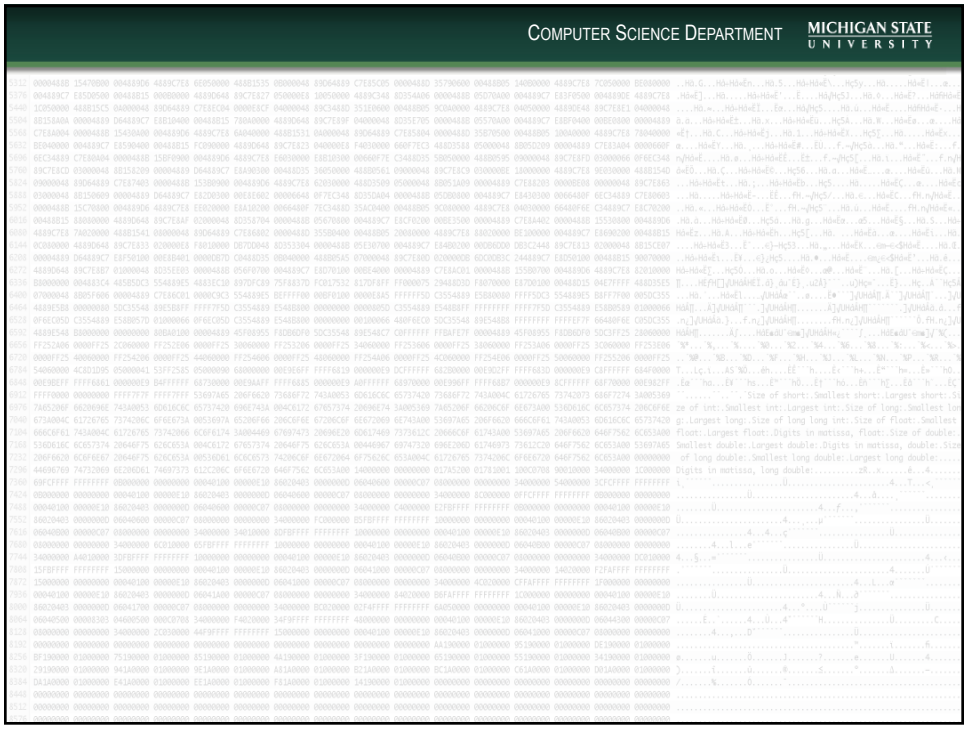
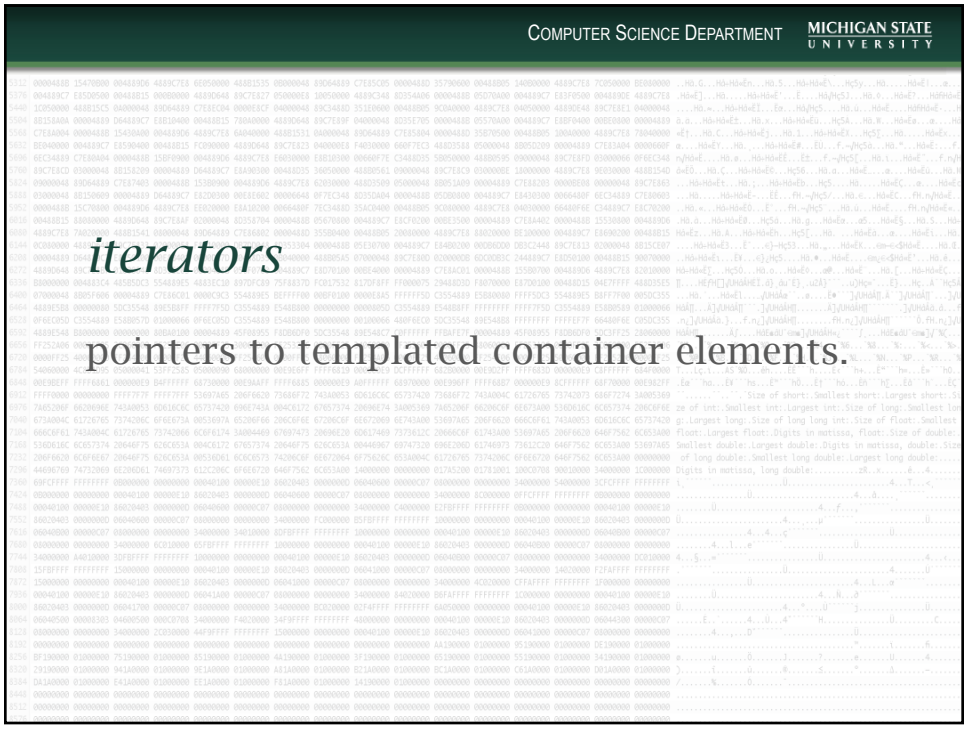

