**STANDARD TEMPLATE LIBRARY**

3 things:

* Iterators (kind of generalized pointers)
* Containers (stl classes, vector is a container)
* Generic algorithms

**ITERATORS**

Abstraction of iterators

* Designed to hide details of implementation
* Provide uniform interface across different container class

Each container class has “own” iterator type

* All the containers have their iterators defined as a public inner class inside them
* Similar to how each data type has own pointer type

They behave like pointers

* \* (gives access to data pointed to by p), ++, --, [ ], arithmetic operations, all the operations that are available for pointers work fine, they are overloaded

Each container has functions begin() and end()

* obj.begin() 🡪 returns iterator for 1st item in obj
* obj.end() 🡪 returns “test” value for end (will point to not last element, to end)

cbegin and cend are const iterators (C++11) so you cannot modify that object with a dereferencing operator bc dereferencing operator returns a constant object.

Whatever you can do with pointers, you can do with iterators

* Instead of saying point to the begin of the array (with getting address of beginning of the array), we use begin() function

**Cycling with Iterators**

Recall cycling ability:

for (p = c.begin(); p!=c.end(); ++p) {process\*p}

\*p 🡪 current data item

int arr[10] = {1,7,10,11,-10,…}

int \*pt = arr;

for(int i = 0; i < 10; ++i, ++pt) cout << \*pt;

list<int> iv(10);

int \*pt = &iv[0];

for(int i = 0; i < 10; ++i, ++pt) cout << \*pt;

In green part, can I do ++pt? Because you know how linked lists work. Each node points to next node. So ++pt would be trouble for lists. But I like this notation. I like to use pointers to point to the elements of the linked lists or the array or the vector or the hashset whatever container I have. I like to generalize this idea of using pointers for any kind of containers.

If you use vector instead of list, most of the time ++pt will work bc vectors keep their elements as an array. BUT you are not supposed to use this way. Because this is abstraction, you don’t know implementation of vector. For example if you have vector<bool> and bool pointer for pt, then it will not work for sure. ++pt will point to next byte but each element is 1 bit in vector of bool (vector of bool was rewritten for efficiency).

So iterators have been invented.

**CHECK 19.01**

Iterators for vectors of ints are of type:

std::vector<int>::iterator

Iterators for lists of ints are of type:

std::list<int>::iterator

Vector is in std namespace so need:

using std::vector<int>::iterator;

**Kinds of Iterators**

Different containers 🡪 different iterators

Vector iterators

* Most general form
* All operations work with vector iterators
* Vector container great for iterator examples

**CHECK 19.02, 19.03, 19.05 in order**

There is no hierarchy between containers in C++.

**Iterator Classifications**

Forward iterators:

* ++ works on iterators

Bidirectional iterators:

* Both ++ and -- work on iterator

Random-access iterators:

* ++, --, and random access all work with iterator

These are “kinds” of iterators, not types!

**Constant and Mutable Iterators**

Dereferencing operator’s behaviour dictates

Constant iterator:

* \* produces read-only version of element
* Can use \*p to assign to variable or output, but cannot change element in container
  + E.g., \*p = <anything>; is illegal

Mutable iterator:

* \*p can be assigned value
* Changes corresponding element in container
* i.e.: \*p returns an lvalue

**Reverse Iterators**

To cycle elements in reverse order

* Requires container with bidirectional iterators

Might consider:

iterator p;

for(p = container.end(); p!=container.begin(); --p) cout << \*p;

* But recall end() is just “sentinel”, begin() not!
* end() returns not last element’s pointer, returns END.
* DON’T WRITE THIS KIND OF CODE
* Might work on some systems, but not most

To correctly cycle elements in reverse order:

reverse\_iterator p;

for(rp = container.rbegin(); rp!=container.rend(); ++rp) cout << \*rp;

* rbegin()
  + returns iterator at last element
* rend()
  + returns sentinel “end” marker
* With reverse\_iterator, if you do ++ it goes backwards.

**Compiler Problems**

Some compilers problematic with iterator declarations

Consider our usage:

using std::vector<char>::iterator;

…

iterator p;

Alternatively:

std::vector<char>::iterator p;

And others…

* Try various forms if compiler problematic.
* Write in different way.

**AUTO**

C++11 auto keyword can make your code more readable when it comes to templates and iterators.

Instead of:

vector<int>::iterator p = v.begin();

We can do the same thing more compactly with auto:

auto p = v.begin();

