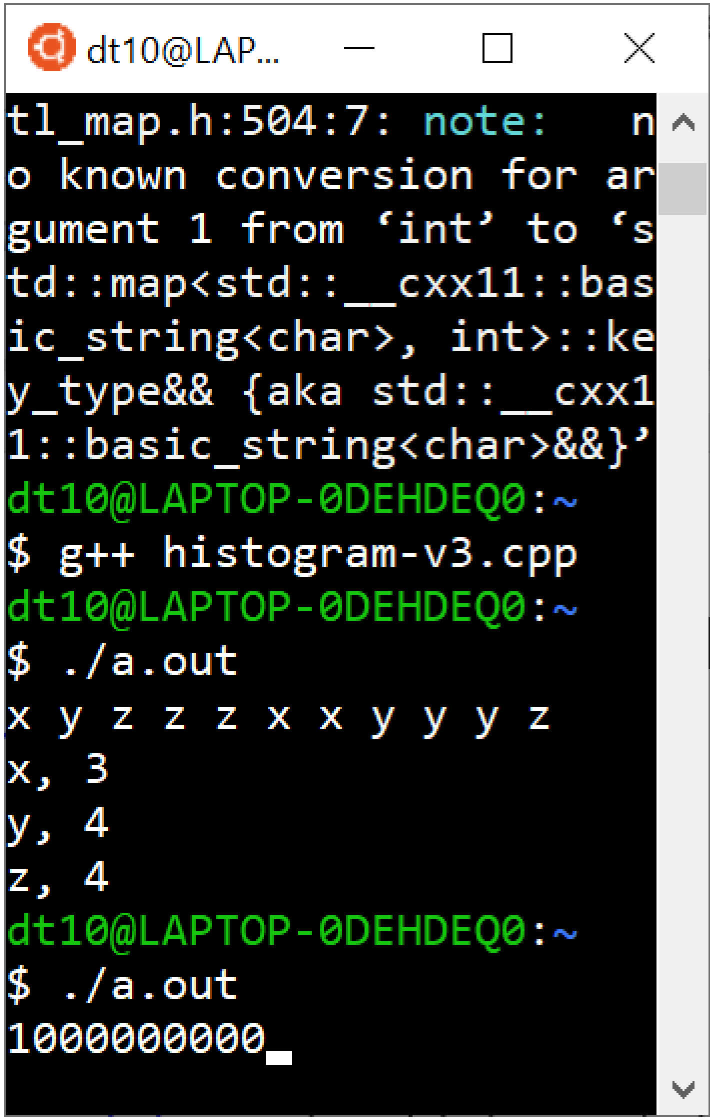
Motivating example: histograms

```
int main()
{

  map<string,int> histogram;

  string x;
  while( cin >> x) {
     int count = histogram[x];
     count = count + 1;
     histogram[x] = count ;
  }
```

```
for(int i=0;i<histogram.size();i++){
  cout<<i<<", "<<histogram[i]<<endl;
}
```

Printing : TODO

```
}
```

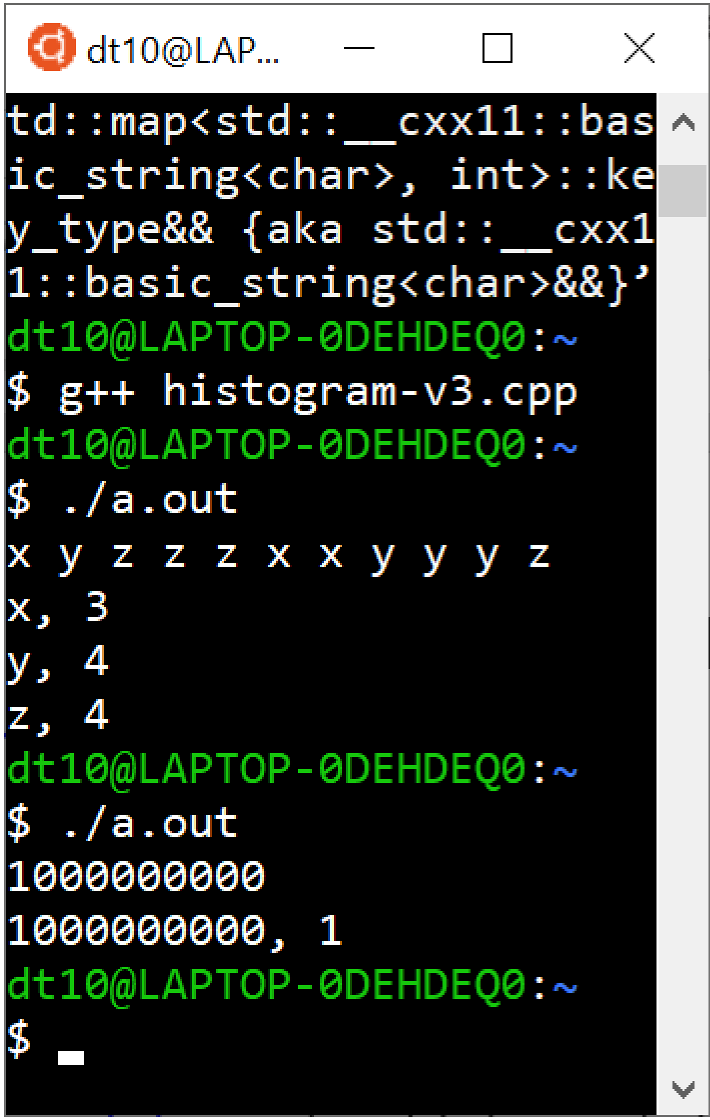


```
tl_map.h:504:7: note:   n
o known conversion for ar
gument 1 from ‘int’ to ‘s
td::map<std::__cxx11::bas
ic_string<char>, int>::ke
y_type&& {aka std::__cxx1
1::basic_string<char>&&}’
dt10@LAPTOP-0DEHDEQ0:~
$ g++ histogram-v3.cpp
dt10@LAPTOP-0DEHDEQ0:~
$ ./a.out
x y z z z x x y y y z
x, 3
y, 4
z, 4
dt10@LAPTOP-0DEHDEQ0:~
$ ./a.out
1000000000_
```

Motivating example: histograms

```
int main()
{

  map<string,int> histogram;

  string x;
  while( cin >> x) {
     int count = histogram[x];
     count = count + 1;
     histogram[x] = count ;
  }


  for(int i=0;i<histogram.size();i++){
    cout<<i<<", "<<histogram[i]<<endl;
  }
}
```