COMPUTER SCIENCE DEPARTMENT
MICHIGAN STATE UNIVERSITY


COMPUTER SCIENCE DEPARTMENT
MICHIGAN STATE UNIVERSITY
iterators
pointers to templated container elements.

in general

Essentially, an iterator is a ***pointer*** to a value in a container.

- does not require an `&`, accomplished with other operators
  - in fact, iterators are objects!!!
- common across all containers
- only way to effectively get access to ***every*** container as not all containers allow `.at` or `[]` (non sequences).

common interface

The result of iterators being common interface to all containers, many of the ***generic algorithms*** depend on iterators:

- generic algorithms work on a container of every type.
- access to how the generic algorithms work is via iterators

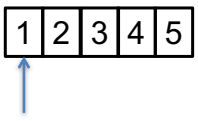
creating an iterator

```
vector<int> v={1,2,3,4,5};
auto v_start = v.begin();
auto v_end = v.end();
string s = "hi mom"
auto s_start = s.begin();
```

`begin()` and `end()` respectively:

- return an iterator to first element
- return an iterator to ***one past*** the last element


```
vector<int>v={1,2,3,4,5};
auto v_start = v.begin();
auto v_last = v.end();
```



`v_start` `v_last`

`v_last` one past the end, type `vector<int>::iterator`

half-open range

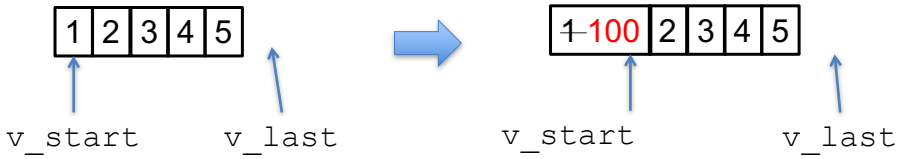
We saw this in Python as well. The reasoning is:

1. Have a stopping point (is your iterator less than the end)
2. For an empty range, begin() == end() so no special testing required

what type

Iterator type is *dependent on the container* they point to (huge surprise):

- `v_start, v_end` are of type `vector<int>::iterator`
- `s_start` is of type `string::iterator`




```
vector<int>::iterator v_start;
vector<int> v{1,2,3,4,5};
v_start = v.begin()
cout << *v_start;  // first element, 1
*v_start = 100;    // assign first to 100
cout << *v_start;  // first element, now 100
```


3 ways to iterate (one more coming)

vector<int> v = {1,2,3,4,5};

```
for(decltype(v.size()) i=0; i<v.size(); i++)
    cout << v[i] << endl;
```

```
for(auto element : v)
    cout << element << endl;
```

```
for(auto ptr = v.begin(); ptr<v.end(); ptr++)
    cout << *ptr << endl;
```

pointer arithmetic

So what does `++ptr` mean?

For some (more on that later) iterators and all pointers, adding one means ***go to the next element***

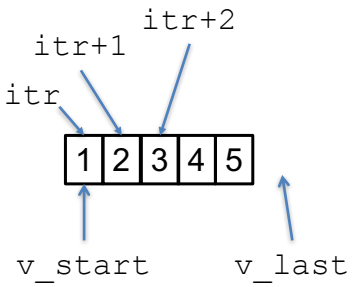
We don't add one to the address (which is what a pointer has as a value), we add enough to the address to get to the next value

how does it know how much to add?

Why types of course!

If it is a `long`, add 8 to the address (8 bytes to a `long`), if it is a `double`, add 8, a vector of `int`, add whatever (the compiler knows!).

