The Standard Library

Warwick RSE

28/09/2023

Concept

What is the Standard Library?

- In the 1990s Alexander Stepanov developed something called the **Standard Template Library** (STL)
  - Implemented generic solutions to many common tasks in programming, mainly around data structures and manipulation of data structures
- Was seen as so valuable that it became a core part of the C++ language in the 90s (being renamed Standard Library) and every new release of the C++ language since 2011 has included some expansion of the standard library
- As the name “library” implies the standard library is itself written in C++ and the implementation doesn’t **have** to be closely tied to a compiler
  - It usually is though!

What is the Standard Library?

- For most common data structures there is an implementation in the standard library
- The implementations are not “optimal” for every possible problem, but are “optimal enough” for most purposes
- We’re going to talk about the two most common containers here
  - Vector - A growable array
  - Map - An associative array (mapping a key to a value)

Vector

What is a vector?

- Like an array, but you can add and remove elements
- Elements can be of almost any type **but**
- All of the elements must be of the same type
- Possible but very difficult to store different types in a vector
  - **DOING THIS IS A BAD IDEA IN ANY COMPILED LANGUAGE, INCLUDING C++**

std::vector

```
#include <vector>
#include <iostream>

const int n_elements=10;

int main(){
  std::vector<int> v;
  for (int i =0;i<n_elements;++i){
    v.push_back(i);
  }

  for (int i =0;i<v.size();++i){
    std::cout << "Element " << i << " has value << " << v[i] <<"\n";
  }
}
```

std::vector

```
#include <vector>
#include <iostream>

const int n_elements=10;

int main(){
  std::vector<int> v;
  for (int i =0;i<n_elements;++i){
    v.push_back(i);
  }
 for (int i =0;i<v.size();++i){
    std::cout << "Element " << i << " has value << " << v[i] <<"\n";
  }
}
```

- Have to include **<vector>** header to use vectors

std::vector

```
#include <vector>
#include <iostream>

const int n_elements=10;

int main(){
  std::vector<int> v;
  for (int i =0;i<n_elements;++i){
    v.push_back(i);
  }
 for (int i =0;i<v.size();++i){
    std::cout << "Element " << i << " has value << " << v[i] <<"\n";
  }
}
```

- Create an instance of a vector
  - **std**:: is a **namespace** all standard library items are in the **std** namespace
  - <int> is not another header file, but a template parameter - says to make a vector holding **int**s here

std::vector

```
#include <vector>
#include <iostream>

const int n_elements=10;

int main(){
  std::vector<int> v;
  for (int i =0;i<n_elements;++i){
    v.push_back(i);
  }
 for (int i =0;i<v.size();++i){
    std::cout << "Element " << i << " has value << " << v[i] <<"\n";
  }
}
```

- You can add elements to the end of a vector using the **push_back** method
- You call it like **v.push_back** because you need to know **which** vector to add to

std::vector

```
#include <vector>
#include <iostream>

const int n_elements=10;

int main(){
  std::vector<int> v;
  for (int i =0;i<n_elements;++i){
    v.push_back(i);
  }
 for (int i =0;i<v.size();++i){
    std::cout << "Element " << i << " has value << " << v[i] <<"\n";
  }
}
```

- You can find out how many elements there are in a vector using the **size** method

std::vector

```
#include <vector>
#include <iostream>

const int n_elements=10;

int main(){
  std::vector<int> v;
  for (int i =0;i<n_elements;++i){
    v.push_back(i);
  }
 for (int i =0;i<v.size();++i){
    std::cout << "Element " << i << " has value << " << v[i] <<"\n";
  }
}
```

- You can access an element of a vector with [] just like you can with a normal array

What is it good for?

- Already seen one useful feature of **std::vector**
  - You can add elements to it indefinitely, you don't have to specify the size in advance
- It also guarantees that it stores the underlying data **contiguously** - that is one item after the other in memory
- This data layout is the same as a normal array
- If you have a function in a library that needs an array you can get access to the underlying memory with **myvector.data()**

What else is it good for?

- As well as defining the containers, the standard library also defines algorithms (in the header **algorithm**) that you can apply to containers. For example
- **std::sort** - Sorts the content of the container, guaranteed O(Nln(N)) average complexity (c.f. quicksort, mergesort), can have custom comparison
- **std::find** - Find a value in the container (doesn't assume sorted, there is also **std::binary_search** if it is sorted)
- **std::min_element** and **std::max_element** - Find minimum or maximum value
- **std::for_each** - Apply a function to each element of the container

What's the problem?