Printing : TODO

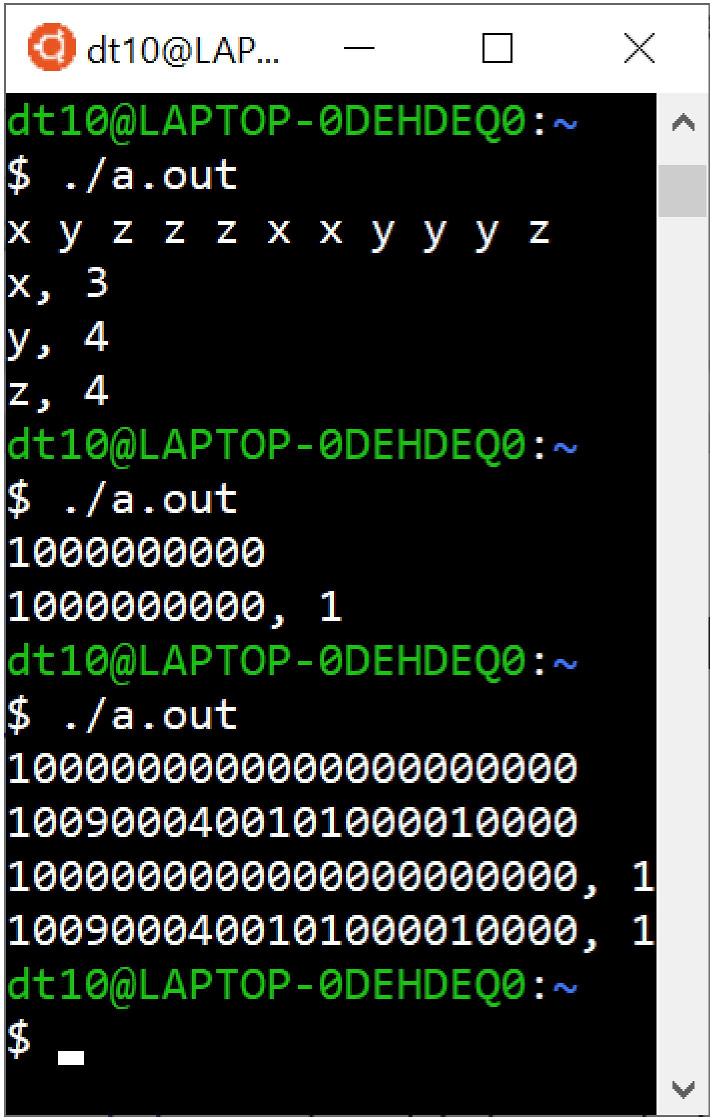


```
td::map<std::__cxx11::bas
ic_string<char>, int>::ke
y_type&& {aka std::__cxx1
1::basic_string<char>&&}’
dt10@LAPTOP-0DEHDEQ0:~
$ g++ histogram-v3.cpp
dt10@LAPTOP-0DEHDEQ0:~
$ ./a.out
x y z z z x x y y y z
x, 3
y, 4
z, 4
dt10@LAPTOP-0DEHDEQ0:~
$ ./a.out
1000000000
1000000000, 1
dt10@LAPTOP-0DEHDEQ0:~
$ _
```

Motivating example: histograms

```
int main()
{

  map<string,int> histogram;

  string x;
  while( cin >> x) {
     int count = histogram[x];
     count = count + 1;
     histogram[x] = count ;
  }


  for(int i=0;i<histogram.size();i++){
    cout<<i<<", "<<histogram[i]<<endl;
  }
}
```

Printing : TODO



```
dt10@LAPTOP-0DEHDEQ0:~
$ ./a.out
x y z z z x x y y y z
x, 3
y, 4
z, 4
dt10@LAPTOP-0DEHDEQ0:~
$ ./a.out
1000000000
1000000000, 1
dt10@LAPTOP-0DEHDEQ0:~
$ ./a.out
1000000000000000000000
1009000400101000010000
1000000000000000000000, 1
1009000400101000010000, 1
dt10@LAPTOP-0DEHDEQ0:~
$ _
```

map<K,V> vs vector<T>

Indices versus keys

vector<T> : maps indices (naturals) to values from T

map<K,V> : maps keys of type K to values from V

Sparse versus dense

vector<T> : all values in range [0,size()) are allocated

map<K,V> : only store values if the key has been added

Iteration

vector<T> : we can iterate over integers in [0,size())

map<K,V> : ?

map<K,V> : *approximate* API

```
template<class Key, class Value>
class map
{
public:
    map();
    map(const map &);
    ~map();

    map &operator=(const map &);

    int size() const;

    void insert(const pair<Key,Value> &key_value);

    const Value &at(const Key &key) const;
    Value &at(const Key &key);

    Value &operator[](const Key &key);
};
```

Types that can go in a map

- We have two types involved:
  - K : the type of the key
  - V : the type of the value
- Both types must be copyable and assignable
  - The container needs to move them about internally
- The Key type must also be "Comparable"

```
template<class T>
bool Comparable(const T &x, const T & y)
{
    // Can tell if one value is less than the other
    return x < y;
}
```

Implementation of map

```
template<typename K, typename V>
class map
{
private:
    struct node
    {
        K key;
        V value;
        node *left;
        node *right;
    };

    node m_root;
public:
    V &at(const K &v)
    {
        return find_node(m_root, v)->value;
    }
};
```

Implementation of map

- The STL map is always some kind of sorted tree
  - The search functions would look familiar to you
  - The internal nodes are fairly understandable
- The complexity comes from balancing the tree
  - map ***guarantees*** that operations take ~log size steps
  - How it does that depends on the implementation
- A lot of the value comes from not needing to care
  - We do not care how g++ implements map as long as:

    It is functionally correct : all the operations work

    Its performance fits the documented requirements

Iterators and iteration

Motivating example: histograms

```
int main()
{

  map<string,int> histogram;

  string x;
  while( cin >> x) {
     int count = histogram[x];
     count = count + 1;
     histogram[x] = count ;
  }

```

Printing : TODO

[redacted]

```
}
```

Motivating example: histograms

```
int main()
{

  map<string,int> histogram;

  string x;
  while( cin >> x) {
     int count = histogram[x];
     count = count + 1;
     histogram[x] = count ;
  }


  for(int i=0;i<histogram.size();i++){
    cout<<i<<","<<histogram[i]<<endl;
  }
}
```

How do we find out what keys are in the histogram?

The "iterator" concept

- The STL uses the idea of iterators extensively
  - Used to access and manipulate containers
  - Used to pass arguments to algorithms
- An "iterator" is an abstract version of a pointer
  - An iterator is any type that "behaves" enough like a pointer
  - A pointer is a type that behaves like a pointer
  - A pointer can be used as an iterator
- Iterators are easier to understand as pointers

  "What would a pointer do in these circumstances"?

Don't worry : you only need to use it

- You will ***not*** need to implement an iterator
  - Very few people need to do that
  - Actually quite hard to get completely right
- You only need to ***use*** iterators
  - They are quite easy to use in practise
  - Just think of them as fancy pointers

vector<T> and accumulate

```
template<class T>
T accumulate(const T *begin, const T *end, T identity)
{
    T sum=identity;
    while(begin != end){
        sum += *begin;
        begin++;  // Sugar for begin+=1
    }
    return sum;
}
```

```
float sum_vector(const vector<float> &v)
{
    return accumulate( &v[0], &v[v.size()], 0.0f );
}
```

vector<T> and accumulate

```
template<class T>
T accumulate(const T *begin, const T *end, T identity);

const float *begin(const vector<float> &v)
{
    return &v[0];
}

const float *end(const vector<float> &v)
{
    return &v[v.size()];
}


float sum_vector(const vector<float> &v)
{
    return accumulate( &v[0], &v[v.size()], 0.0f );
}
```

vector<T> and accumulate

```
template<class T>
T accumulate(const T *begin, const T *end, T identity);

const float *begin(const vector<float> &v)
{
    return &v[0];
}

const float *end(const vector<float> &v)
{
    return &v[v.size()];
}


float sum_vector(const vector<float> &v)
{
    return accumulate( begin(v), end(v), 0.0f );
}
```

vector<T> and accumulate

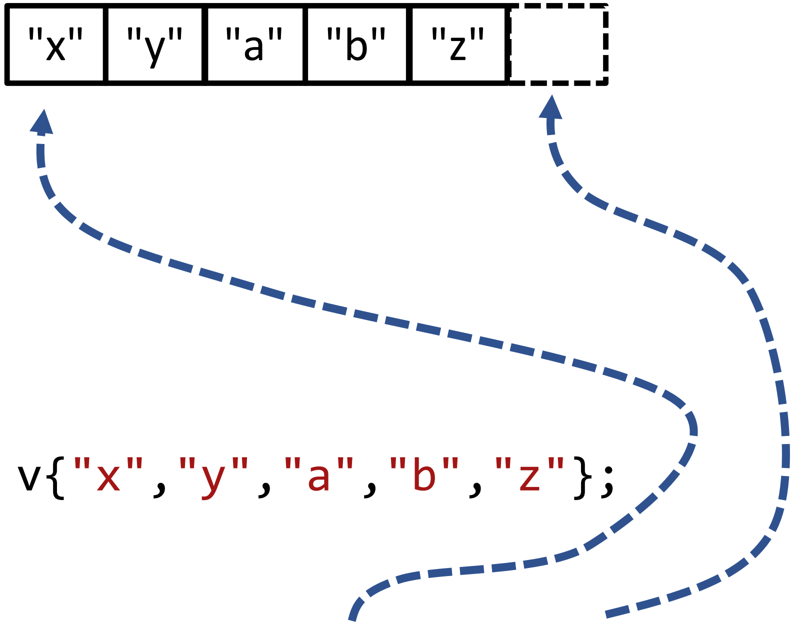
```
template<class T>
T accumulate(const T *begin, const T *end, T identity);

template<class T>
class vector
{
public:
    const T *begin() const
    { return &m_data[0]; }

    const T *end() const
    { return &m_data[size()]; }
};

float sum_vector(const vector<float> &v)
{
    return accumulate( v.begin(), v.end(), 0.0f );
}
```