![Diagram

Description automatically generated](data:image/jpeg;base64,/9j/4AAQSkZJRgABAQEAYABgAAD/4RDaRXhpZgAATU0AKgAAAAgABAE7AAIAAAAFAAAISodpAAQAAAABAAAIUJydAAEAAAAKAAAQyOocAAcAAAgMAAAAPgAAAAAc6gAAAAgAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAE1lcnQAAAAFkAMAAgAAABQAABCekAQAAgAAABQAABCykpEAAgAAAAM2OAAAkpIAAgAAAAM2OAAA6hwABwAACAwAAAiSAAAAABzqAAAACAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAMjAyMToxMjoyNiAwMDowNTozOAAyMDIxOjEyOjI2IDAwOjA1OjM4AAAATQBlAHIAdAAAAP/hCxdodHRwOi8vbnMuYWRvYmUuY29tL3hhcC8xLjAvADw/eHBhY2tldCBiZWdpbj0n77u/JyBpZD0nVzVNME1wQ2VoaUh6cmVTek5UY3prYzlkJz8+DQo8eDp4bXBtZXRhIHhtbG5zOng9ImFkb2JlOm5zOm1ldGEvIj48cmRmOlJERiB4bWxuczpyZGY9Imh0dHA6Ly93d3cudzMub3JnLzE5OTkvMDIvMjItcmRmLXN5bnRheC1ucyMiPjxyZGY6RGVzY3JpcHRpb24gcmRmOmFib3V0PSJ1dWlkOmZhZjViZGQ1LWJhM2QtMTFkYS1hZDMxLWQzM2Q3NTE4MmYxYiIgeG1sbnM6ZGM9Imh0dHA6Ly9wdXJsLm9yZy9kYy9lbGVtZW50cy8xLjEvIi8+PHJkZjpEZXNjcmlwdGlvbiByZGY6YWJvdXQ9InV1aWQ6ZmFmNWJkZDUtYmEzZC0xMWRhLWFkMzEtZDMzZDc1MTgyZjFiIiB4bWxuczp4bXA9Imh0dHA6Ly9ucy5hZG9iZS5jb20veGFwLzEuMC8iPjx4bXA6Q3JlYXRlRGF0ZT4yMDIxLTEyLTI2VDAwOjA1OjM4LjY4MzwveG1wOkNyZWF0ZURhdGU+PC9yZGY6RGVzY3JpcHRpb24+PHJkZjpEZXNjcmlwdGlvbiByZGY6YWJvdXQ9InV1aWQ6ZmFmNWJkZDUtYmEzZC0xMWRhLWFkMzEtZDMzZDc1MTgyZjFiIiB4bWxuczpkYz0iaHR0cDovL3B1cmwub3JnL2RjL2VsZW1lbnRzLzEuMS8iPjxkYzpjcmVhdG9yPjxyZGY6U2VxIHhtbG5zOnJkZj0iaHR0cDovL3d3dy53My5vcmcvMTk5OS8wMi8yMi1yZGYtc3ludGF4LW5zIyI+PHJkZjpsaT5NZXJ0PC9yZGY6bGk+PC9yZGY6U2VxPg0KCQkJPC9kYzpjcmVhdG9yPjwvcmRmOkRlc2NyaXB0aW9uPjwvcmRmOlJERj48L3g6eG1wbWV0YT4NCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgPD94cGFja2V0IGVuZD0ndyc/Pv/bAEMABwUFBgUEBwYFBggHBwgKEQsKCQkKFQ8QDBEYFRoZGBUYFxseJyEbHSUdFxgiLiIlKCkrLCsaIC8zLyoyJyorKv/bAEMBBwgICgkKFAsLFCocGBwqKioqKioqKioqKioqKioqKioqKioqKioqKioqKioqKioqKioqKioqKioqKioqKioqKv/AABEIAFQCJAMBIgACEQEDEQH/xAAfAAABBQEBAQEBAQAAAAAAAAAAAQIDBAUGBwgJCgv/xAC1EAACAQMDAgQDBQUEBAAAAX0BAgMABBEFEiExQQYTUWEHInEUMoGRoQgjQrHBFVLR8CQzYnKCCQoWFxgZGiUmJygpKjQ1Njc4OTpDREVGR0hJSlNUVVZXWFlaY2RlZmdoaWpzdHV2d3h5eoOEhYaHiImKkpOUlZaXmJmaoqOkpaanqKmqsrO0tba3uLm6wsPExcbHyMnK0tPU1dbX2Nna4eLj5OXm5+jp6vHy8/T19vf4+fr/xAAfAQADAQEBAQEBAQEBAAAAAAAAAQIDBAUGBwgJCgv/xAC1EQACAQIEBAMEBwUEBAABAncAAQIDEQQFITEGEkFRB2FxEyIygQgUQpGhscEJIzNS8BVictEKFiQ04SXxFxgZGiYnKCkqNTY3ODk6Q0RFRkdISUpTVFVWV1hZWmNkZWZnaGlqc3R1dnd4eXqCg4SFhoeIiYqSk5SVlpeYmZqio6Slpqeoqaqys7S1tre4ubrCw8TFxsfIycrS09TV1tfY2dri4+Tl5ufo6ery8/T19vf4+fr/2gAMAwEAAhEDEQA/APdtb8SNpGq2Wm22jahq13eQzTrHZNAuxImjVixlkQdZkwASevpVb/hKNX/6ETxB/wB/9P8A/kqi+/5KnoX/AGBdS/8AR9jXSUAc3/wlGr/9CJ4g/wC/+n//ACVR/wAJRq//AEIniD/v/