- Mostly not much - vectors work well, are highly optimised and can fall back to working like arrays for working with older library code
- The problem is **how** it implements that growing behaviour
- Since there is a requirement that the underlying memory be contiguous it does sometimes have to reallocate that memory to store more items
  - No longer in the same place in memory
- When it does so, various things break, most notably **iterators**

What is an iterator?

- An iterator is an object that provides access to the elements in an STL container
- The idea is that you ask a container to give you an iterator to the beginning of the items
- Then you either
  - Move the iterator to another item by calling **++**,**+=**,**--**, **-=** or similar on it
  - Get the current item by **dereferencing** the iterator with ***
    - If you are a C (or old style C++) programmer this syntax looks like pointer dereferencing

What is an iterator?

```cpp
#include <vector>
#include <iostream>

int main(){

  std::vector<int> v;
  for (int i=0; i<10;++i){
    v.push_back(i*2);
  }

  for(auto it = v.begin(); it!=v.end();++it){
    std::cout << *it << "\n";
  }

}
```

What is an iterator?

```cpp
#include <vector>
#include <iostream>

int main(){

  std::vector<int> v;
  for (int i=0; i<10;++i){
    v.push_back(i*2);
  }

  for(auto it = v.begin(); it!=v.end();++it){
    std::cout << *it << "\n";
  }

}
```

- Use the begin method of your container to get the iterator to the first item
- The actual type of an iterator is moderately complex and depends on the container and the data in the container
- Just use **auto** in general

What is an iterator?

```
#include <vector>
#include <iostream>

int main(){

  std::vector<int> v;
  for (int i=0; i<10;++i){
    v.push_back(i*2);
  }

  for(auto it = v.begin(); it!=v.end();++it){
    std::cout << *it << "\n";
  }

}
```

- To check if you have reached the last element compare your iterator to that returned by the **end** method of your container
- The termination condition here is **!=v.end()** which feels a bit odd since you are used to testing for **<** or **<=**
- Here you want it to stop as soon as it reaches the special **end** marker item
- **NB! end()** doesn't return the last item, it returns a special marker after the last item

What is an iterator?

```
#include <vector>
#include <iostream>

int main(){

  std::vector<int> v;
  for (int i=0; i<10;++i){
    v.push_back(i*2);
  }

  for(auto it = v.begin(); it!=v.end();++it){
    std::cout << *it << "\n";
  }

}
```

- Increment the iterator just like a loop variable in a normal loop
- Can increment in pretty much any way you can increment a normal number

What is an iterator?

```cpp
#include <vector>
#include <iostream>

int main(){

  std::vector<int> v;
  for (int i=0; i<10;++i){
    v.push_back(i*2);
  }

  for(auto it = v.begin(); it!=v.end();++it){
    std::cout << *it << "\n";
  }

}
```

- To access the item that the iterator is referring to use the *** (dereference)** operator
- The iterator is **not** just a pointer to the item but if you are happy with pointers it is a good guide
- Changing the value that you get from the iterator changes the value in the container

Iterator invalidation

- When you add an item to a vector the items may have to be moved in memory if the vector grows
- This **invalidates** the iterator
  - **It no longer works**
- Iterators are also invalidated by removing items
- You have to be careful iterating through a vector using an iterator to either add or remove items
- You can use the **erase** or **insert** method of a vector to add or remove items and it gives you a **new** iterator but you can’t use the original one

Memory Contiguity

- Very common thing to want to do is to remove items from a vector based on a condition
- You can just loop through, test each element and call the **erase** method
- Performance can be bad though
- Vector is required to store the items **contiguously** in memory so when you erase an item the items above it have to be copied down
  - Removing a range is handled automatically with a single copy down
  - Removing individual items based on a condition **isn't** if you use the erase method