vector<T> and accumulate

```
template<class T>
T accumulate(const T *begin, const T *end, T identity);

template<class T>
class vector
{
public:
    const T *begin() const
    { return &m_data[0]; }

    const T *end() const
    { return &m_data[size()]; }
};

float sum_vector(const vector<float> &v)
{
    return accumulate( v.begin(), v.end()-10, 0.0f );
}
```

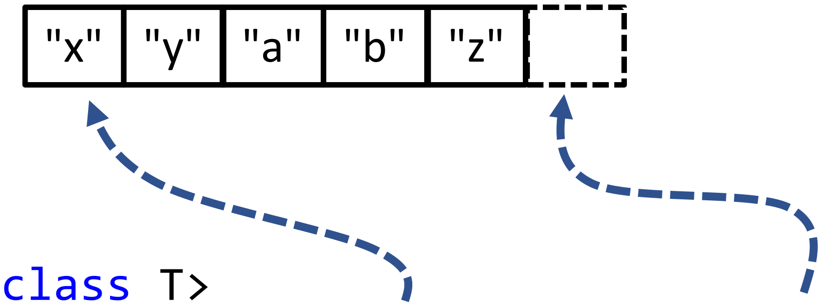
Accumulate all but the last 10 values

Accumulating : vector iterators



```
int main()
{
    vector<string> v{"x","y","a","b","z"};


    string acc=accumulate(v.begin(), v.end(), string(""));

    cout << acc << endl;  // Prints xyabz
}
```

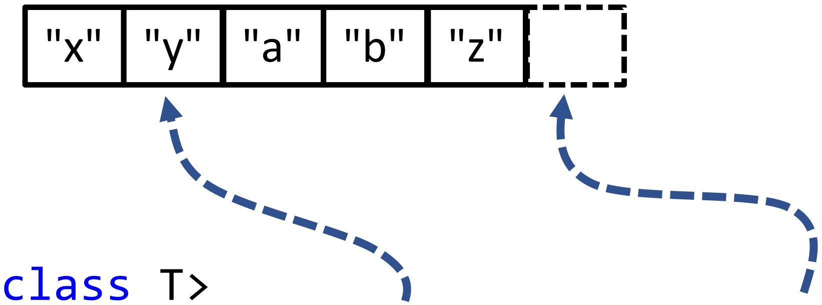
Accumulating : vector iterators



"x" "y" "a" "b" "z"

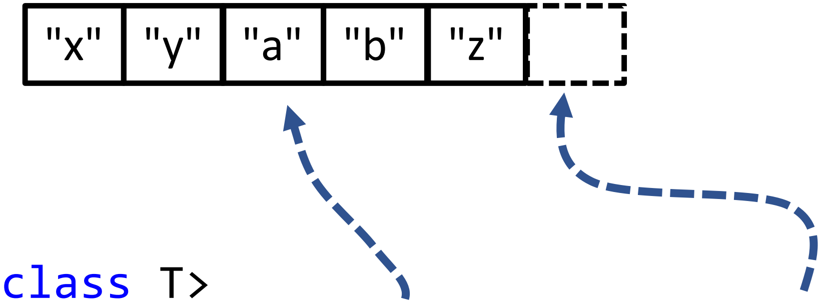
```
template<class T>
T accumulate(const T *begin, const T *end, T identity)
{
    T sum=identity;
    while(begin != end){
        sum += *begin; // sum=="x"
        begin++;
    }
    return sum;
}
```

Accumulating : vector iterators



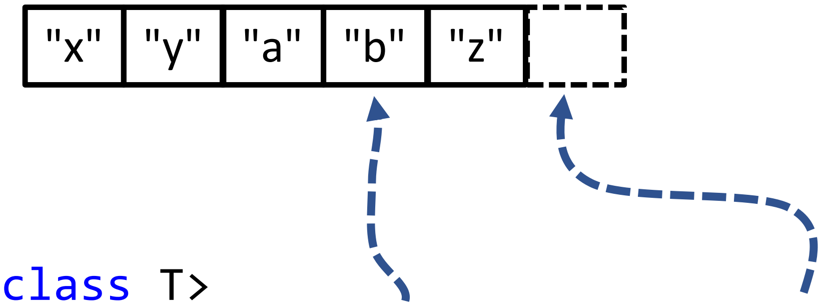
```
template<class T>
T accumulate(const T *begin, const T *end, T identity)
{
    T sum=identity;
    while(begin != end){
        sum += *begin; // sum=="xy"
        begin++;
    }
    return sum;
}
```

Accumulating : vector iterators



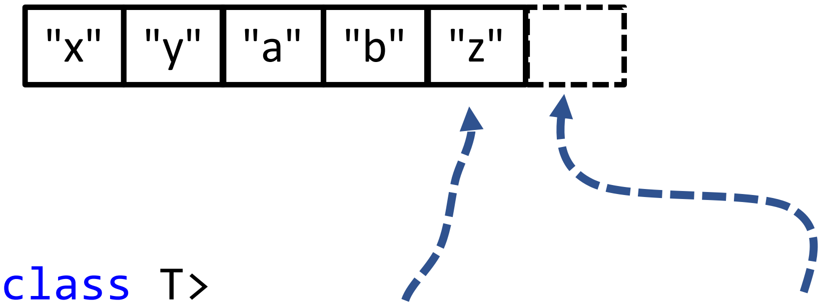
```
template<class T>
T accumulate(const T *begin, const T *end, T identity)
{
    T sum=identity;
    while(begin != end){
        sum += *begin; // sum=="xya"
        begin++;
    }
    return sum;
}
```

Accumulating : vector iterators



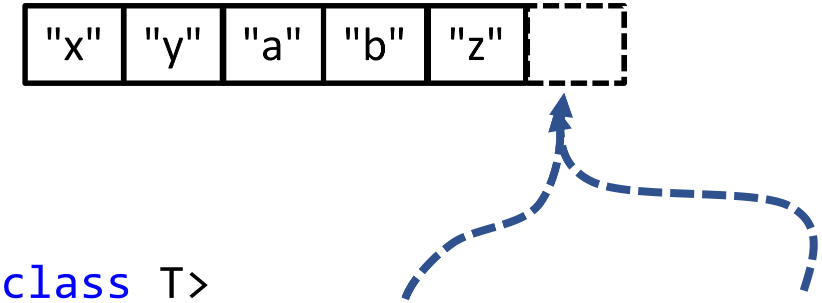
```
template<class T>
T accumulate(const T *begin, const T *end, T identity)
{
    T sum=identity;
    while(begin != end){
        sum += *begin; // sum=="xyab"
        begin++;
    }
    return sum;
}
```

Accumulating : vector iterators



```
template<class T>
T accumulate(const T *begin, const T *end, T identity)
{
    T sum=identity;
    while(begin != end){
        sum += *begin; // sum=="xyabz"
        begin++;
    }
    return sum;
}
```