p//AMlV0lFAHN/8JRq//QieIP8Av/p//wAlUf8ACUav/wBCJ4g/7/6f/wDJVdJRQBzf/CUav/0IniD/AL/6f/8AJVH/AAlGr/8AQieIP+/+n/8AyVXSUUAc3/wlGr/9CJ4g/wC/+n//ACVR/wAJRq//AEIniD/v/p//AMlV0lFAHN/8JRq//QieIP8Av/p//wAlUf8ACUav/wBCJ4g/7/6f/wDJVdJRQBzf/CUav/0IniD/AL/6f/8AJVaWga1H4g0hb+K1uLT99NA8Fzs8yN4pXicHYzL95G5BIxWlXN+A/wDkXbr/ALDWq/8ApwuKAOkooooAKKKKAMTW/EjaRqtlptto2oatd3kM06x2TQLsSJo1YsZZEHWZMAEnr6VW/wCEo1f/AKETxB/3/wBP/wDkqi+/5KnoX/YF1L/0fY10lAHN/wDCUav/ANCJ4g/7/wCn/wDyVR/wlGr/APQieIP+/wDp/wD8lV0lFAHN/wDCUav/ANCJ4g/7/wCn/wDyVR/wlGr/APQieIP+/wDp/wD8lV0lFAHN/wDCUav/ANCJ4g/7/wCn/wDyVR/wlGr/APQieIP+/wDp/wD8lV0lFAHN/wDCUav/ANCJ4g/7/wCn/wDyVR/wlGr/APQieIP+/wDp/wD8lV0lFAHN/wDCUav/ANCJ4g/7/wCn/wDyVR/wlGr/APQieIP+/wDp/wD8lV0lFAHN/wDCUav/ANCJ4g/7/wCn/wDyVWloGtR+INIW/itbi0/fTQPBc7PMjeKV4nB2My/eRuQSMVpVzfgP/kXbr/sNar/6cLigDpKKKKACiiigDE1vxI2karZabbaNqGrXd5DNOsdk0C7EiaNWLGWRB1mTABJ6+lVv+Eo1f/oRPEH/AH/0/wD+SqL7/kqehf8AYF1L/wBH2NdJQBzf/CUav/0IniD/AL/6f/8AJVH/AAlGr/8AQieIP+/+n/8AyVXSUUAc3/wlGr/9CJ4g/wC/+n//ACVR/wAJRq//AEIniD/v/p//AMlV0lFAHN/8JRq//QieIP8Av/p//wAlUf8ACUav/wBCJ4g/7/6f/wDJVdJRQBzf/CUav/0IniD/AL/6f/8AJVH/AAlGr/8AQieIP+/+n/8AyVXSUUAc3/wlGr/9CJ4g/wC/+n//ACVR/wAJRq//AEIniD/v/p//AMlV0lFAHN/8JRq//QieIP8Av/p//wAlVpaBrUfiDSFv4rW4tP300DwXOzzI3ileJwdjMv3kbkEjFaVc34D/AORduv8AsNar/wCnC4oA6SiiigAooooAxNb8SNpGq2Wm22jahq13eQzTrHZNAuxImjVixlkQdZkwASevpVb/AISjV/8AoRPEH/f/AE//AOSqL7/kqehf9gXUv/R9jXSUAc3/AMJRq/8A0IniD/v/AKf/APJVH/CUav8A9CJ4g/7/AOn/APyVXSUUAc3/AMJRq/8A0IniD/v/AKf/APJVH/CUav8A9CJ4g/7/AOn/APyVXSUUAc3/AMJRq/8A0IniD/v/AKf/APJVH/CUav8A9CJ4g/7/AOn/APyVXSUUAc3/AMJRq/8A0IniD/v/AKf/APJVH/CUav8A9CJ4g/7/AOn/APyVXSUUAc3/AMJRq/8A0IniD/v/AKf/APJVH/CUav8A9CJ4g/7/AOn/APyVXSUUAc3/AMJRq/8A0IniD/v/AKf/APJVWdE8SNq+q3um3OjahpN3ZwwztHetA29JWkVSpikcdYXyCQenrW3XN2P/ACVPXf8AsC6b/wCj76gDpKKKKAObvv8Akqehf9gXUv8A0fY10lc3ff8AJU9C/wCwLqX/AKPsa6SgAooooAhvLZL2yltpmlVJFKsYZnicD2dCGU+4INeD+EvG+vaD8K9D1SC5fVdR1jxANOlk1e6nnAQs4G3L/IRgcgfUGveLu1jvbOW2maVY5VKsYZnicD2dCGU+4INcxafC/wAI2WnW1hb6bMLS1vEvoIXvrh1inXOHUM5x1OR0PcGuqhUpwTVRXu1+G4parT+tH+pxNz8V/ElvprQiDSm1GHxX/YEk3kSCF0I4kCeZkN7biKfc/FjX7GVrCe30ya9tvFEOjTzxxSLFJDICd6qZCVfjoWYD3ro/E3wn0nW7eKLTZn0rdra6zdvGZHa4lAIbB3gxk56qeMcCtVvhv4UbQ30ltLJtXvPtzMbmXzmuM580zbvMLe+72rdVcMkm4/17v/B08+gpJ308/wD2636fceZeK/HXijV4yllfQaWNP8bJpKm2SUNNGPu+YRINy9dy8BuPu459xgEy28YunSSYKBI8aFFZsckKSSBntk/U1zH/AArPwn/ZM+nNpsrwXF6L+RnvZ2lNwOkvml94b3DevrXTwQrbW8cEZcpGoRTJIzsQBjlmJJPuSSawrVKcoKMFt/kv1uN3cr/1u3+ViSiiiuUYV5Bq2q/EDS/h7fSfDzR7O+f+1tW82ZpS9xEPtt3kxwEBWIwhHzMSTjyz1r1+ub8B/wDIu3X/AGGtV/8AThcUAcj8L9b+IOofDuxudQ0vT767ea6E82parLbT7xcyBlaIWzBNpBUKDgBQMDoOu+3eN/8AoXvD/wD4Pp//AJDrpKKAOb+3eN/+he8P/wDg+n/+Q6Pt3jf/AKF7w/8A+D6f/wCQ66SigDjYJdVm+JOitrlnZ2dx/ZOpBY7O7a4Qr51jglmjjIOc8YPQc84HZVzd9/yVPQv+wLqX/o+xrpKACvL/AIt39/beJvBVpZS6oYLy+ljuLXTb1raS5XauF3CRB+bD616hWZqfhzStZ1LTb/UrXzrnS5TNZyeYy+U5GCcAgHp3zWtGahUUpbIHs0cXP4k1LStb8O+E9BsJ9Nu9WNxNJL4guGv2gjjXOPluGLFjjA8zAHbnjC/4W14n1K28L22iafpKanq13eWFyLwy+SJoAAHQqdwUk5wQT2z/ABV6drnhnSfETWr6rbO81nIZLaeCeSCaFiMHbJGysAR1AODxnoKqw+BPDVv/AGL9n0tIv7CLnTwkjqIS4+YnB+cnGSWyScnqTXRCrRsnON31/Hz9NPK4mn0/rR/rqcHqvxK8X6NqGs6Bc2Ojza/Elm2mLEsgiuPMUmUkFskLsfB46c1KnxbnutP1zV1ls7HR9PsbJ4bh7OS4eS6nRZDGEEibsBguMrgsCTgGu+u/COhX3iqz8SXWnpJq9lGYoLkuwKKc8bQdp+83JBxmqNv8OPCVr4cvdBt9GiTTL6bz7iASP80nB3Bt25SNoxgjGOMUKrQ5fejrp+l/wX3t9BWd9Nv6/X8Dgrv4seJ9N8O+M5NQ0u0i1Pw6LOSETQGNZFnI4eNJpMEDJ4kPUZAwRWlN4y8bDVdD0C6h0G3v9bS4uvtESTTxQWqRhlXaWQtISTk5A9vTqG+GnhSSz1W2m06WaPWFiW/M17PI9wIjlMuzlsj1Bye+av6t4P0TWxYHULWTzNNBFpNBdSwSwgrtIEkbK2COCM80Sq0HtH8P7tu/82voGv8AXy/4P4HIfs//APJHNNz18+4/9HNXpVZnh/w7pfhbRotK0G1+y2UTMyReY74LEk8sSepPetOscRUVWrKa2bCKsgryDVtV+IGl/D2+k+Hmj2d8/wDa2rebM0pe4iH227yY4CArEYQj5mJJx5Z616/XN+A/+Rduv+w1qv8A6cLisCjkfhfrfxB1D4d2NzqGl6ffXbzXQnm1LVZbafeLmQMrRC2YJtIKhQcAKBgdB1327xv/ANC94f8A/B9P/wDIddJRQBzf27xv/wBC94f/APB9P/8AIdH27xv/ANC94f8A/B9P/wDIddJRQBxsEuqzfEnRW1yzs7O4/snUgsdndtcIV86xwSzRxkHOeMHoOecDb8WTS23gvWp7eR4po7Cd0kjYqyMI2III6EHvVK+/5KnoX/YF1L/0fY1vXtnBqFhPZXieZb3EbRSpkjcrDBGRyOD2qZpyi0iotKSbPBvh9rOrx+IvA0upXeuW9lrWlTpO2qai13FqU4UMDGnmP5RxzkhMgEY65X4f+M79PDPhbw34U0zStJfWDqE80kiTzwwxxM2dqNNvLMQOr4Hp6eraR8OPC+h3tldWFhMZdPjaKz+0309wtqrYBEayuypkDGVAOKjX4YeEY9N0+xt9Mlt4tNeWSze3vZ4pYTJnzMSq4fDZORnFVLW/nf8ANv8AX9CY6Lz0/K39ff5Hi/w1mt4rH4VRz6dbXM015qnl3MjSh7cqScoFcKc99wb2xXSab8YfGP8AwrK88barpmiSWCk21rBamcStcGYRhnByBGAc4BJJHUZ49J074c+FdJXRxp+l+UNEeaSw/wBJlbyWl/1h5Y7s/wC1nHbFT2fgTw1YeEZfDFvpUR0aYsZLSV2lVix3E5Yk5zyDng4xjFNu9/66/wCQ9L/13f8Amjz+9+LPiTQtG1651TQzdLaLbjTr46XdadBO8rBCrpPlhsY5yD8w9KT4m3njzStP8OtNr+mwG412zhD6baTwFi+7KuPPO6MYGVyN2f4cc95H8PvDi2N3Z3FpcX1tdwC3li1C/uLtfLByFXzXbZg8jbgggHsKhm+Gnha505rK7s7u5iaSKUNcalcyyRtFnyykjSF027mxtI6mjT8V91/6/rUWv4P8v8/66LzbVfGh8D+NPiBqMWkWE+rW8OmwfakMsa3EsqABpFaRlVFPOFAOBgk/erR8U/FfxJ4Ss/FNleQaVeatosNpdQXEUEkcEsczqpVozIWDKScEPgjHAxg9/J8P/DE/9qm50z7SdYgit783E8kpnSNdqZLMcMAB8ww2RnOeahf4aeE5tG1DTLjTJLi31No2vGnvJ5Jp/LxsDSs5kwu0YG7A545NJbJf1/wO/wCBV1zX/rp/X4+nHt8VdT0w+L7bxHPptnNokdnLbXNvZSzK/nruERi81S7Z+UMGQHOSFAIrl/HPjTXPEPwx+IOieJdPjtLrSBYOrLEIWdJpEYBoxLKARjqJDwRwCDXrt78O/C2ovqr3ulLM2rpCl6Wmk/eCEAR4+b5SuByuDUDfDDwlJZatazabLPHrIhF+Zr64ke48o5j3Ozlsj1Bye+aen5foKL5Wn2t+l/1OksP+