Because of the type, pointer arithmetic changes based on that type, adding or subtracting so move to the next element!




```
vector<int>v={1,2,3,4,5};
auto v_start = v.begin();
auto v_last = v.end();

for(auto itr = v.begin();
    itr != v.end();
    ++itr){
    cout << *itr << endl;
    } // of for
```


true for "just pointers"

Pointers (initialized via the & operator) behave the same way using pointer arithmetic

- address is incremented to the "next" element, based on type
- when we get to "good old fashioned arrays", this will be useful.

range for is shortcut for iterator

for range is really a convenience, get's translated into a ptr based loop

```
for (type element : collection){
…
 }
```

```
for (auto pos=collection.begin(), end=collection.end();
      pos!=end; ++pos){
   type element = *pos;
…
}
```


efficiency considerations(1)

Which is more efficient: `++pos` or `pos++`

- `++pos`, since previous value does not need to be stored.

Why `pos != end` instead of `pos < end`?

- not every collection supports < in their iterators (more later). `!=end` is more general but more susceptible to error. Programmer call!

type of the auto element

First, a little background.

`auto` is a great way to declare a variable, but it does have its drawbacks:

- ***it does not preserve*** `const`
- it ***does not preserve*** `&`

You have to add this back yourself


```
for (auto pos=collection.begin(), end=collection.end();
      pos!=end; ++pos){
   auto element = *pos;
…
}
```

copy

```
for (auto element : collection){
…
}
```

What type, `auto element`?

- if it is a standard type, `*pos` derefs and makes a ***copy*** to element
  - change to `element` does ***not*** change the underlying collection. May be what you want

```
for (auto pos=collection.begin(), end=collection.end();
     pos!=end; ++pos){
auto &element = *pos;
…
}
```

```
for (auto &element : collection){
…
}
```

What if it is `auto &element`?

- if we add `&` to the `auto` type, `*pos` derefs and `element` is an ***alias*** to that deref
  - change to `element` ***does*** change the underlying collection. May be what you want


```
for (auto pos=collection.begin(), end=collection.end();
     pos!=end; ++pos){
const auto &element = *pos;
…
}
```

```
for (const auto &element : collection){
…
}
```

- if we add `const &` to the `auto` type, `*pos` derefs and `element` is an alias to that deref
  - no copy but ***cannot*** change the underlying collection. May be what you want

dereference and parens

What is the difference between the code below:

```
vector<int>v={5,4,3,2,1};
auto v_start = v.begin();
cout << *v_start + 1;  // 6
cout << *(v_start + 1) // 4
```

deref, add one to the value

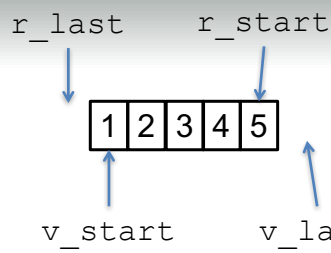
add one to the pointer, deref

* has operator precedence!

Some iterator types

- `begin(), end()`
  - like we have discussed
- `cbegin(), cend()`
  - constant iterators. You can read but you ***cannot write*** to the ptr.
- `rbegin(), rend()`
  - reverse iterators.
- `crbegin, crend()`
  - constant reverse

reverse




```
vector<int>v={1,2,3,4,5};
auto v_start = v.begin();
auto v_last = v.end();
auto r_start = v.rbegin();
auto r_last = v.rend();
```

half-open range is now reversed.

reverse a string

```
string my_str = "hi mother", rev_str="";

 for(auto pos = my_str.rbegin();
          pos < my_str.rend(); ++pos)
   rev_str += *pos;
```

Weirdly, `++pos` means go ***backwards*** one (because it is a reverse iterator, forwards (++) is backwards through container)

general classes of iterators

There are classes of iterators based on the kinds of operations you can perform on them. These restrictions (or allowances) are dictated by their associated containers:

- forward iterators
- bi-directional iterators
- random iterators

forward iterators

given an iterator `itr` on a container, only allow `++itr;`

- cannot go backwards, `--itr;`
- cannot go to a particular index, cannot do pointer math
- no < compare, but != OK
- associated with `forward_list`, ouput iterators, input iterators

bi-directional iterators

For a particular iterator `itr`, can go forward (`++itr`) and backward (`--itr`)

- cannot go to a particular index or do pointer arithmetic
- **cannot** do <, **can** do !=
- associated with maps, sets

random access

can do all of the things lists:

- associated with strings, vectors, lists (sequence containers).