Accumulating : vector iterators



```
template<class T>
T accumulate(const T *begin, const T *end, T identity)
{
    T sum=identity;
    while(begin != end){
        sum += *begin; // sum=="xyabz"
        begin++;
    }
    return sum;
}
```

list<T> and accumulate

```
template<class T>
T accumulate(const T *begin, const T *end, T identity);

template<class T>
class list
{
    struct node{
        T value;
        node *next;
    };
    node *m_begin;
public:
    const node *begin() const { return m_begin; }
    const node *end() const { return nullptr; }
};
```

list<T> and accumulate

```
template<class T>
T accumulate(const T *begin, const T *end, T identity);

template<class T>
class list
{
    struct node;
    node *m_begin;
public:
    const node *begin() const { return m_begin; }
    const node *end() const { return nullptr; }
};

float sum_list(const list<float> &l)
{
    return accumulate( l.begin(), l.end(), 0.0f );
}                    list::node* list::node* float
```

list<T> and accumulate

- We can make it consistent as long as
  - begin() returns a type that "looks like" a pointer
  - end() returns a type that "looks like" a pointer
  - incrementation (++) works on the iterator type
  - de-referencing (*) returns the actual value
- Operator overloading lets us do all that

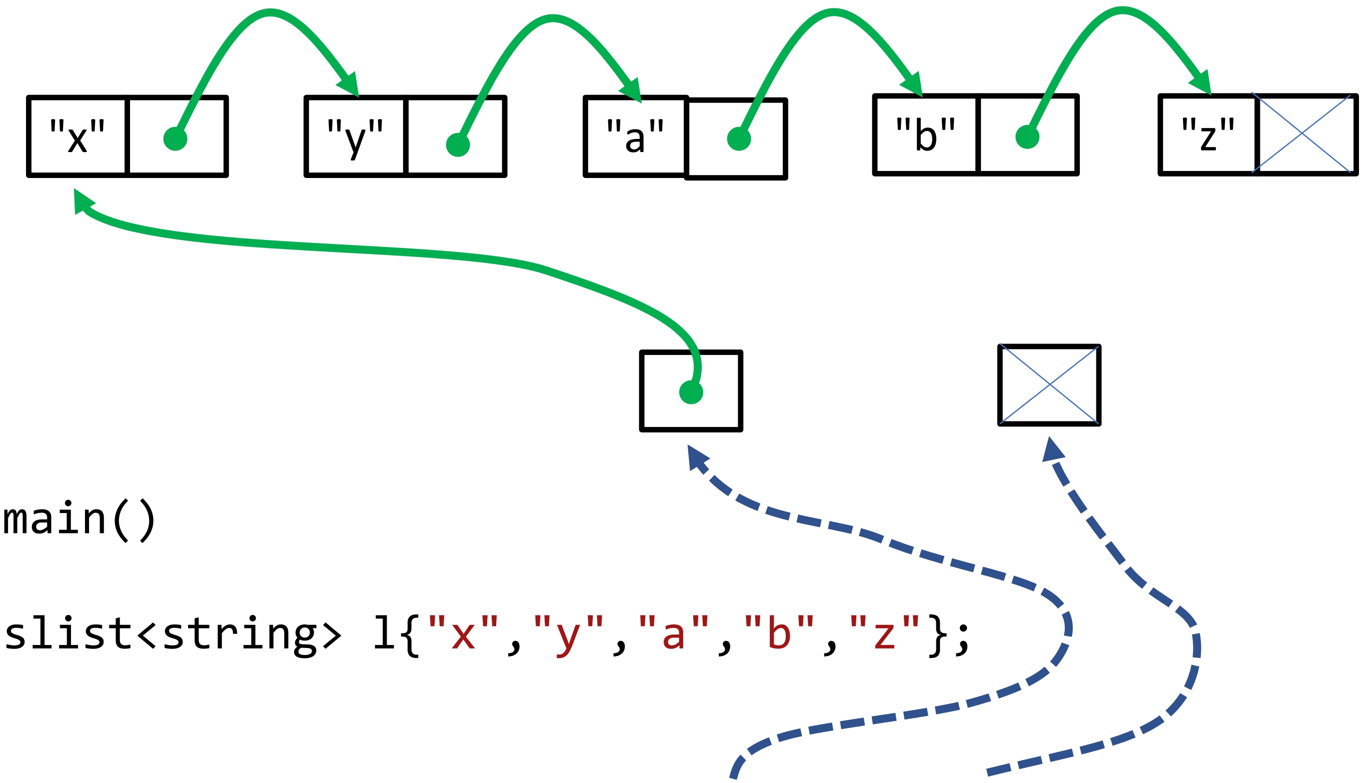
...but, we only want to understand it, not to do it.

A *sketch* of a list iterator

```
template<class T>
class list
{
 struct node
 {
  T value;
  node *next;
 };

  node *m_begin;
public:
 NodeIt begin() const
 { return NodeIt{m_begin}; }

 NodeIt end() const
 { return NodeIt{nullptr}; }
};
```

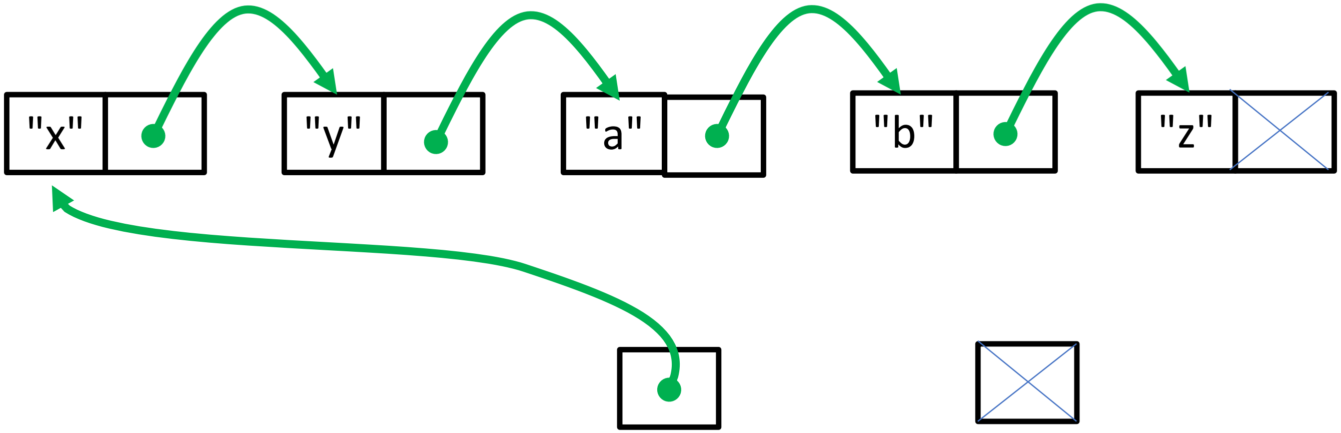
```
template<class T>
class NodeIt
{
 list<T>::node *n;

 NodeIt &operator++()
 {
  n=n->next;
  return *this;
 }

 T &operator *()
 { return n->value; }

 bool operator!=(const NodeIt &o)
 { return n != o.n; }
};
```

Accumulating : list iterators



```
int main()
{
    slist<string> l{"x","y","a","b","z"};


    string acc=accumulate(l.begin(), l.end(), string(""));

    cout << acc << endl;  // Prints xyabz
}
```