Qbbf9cU/kK8o1bVfiBpfw9vpPh5o9nfP/a2rebM0pe4iH227yY4CArEYQj5mJJx5Z6167GixRrHGMKgCqPQCud8B/wDIu3X/AGGtV/8AThcUSd22RFWikcj8L9b+IOofDuxudQ0vT767ea6E82parLbT7xcyBlaIWzBNpBUKDgBQMDoOu+3eN/8AoXvD/wD4Pp//AJDrpKKRRzf27xv/ANC94f8A/B9P/wDIdH27xv8A9C94f/8AB9P/APIddJRQB4p8YNV8Q2GlG+uRFo+pRaNe+S+l6hJKVU3mmjPmGONgTlhgDp35wGeBvFV0viXxNeHUdQ0zRdJ0iP7Vb69cyXkkd1jd56xmRn8vGeFYBsrjtjufFmhab4m8caVpGt232mxudF1ASxb2Tdi4sWHKkEcgdDWleeAPDN/qv9pXWmB7o2RsHYTSKJYCMeW6hgrjB/iBPA9Bjro1qcKUoSWrv+S/ry3B62/rt/X/AA5wNl8WPEe/U0vdPt9q+G31zT55bT7OZApwN0a3EuVOQfvIwweOQaZZfGDXbSIT67Y6fPHN4X/t2FLNXjKuG2+WzMzZB65AGM4+bGT3Nt8M/ClqZTHp0zmawbTXaa+nlP2Y9YgWc4HHGMY7YqxD4A8MQSQumlKTBpp0pA8sjqLU8+WQWIP1OT71pKrhne0f618/QSv1/r4f+D955EnirVNK+JqeK/EiWl1cr4KN99nsYmhUAuGEeWd8nJ5b/wAd456V/id4j0u1T+2IdLuJ7/w3JrVm1rDIiwyIhYxSAyNvXBHzKV6EY5yOy0v4beFdIvPtVppsjyCzNgPtV5PcqLc/8sgsrsoT2xilsvhx4WsLa5gg012jurP7A/nXc0rLb8/ukZ3JjTk8IQOnoKqpXoS+z07evnpuvWwoprV+X/tv+T+889i+LHi9lhSS20QSXfhltdjdYZsQ7AxKEeZ8+7bxyu3d/Ft+b0nw54pTVfh3Y+J9RVLVJbAXdwFPyx4XLYz24NRD4d+Fg0JGl8w6Y2kx/wCkS/LakEGP73ufm+971ePhPRj4R/4Rj7Iw0fyPs/2ZZ5B+7/u7w279eaxrVKE1aEba/hr+lvuCKaav8/uX63+88c+Ffj67l8ab9Z8QtqUfiKyuL1LD7X5x0+WOV2EKpk7AYuccdOnFdfpHj7xFefD+Tx7dxaWNHWC4uRpkUUn2gIm8IPP3lSxZRn92AASe2K7GbwdoU66Kr2OBoRU6dsmdPIwoUDhhuGABhsg96rWfw/8ADNhcNJaacyRM0jfZPtMptQZFKvi3LeUMhmHC9zWlStQm21G39P8AS2u6t1uCTvd+r/r+t/LXldT8d+KPDnw2HjDVk0i9jubSKSC0tIJYzDLKyBAzl33qAxJwFJIAHXiE/EDxtHpfiE2/hqXVrmyt4pdPuI9IurKO4LEB1MUxLsUJJwp+YDtXW2/w58L29jNZDTpJrOW3Nsba5vJp4kiLBtqI7lU5VSNoGCBjpUsHgbRbaKRIX1VfMRYy51q8LqinIVXMu5BnspGe9T7Whr7vXt07b/8AD3BJ6f1/X9fLj5/iNrzeGHuNBjj13UodQhtr6GDQ7mGTTUYZYyWzSmSQjn7rAe/FaHgDX4vEfjTW76HU7bU8aXYxPNbWclqqstxfAqYpGZlI9z71v2/gLQLSzlt7OK+thNObiaWDU7mOaeQjGZJVkDv9GYgVneFdC0zw34+1nTtEtEtLWPR9PYIpLEsZ74lmYklifUkmoqTpOLUVr8v69LfMST0/rudpRRRXKWcl4ivl0jx9ompXNpqE1oumX8DSWWnz3Wx3ltGUMIkYrkRvgkY+U1Z/4TzSP+fPxB/4Tmof/GKi8QHUrzxpo+kWGt3mk282n3t1M1nFAzyNHJbKgJljcAATP0A7elS/8Ivq/wD0PfiD/vxp/wD8i0AH/CeaR/z5+IP/AAnNQ/8AjFH/AAnmkf8APn4g/wDCc1D/AOMUf8Ivq/8A0PfiD/vxp/8A8i0f8Ivq/wD0PfiD/vxp/wD8i0AH/CeaR/z5+IP/AAnNQ/8AjFH/AAnmkf8APn4g/wDCc1D/AOMUf8Ivq/8A0PfiD/vxp/8A8i0f8Ivq/wD0PfiD/vxp/wD8i0AH/CeaR/z5+IP/AAnNQ/8AjFH/AAnmkf8APn4g/wDCc1D/AOMUf8Ivq/8A0PfiD/vxp/8A8i0f8Ivq/wD0PfiD/vxp/wD8i0AH/CeaR/z5+IP/AAnNQ/8AjFH/AAnmkf8APn4g/wDCc1D/AOMUf8Ivq/8A0PfiD/vxp/8A8i0f8Ivq/wD0PfiD/vxp/wD8i0AH/CeaR/z5+IP/AAnNQ/8AjFH/AAnmkf8APn4g/wDCc1D/AOMUf8Ivq/8A0PfiD/vxp/8A8i0f8Ivq/wD0PfiD/vxp/wD8i0AH/CeaR/z5+IP/AAnNQ/8AjFHgISf8Iu8ktvcW/n6nqM6R3MDwybJL2Z0JRwGXKspwQDg0f8Ivq/8A0PfiD/vxp/8A8i1J4Ju7288Ns2p3kl9cQ6hfWv2iVEV5Fhu5YkLBFVc7UXOAKAOgooooAKKKKAOS8RXy6R4+0TUrm01Ca0XTL+BpLLT57rY7y2jKGESMVyI3wSMfKas/8J5pH/Pn4g/8JzUP/jFdJRQBzf8Awnmkf8+fiD/wnNQ/+MUf8J5pH/Pn4g/8JzUP/jFdJRQBzf8Awnmkf8+fiD/wnNQ/+MUf8J5pH/Pn4g/8JzUP/jFdJRQBzf8Awnmkf8+fiD/wnNQ/+MUf8J5pH/Pn4g/8JzUP/jFdJRQBzf8Awnmkf8+fiD/wnNQ/+MUf8J5pH/Pn4g/8JzUP/jFdJRQBzf8Awnmkf8+fiD/wnNQ/+MUf8J5pH/Pn4g/8JzUP/jFdJRQBzf8Awnmkf8+fiD/wnNQ/+MUeAhJ/wi7yS29xb+fqeozpHcwPDJskvZnQlHAZcqynBAODXSUUAFFFFABRRRQByXiK+XSPH2ialc2moTWi6ZfwNJZafPdbHeW0ZQwiRiuRG+CRj5TVn/hPNI/58/EH/hOah/8AGK6SigDm/wDhPNI/58/EH/hOah/8Yo/4TzSP+fPxB/4Tmof/ABiukooA5v8A4TzSP+fPxB/4Tmof/GKP+E80j/nz8Qf+E5qH/wAYrpKKAOb/AOE80j/nz8Qf+E5qH/xij/hPNI/58/EH/hOah/8AGK6SigDm/wDhPNI/58/EH/hOah/8Yo/4TzSP+fPxB/4Tmof/ABiukooA5v8A4TzSP+fPxB/4Tmof/GKP+E80j/nz8Qf+E5qH/wAYrpKKAOb/AOE80j/nz8Qf+E5qH/xijwEJP+EXeSW3uLfz9T1GdI7mB4ZNkl7M6Eo4DLlWU4IBwa6SigAooooAKKKKAOS8RXy6R4+0TUrm01Ca0XTL+BpLLT57rY7y2jKGESMVyI3wSMfKas/8J5pH/Pn4g/8ACc1D/wCMV0lFAHN/8J5pH/Pn4g/8JzUP/jFH/CeaR/z5+IP/AAnNQ/8AjFdJRQBzf/CeaR/z5+IP/Cc1D/4xR/wnmkf8+fiD/wAJzUP/AIxXSUUAc3/wnmkf8+fiD/wnNQ/+MUf8J5pH/Pn4g/8ACc1D/wCMV0lFAHN/8J5pH/Pn4g/8JzUP/jFH/CeaR/z5+IP/AAnNQ/8AjFdJRQBzf/CeaR/z5+IP/Cc1D/4xR/wnmkf8+fiD/wAJzUP/AIxXSUUAc3/wnmkf8+fiD/wnNQ/+MVW8O3y6v4+1vUra01CG0bTLCBZL3T57Xe6S3bMFEqKWwJEyQMfMK62igAooooA5u+/5KnoX/YF1L/0fY10lc3ff8lT0L/sC6l/6Psa6SgAooooAxNV8X6Lo2qR6bd3E0t/JEZhaWdpNdSrGCBvZIlZlXJAyQAe1XdH1nTvEGkwano13Hd2c67o5Yz19QQeQR0IOCDwRXCQxXXhH4veIdZ1Sx1C707W7WA2t1Y2ct15TRDa0TLGGZc53AkbTjrkEVw154T8QReErfWJNHvVW68Zprf8AZ0cLSXFtasTy0a5IboSoyRnnvjthh6cktbXtr0u918vXp5oUm1e39aX/AD0PcNa8Q6X4ditJNYufs6Xl0lpAfLZ98r52r8oOM4PJ4960q+cdQ8L3up6XqepTeGL+5i/4TkXwSbS5POeyb77CJk3lD8uQBzjkcVW8R6fZ6l4x8f2Wk6Dc3d/Jb6cujx22nSBrRtiYbIUfZ8DH3tvAI9RWkcHGVlza9dP8Pn5/gDdn/Xdr/gn0vWbaeIdLvtf1DRbW58zUNNWNrqHy2HliQZT5iMHI9CfevBvGeheIIZvH1rcaPql7da1Y6YLaW0sZZ455I9nm/OilVIIbrj+Vdn4S8PRaT8b9evLzw/JCt5a2smn3o05jGr+URN+9C7Y2JznJBY+uaj6rBQcnK+l1+H+b+75CctP68v8AP8D1iiiiuEoK5vwH/wAi7df9hrVf/ThcV0lc34D/AORduv8AsNar/wCnC4oA6SiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigDm77/kqehf9gXUv/R9jXSVz+uaHqt5r+navoep2djcWdrcWrLeWLXKSLM0LEgLLGQQYB3P3jUf2Hxv/wBDD4f/APBDP/8AJlAHSUVzf2Hxv/0MPh//AMEM/wD8mUfYfG//AEMPh/8A8EM//wAmUAdJRXN/YfG//Qw+H/8AwQz/APyZR9h8b/8AQw+H/wDwQz//ACZQB0lVYNMsLa/uL62sreK7ugouLiOJVkm2jC7mAy2B0z0rF+w+N/8AoYfD/wD4IZ//AJMo+w+N/wDoYfD/AP4IZ/8A5Mp3aA6Siub+w+N/+hh8P/8Aghn/APkyj7D43/6GHw//AOCGf/5MpAdJRXN/YfG//Qw+H/8AwQz/APyZR9h8b/8AQw+H/wDwQz//ACZQB0lc34D/AORduv8AsNar/wCnC4o+w+N/+hh8P/8Aghn/APkyrvhjRp9B0P7HeXcd5cPdXN1LNFAYUZpp3mIVCzEAGTAyx6UAa9FFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAf/Z)