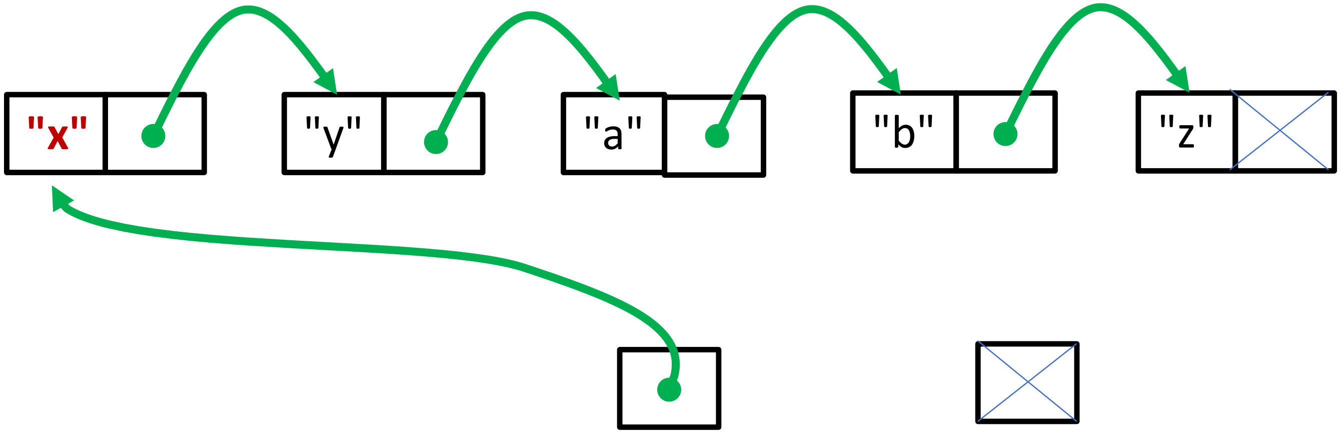
Accumulating : list iterators



```
template<class TIt, class T>
T accumulate(TIt begin, TIt end, T identity)
{
    T sum=identity;
    while(begin != end){
        sum += *begin; // sum==""
        begin++;
    }
    return sum;
}
```

*Overloaded * operator*

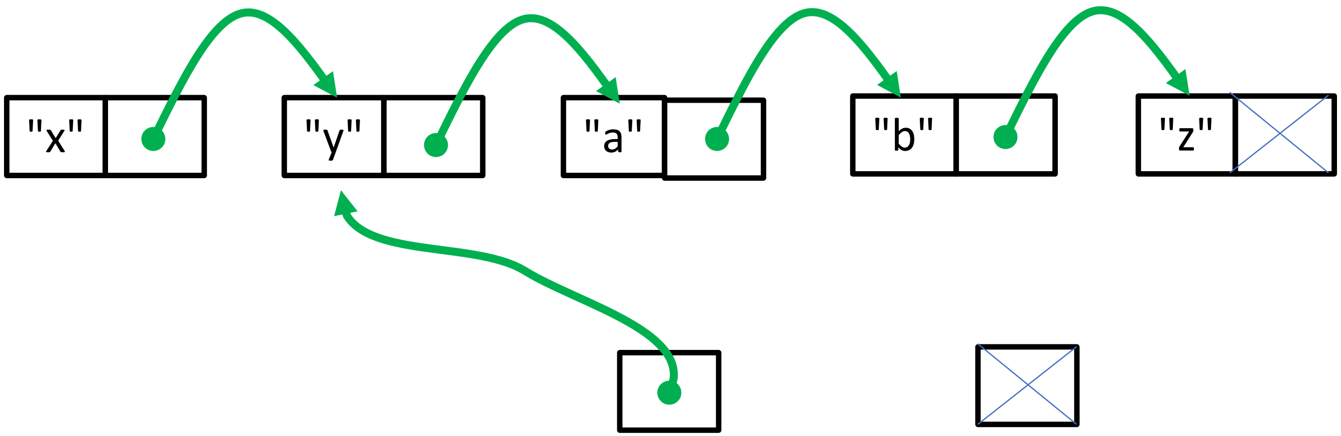
Accumulating : list iterators



```
template<class TIt, class T>
T accumulate(TIt begin, TIt end, T identity)
{
	T sum=identity;
	while(begin != end){
		sum += *begin; // sum=="x"
		begin++;
	}
	return sum;
}
```

*Overloaded * operator*

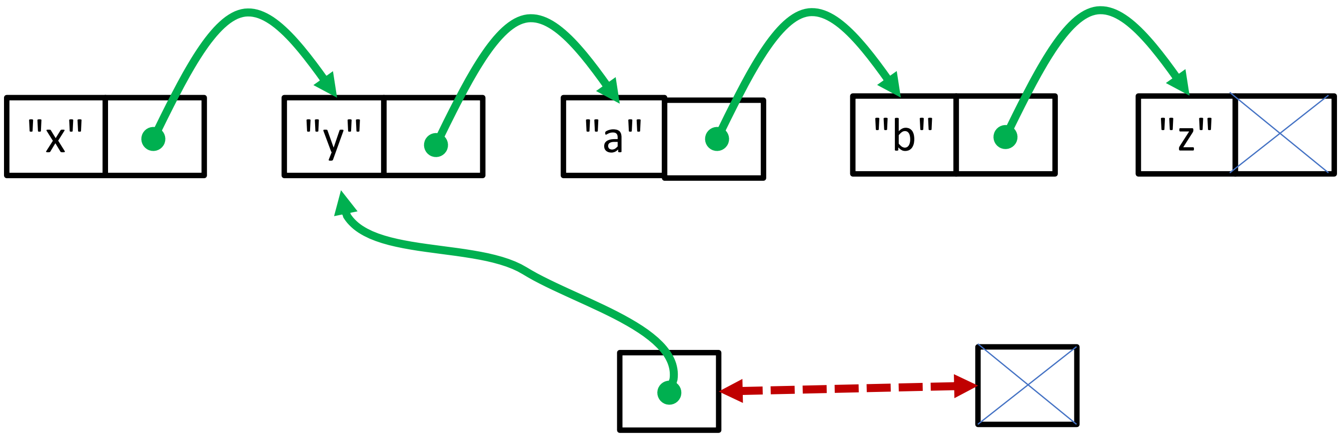
Accumulating : list iterators



```
template<class TIt, class T>
T accumulate(TIt begin, TIt end, T identity)
{
	T sum=identity;
	while(begin != end){
		sum += *begin; // sum=="x"
		begin++;
	}
	return sum;
}
```

Overloaded ++ operator

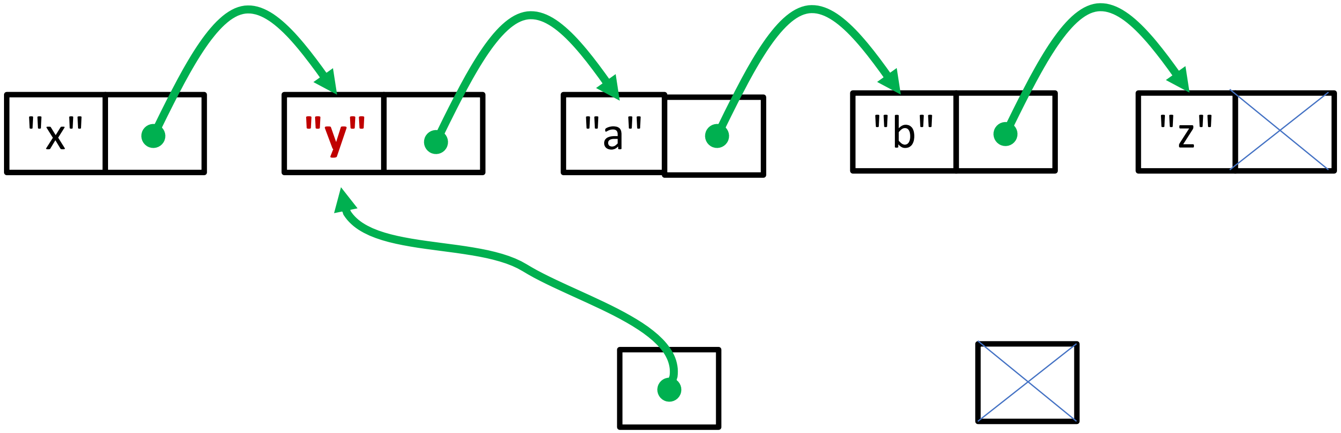
Accumulating : list iterators



```
template<class TIt, class T>
T accumulate(TIt begin, TIt end, T identity)
{
    T sum=identity;
    while(begin != end){
        sum += *begin; // sum=="x"
        begin++;
    }
    return sum;
}
```

Overloaded != operator

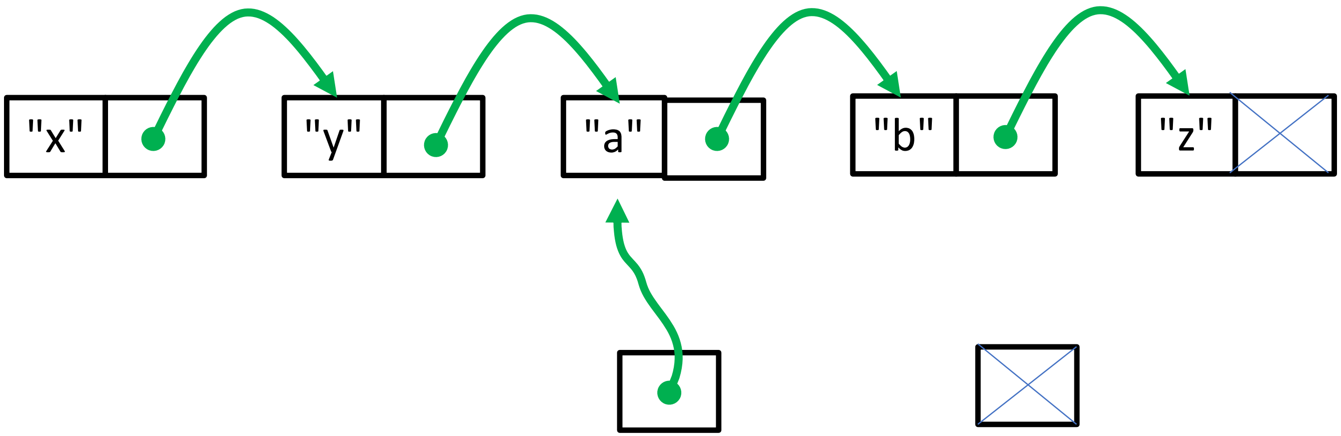
Accumulating : list iterators



```
template<class TIt, class T>
T accumulate(TIt begin, TIt end, T identity)
{
    T sum=identity;
    while(begin != end){
        sum += *begin; // sum=="xy"
        begin++;
    }
    return sum;
}
```

*Overloaded * operator*

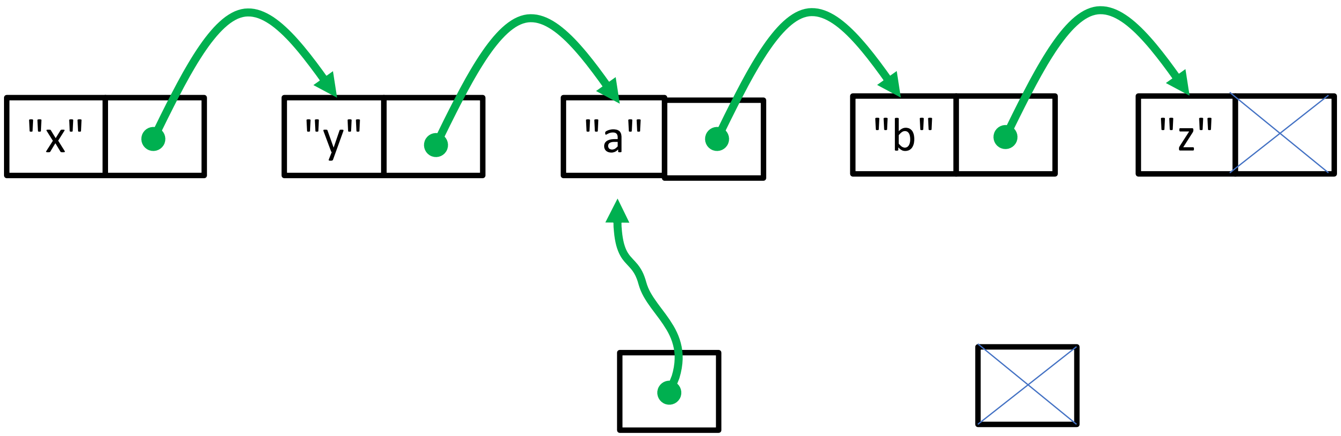
Accumulating : list iterators



```
template<class TIt, class T>
T accumulate(TIt begin, TIt end, T identity)
{
    T sum=identity;
    while(begin != end){
        sum += *begin; // sum=="xy"
        begin++;
    }
    return sum;
}
```

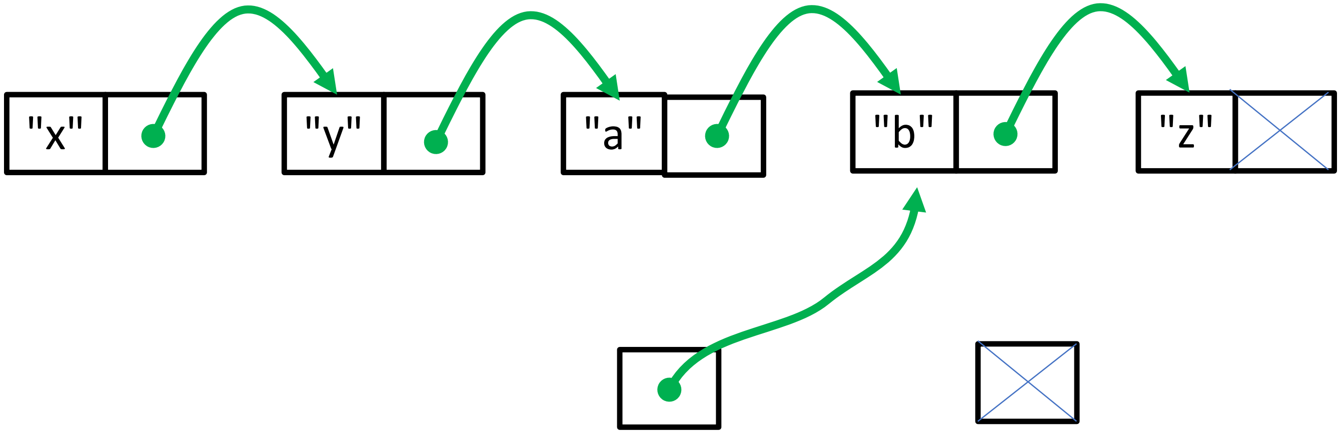
Overloaded ++ operator

Accumulating : list iterators



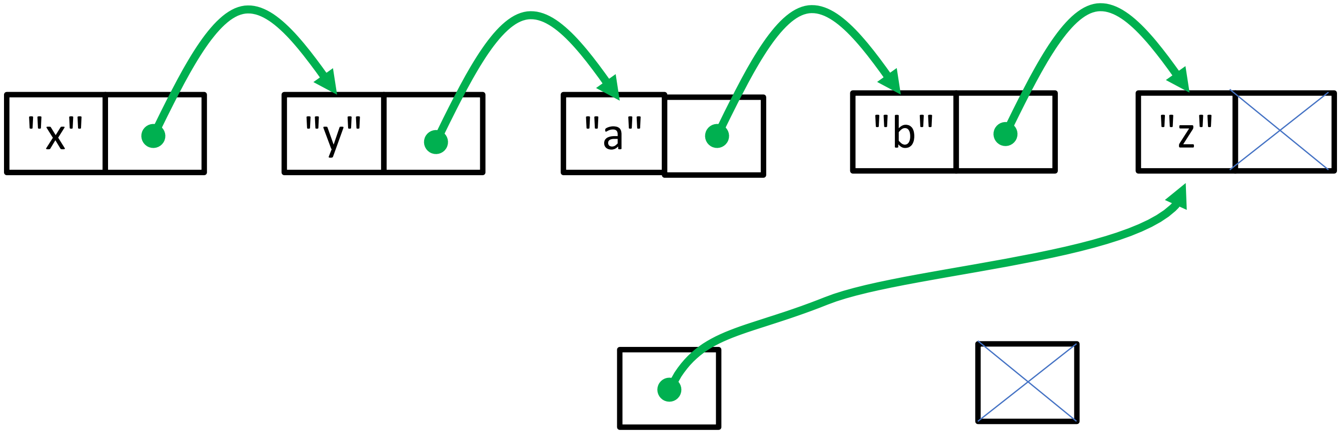
```
template<class TIt, class T>
T accumulate(TIt begin, TIt end, T identity)
{
    T sum=identity;
    while(begin != end){
        sum += *begin; // sum=="xya"
        begin++;
    }
    return sum;
}
```