**CONTAINERS**

Container classes in STL:

* Different kinds of data structures like lists, queues, stacks

Each is template class with parameter for particular data type to be stored

* E.g., lists of ints, doubles or myClass types

Each has own iterators

* One might have bidirectional, another might just have forward iterators

But all operators and members have same meaning

**Sequential Containers**

Arranges list data

* 1st element, next element, … to last element

Vector is sequential container

* There is a sequence in it, you talk about first element, next element…

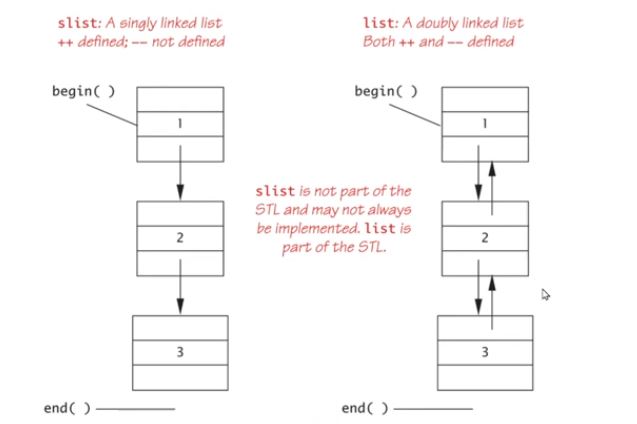
Linked list is sequential container

* Earlier linked lists were “singly linked lists”
  + One link per node

STL has no “singly linked list” (C++11 has it)

* Only “double linked list” : template class list

Sets, unordered sets, maps are not sequential containers



**Container Adapters stack and queue**

Container adapters are template classes

* Implemented “on top of” other classes
* These classes use somebody else’s capabilities to implement themselves

Stack has just these essential functions: “push”, “pop” and “empty”

* Stack use vector’s capabilities to implement itself

Example:

* stack template class by default implemented on top of deque (double ended queue) template class (deque is data structure in between list and vector)
  + Buried in stack’s implementation is deque where all data resides

Others:

* queue, priority\_queue

**Specifying Container Adapters**

Adapter template classes have “default” containers underneath

* But can specify different underlying container
* Examples:
  + stack template class 🡪 any sequence container
  + priority\_queue 🡪 default is vector, could be others

Implementing example:

stack<int, vector<int> >

* Makes vector underlying container for stack

**CHECK 19.10**

template<class T, class R = vector<T> >

class stack{

private:

R data\_;

public:

stack(){};

void push(const T& e) {data\_.push\_back(e);}

T pop() {return data\_.pop\_back();}

bool empty() const {return data\_.size() == 0;}

};

stack<int> s1;

stack<int, list<int> > s2;

stack<int, Array<int> > s3; 🡪 NOT VALID BC ARRAY IS STATIC, IT

DOESN’T HAVE push\_back(), pop\_back()

stack<int, Money> s3; 🡪 NOT VALID BC MONEY DOESN’T HAVE size(),

pop\_back() and push\_back() operations

Stack is implemented as adapter class. Adapter of vector by default.