Accumulating : list iterators



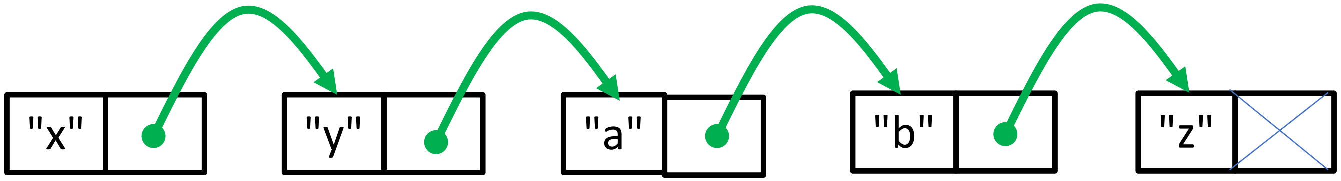
```
template<class TIt, class T>
T accumulate(TIt begin, TIt end, T identity)
{
    T sum=identity;
    while(begin != end){
        sum += *begin; // sum=="xyab"
        begin++;
    }
    return sum;
}
```

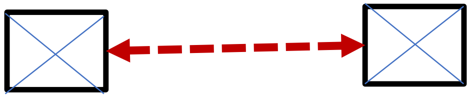
Accumulating : list iterators



```
template<class TIt, class T>
T accumulate(TIt begin, TIt end, T identity)
{
    T sum=identity;
    while(begin != end){
        sum += *begin; // sum=="xyabz"
        begin++;
    }
    return sum;
}
```

Accumulating : list iterators



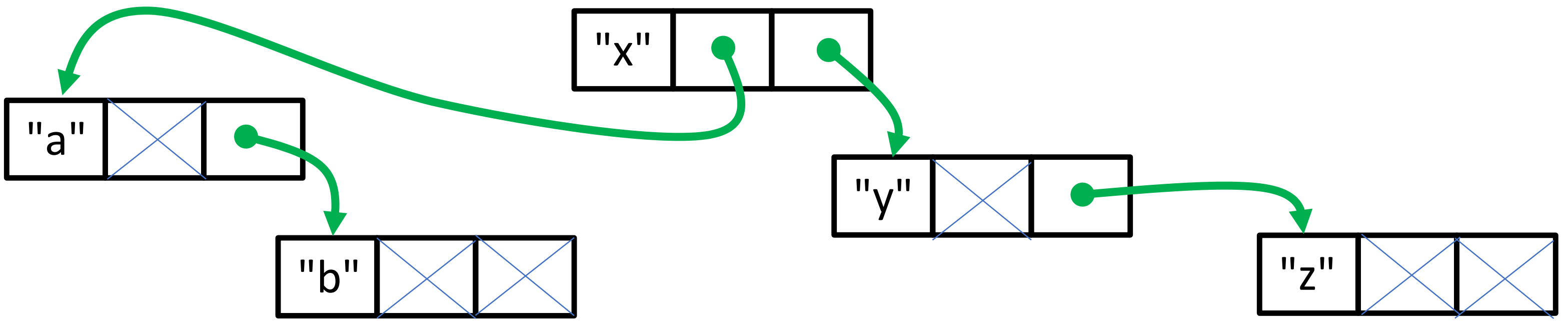


```
template<class TIt, class T>
T accumulate(TIt begin, TIt end, T identity)
{
    T sum=identity;
    while(begin != end){
        sum += *begin; // sum=="xyabz"
        begin++;
    }
    return sum;
}
```

Accumulating over `set<T>`

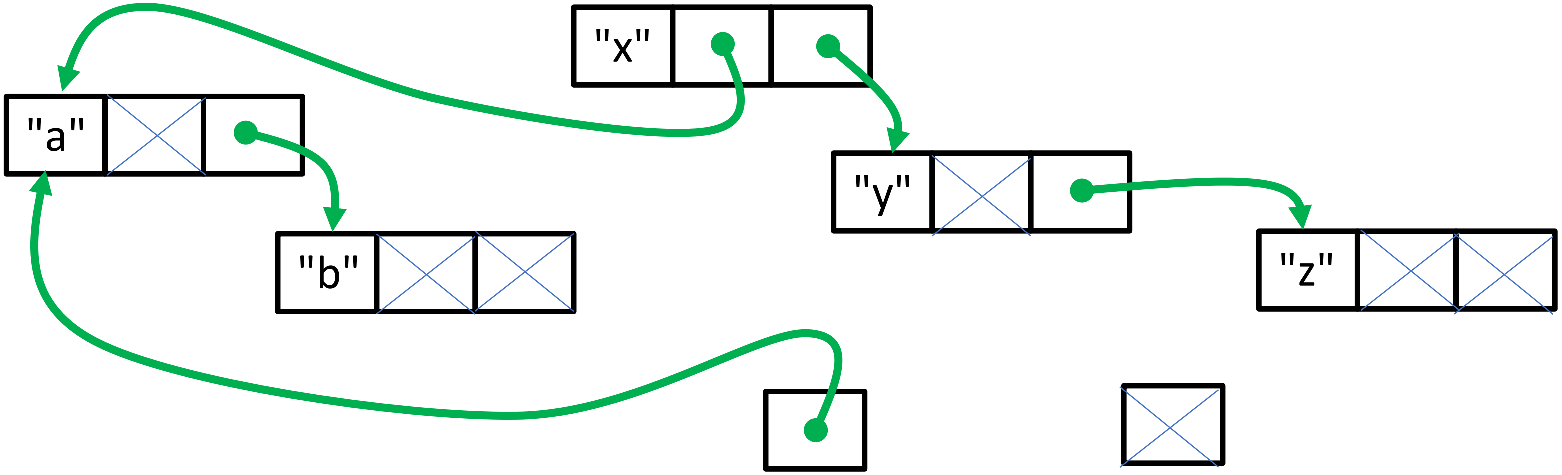
- A set is an ordered collection of items of type T
  - The type T must be Comparable, Assignable, Copyable
  - It behaves very much like a mathematical finite set
- When you insert items in a set:
  - Duplicates are ignored
  - The binary tree is re-balanced
- Sets are a *bit* like `map<T,void>`
  - A mapping of keys to nothing

Accumulating : set iterators



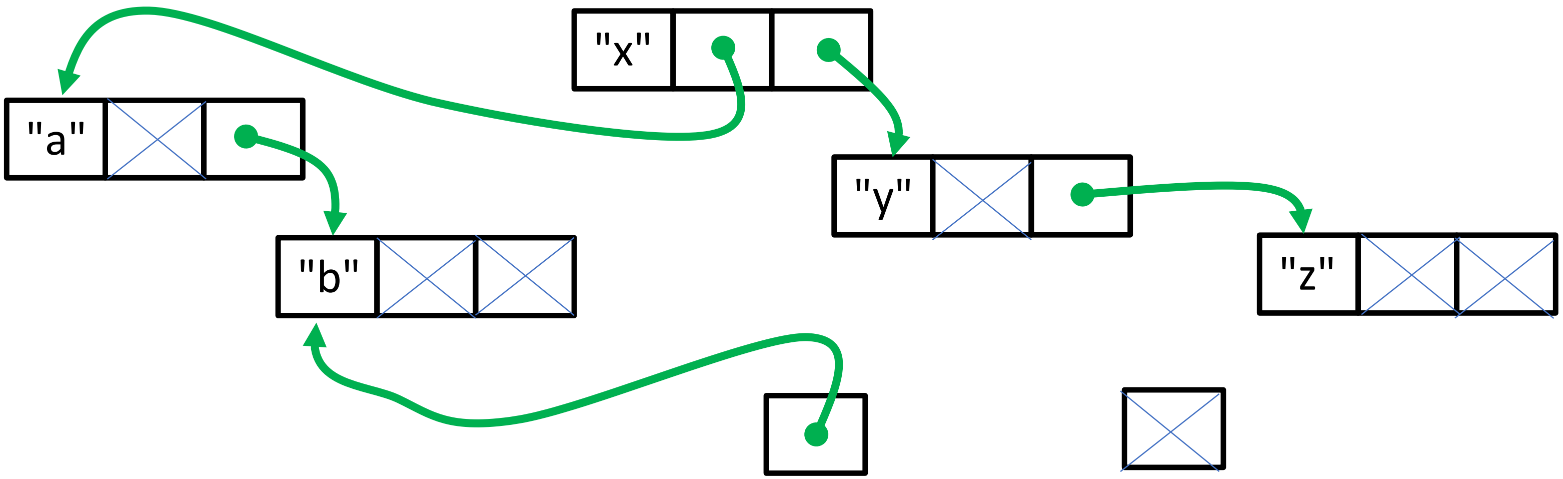
```
int main()
{
    slist<string> l{"x","y","a","b","z"};


    string acc=accumulate(l.begin(), l.end(), string(""));

    cout << acc << endl;  // Prints xyabz
}
```

Accumulating : set iterators



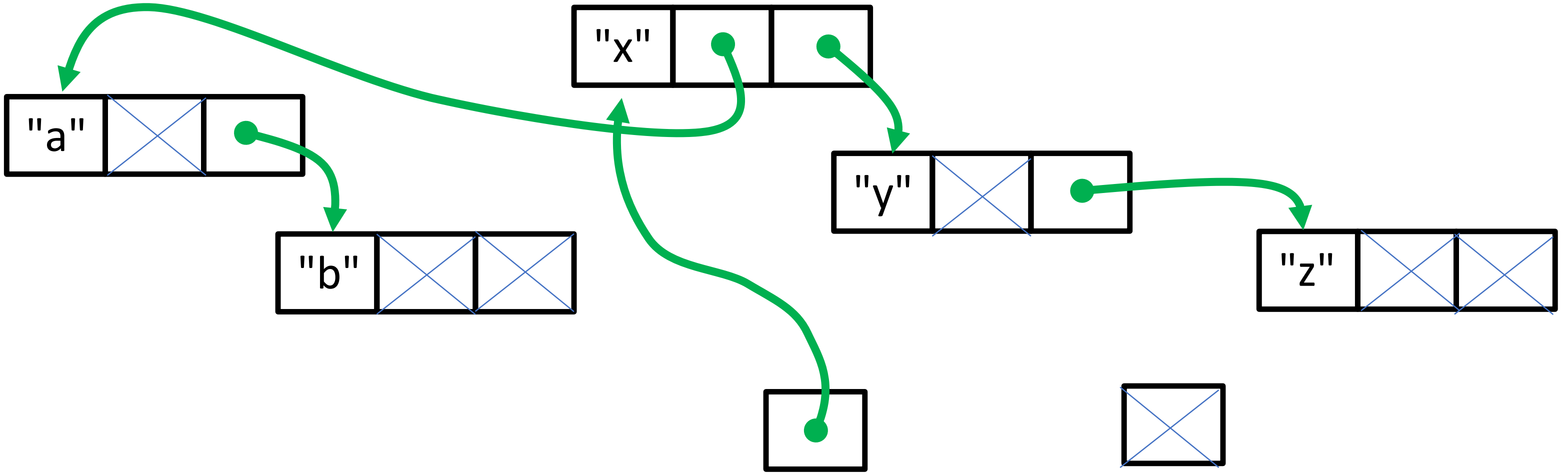
```
template<class TIt, class T>
T accumulate(TIt begin, TIt end, T identity)
{
    T sum=identity;
    while(begin != end){
        sum += *begin; // sum=="a"
        begin++;
    }
    return sum;
}
```

Accumulating : set iterators



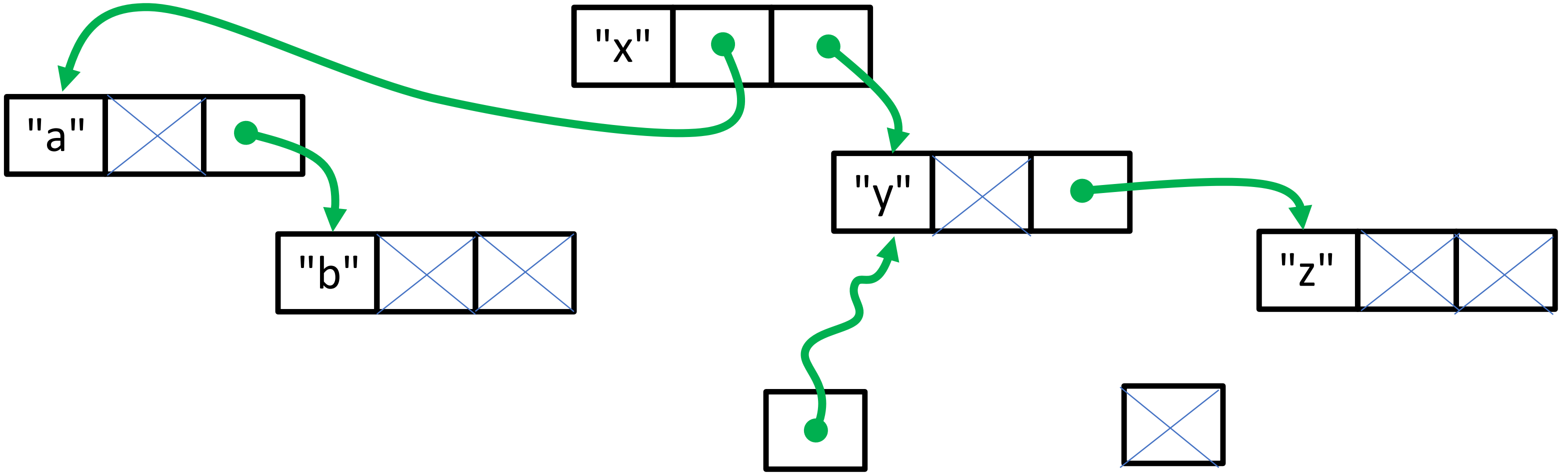
```
template<class TIt, class T>
T accumulate(TIt begin, TIt end, T identity)
{
     T sum=identity;
     while(begin != end){
          sum += *begin; // sum=="b"
          begin++;
     }
     return sum;
}
```

Accumulating : set iterators



```
template<class TIt, class T>
T accumulate(TIt begin, TIt end, T identity)
{
    T sum=identity;
    while(begin != end){
        sum += *begin; // sum=="abx"
        begin++;
    }
    return sum;
}
```

Accumulating : set iterators



```
template<class TIt, class T>
T accumulate(TIt begin, TIt end, T identity)
{
    T sum=identity;
    while(begin != end){
        sum += *begin; // sum=="abxy"
        begin++;
    }
    return sum;
}
```

Iterators are widely used in STL

- The idea is to abstract the idea of pointer
    - Some things "look" like a pointer, but aren't
- The idea of [begin,end) ranges is also common
    - Just like it is widely used with pointers
- Iterators are also used to point at an item
    - Used for finding and deleting elements