Be careful there is no isA relationship between stack and vector. There is hasA relationship.

I don’t have iterator for stack class.

**Associative Containers**

Associative container: simple database

Store data

* Each data item has key

Example:

* key: employee’s SSN
* data: employee’s record as struct
  + Items retrieved based on key

Instead of a[7] = “ABC” we say a[“Ahmet”] = “ABC”

KEY VALUE

**set Template Class**

Simplest associative container possible

Stores elements without repetition

1st insertion places element in set

Each element is own key

* You just have the key, you don’t have the value.

Capabilities:

* Add elements
* Delete elements
* Ask if element is in set

Designed to be efficient

* Stores values in sorted order
* Can specify order:

set<T, *Ordering*> s;

* + Ordering is well-behaved ordering relation that returns bool
  + None specified: use < relational operator

set is specialization of map

* You just ignore the first element of the templated definition.
* Most general version of the set is map.

**CHECK 19.12**

**map Template Class**

A function given as set of ordered pairs

* For each value first, at most one value second in map

Example map declaration:

map<string, int> numberMap;

You are saying that my key is string, my value is integer.

You can also say: map<string, Money> numberMap;

Can use [ ] notation to access the map

* For both storage and retrieval

Stores in sorted order, like set

* Second value can have no ordering impact

**CHECK 19.14**

Access element (mapObj[k])

* If *k* matches the key of an element in the container, the function returns a reference to its mapped value.
* If *k* does not match the key of any element in the container, the function inserts a new element with that key and returns a reference to its mapped value. Notice that this always increases the container size by one, even if no mapped value is assigned to the element (the element is constructed using its default constructor).

Data structure behind map is hash map. Next semester you will see this.

Range for loop works for all kind of containers. As long as they define an iterator.

for (auto i : mySet)

cout << i << “\n”;

This code works for every containers that have iterators.

**Use Initialization, ranged for, and auto with Containers**

C++11’s initialization, ranged for, and auto features make it easier to work with containers.

Consider:

map<int, string> personIDs = { {1, “Walt”}, {2, “Kenrick”}};

set<string> colors = {“red”, “green”, “blue”};

We can easily iterate through each with:

for (auto p : personIDs)

cout << p.first << “ “ << p.second << endl;

for (auto p : colors)

cout << p << “ “;

for(int& i : mySet) cout << i ; 🡪 ERROR

Invalid initialization of reference of type ‘int&’ from expression of type ‘const int’

So you cannot get non-const reference to elements of set.

For the vectors, these work.

* for(int& i : myVector) cout << i;
* for(int& i : myVector) cout << ++i;

**Efficiency**

STL designed with efficiency as important consideration

* Strives to be optimally efficient

Example: set, map elements stored in sorted order for fast searches

**ALGORITHMS**

C++11 STL specifies lots of algorithms for you.

They are all templated functions.

All the algorithms defined in the STL are very efficiently implemented.

*For example sort:*

* Takes 2 random access iterators 🡪 beginning and end of the container
* Sort between 2 iterators
* If I don’t want to sort the integers with respect to natural ordering, I can add another function pointer (name of the function) as parameter to sort function.

bool myFunction (int i , int j) { return i<j; }

struct myclass {

bool operator() (int i,int j) { return (i<j);}

} myobject;

. . .

int myints[] = {32, 71, 12, 45, 26, 80, 53, 33};

vector<int> myvector(myints, myints+8); 🡪

//32, 71, 12, 45, 26, 80, 53, 33

// using default comparison (operator <):

sort (myvector.begin(), myvector.begin()+4);

//(12 32 45 71)26 80 53 33

// using function as comp

sort (myvector.begin()+4, myvector.end(), myfunction);

// 12 32 45 71(26 33 53 80)

// using object as comp

sort (myvector.begin(), myvector.end(), myobject);

//(12 26 32 33 45 53 71 80)

*For example for\_each:*

template<class InputIterator, class Function>

Function for\_each(InputIterator first, InputIterator last, Function fn)

{

while (first!=last) {

fn (\*first);

++first;

}

return fn; // or, since C++11: return move(fn);

}

//Applies function fn to each elements in the range [first, last).

void myfunction (int i) { // function:

std::cout << ' ' << i;

}

struct myclass { // function object type:

void operator() (int i) {std::cout << ' ' << i;}

} myobject;

. . .

std::vector<int> myvector;

myvector.push\_back(10);

myvector.push\_back(20);

myvector.push\_back(30);

std::cout << "myvector contains:";

for\_each (myvector.begin(), myvector.end(), myfunction);

std::cout << '\n';

// or:

std::cout << "myvector contains:";

for\_each (myvector.begin(), myvector.end(), myobject);

std::cout << '\n';

**CHECK 19.17**

**Modifying Sequence Algorithms**

STL functions that change container contents

Recall: adding/removing elements from containers can affect other iterators

* list, slist guarantee no iterator changes
* vector, deque make NO such guarantee

Always watch which iterators are assured to be changed/unchanged