Memory Contiguity

- Fortunately this is common enough that C++ gives you a way to do it “properly”
- The classical way of doing it was called the “erase-remove” idiom
  - Move all of the elements that are not to be removed up to the front of the vector using **std::remove_if** (confusing name, but that is what it does)
  - Remove the empty elements (left at the back of the vector) using the **erase** method of the vector
  - Still have to do this if you want to remove items from a subsection of a vector
- In C++20 they introduced a simpler way - **std::erase_if**

std::erase_if

```
#include <vector>
#include <iostream>

bool condition(int &i){
  //Condition is true if number is divisible by 3
  return (i%3)==0;
}

int main(){

  std::vector<int> v;
  for (int i=0; i<10;++i){
    v.push_back(i*2);
  }

  std::erase_if(v,condition);

  for(auto it = v.begin(); it!=v.end();++it){
    std::cout << *it << "\n";
  }

}
```

std::erase_if

```cpp
#include <vector>
#include <iostream>

bool condition(int &i){
  //Condition is true if number is divisible by 3
  return (i%3)==0;
}

int main(){

  std::vector<int> v;
  for (int i=0; i<10;++i){
    v.push_back(i*2);
  }

  std::erase_if(v,condition);

  for(auto it = v.begin(); it!=v.end();++it){
    std::cout << *it << "\n";
  }

}
```

- That's it!
- You write a function that takes an item from your container and returns a **bool**
- I've had my function take an int **reference** - this is permitted but not required.
  - Can be useful if your stored type is big
- The return value should be true if you want the item removed and false if not

std::erase_if

```cpp
#include <vector>
#include <iostream>

bool condition(int &i){
  //Condition is true if number is divisible by 3
  return (i%3)==0;
}

int main(){

  std::vector<int> v;
  for (int i=0; i<10;++i){
    v.push_back(i*2);
  }

  std::erase_if(v,condition);

  for(auto it = v.begin(); it!=v.end();++it){
    std::cout << *it << "\n";
  }

}
```

- You might have to tell your compiler that you want to use C++20 to get this to compile
- This is particularly true on Macs
- usually adding --std=c++20

Anonymous Functions

- If you have a lot of conditions that you use only once in something like **std::erase_if** (or **std::copy_if** or **std::sort** or any of the other algorithms that take functions as a parameter) then it can seem wasteful to have functions hanging around to only be used once

- There is a solution to that in C++ - **anonymous functions**, also called **lambdas**

- Lambdas are **very** powerful and we can't describe them much here, but we'll show the syntax

Anonymous Functions

```
#include <vector>
#include <iostream>

int main(){

  std::vector<int> v;
  for (int i=0; i<10;++i){
    v.push_back(i*2);
  }

  std::erase_if(v,[](int &i){return (i%3==0);});

  for(auto it = v.begin(); it!=v.end();++it){
    std::cout << *it << "\n";
  }

}
```

- The **[ ]( ){ }** pattern indicates that you are defining a lambda
- We're not going to discuss **[ ]**
- **( )** defines a parameter list just like a function
- Then the body of the function is in the **{ }**
- The return type is implicitly **auto**

Brief return to **auto**

- That automatic return type isn't specific to lambdas
- Any function can have **auto** as its return type
- The compiler infers the return type from the return statements in the function
- **MUST ALL RETURN THE SAME TYPE**
  - This doesn't let you return different types from different paths through the function
- Can make your code confusing - if it is hard to work out the return type maybe don't use auto. Is OK if it is easy but the return type is complex (i.e. returning an iterator)

Classical algorithm example

- **std::erase_if** solves one of the most common things that you want to do with a vector, but it isn't quite the "normal" sort of C++ algorithm function
- As a better example of typical STL algorithms, we'll show **std::sort**
- This function sorts the elements in a vector
  - By default it sorts them in ascending order (technically non-descending order, but mostly that doesn't matter)
  - You can give it a custom function to do other types of comparison or to sort types that are not trivially comparable to each other

std::sort

```cpp
#include <vector>
#include <iostream>

int main(){
  std::vector<int> v;
  //Store numbers 1 to 10
  for (int i=1;i<=10;i++) v.push_back(i);

  //Sort the vector
  //For a normal ascending order sort you should return
  //Whether i1 is < i2
  //Here, we do the opposite so the sort is in descending order
  std::sort(v.begin(),v.end(),[](int i1,int i2){return i1>i2;});
  //Print the result (will be descending order)
  for (auto it = v.begin(); it!=v.end();it++){
    std::cout << *it << "\n";
  }
}
```

std::sort

```
#include <vector>
#include <iostream>

int main(){
  std::vector<int> v;
  //Store numbers 1 to 10
  for (int i=1;i<=10;i++) v.push_back(i);

  //Sort the vector
  //For a normal ascending order sort you should return
  //Whether i1 is < i2
  //Here, we do the opposite so the sort is in descending order
  std::sort(v.begin(),v.end(),[](int i1,int i2){return i1>i2;});
  //Print the result (will be descending order)
  for (auto it = v.begin(); it!=v.end();it++){
    std::cout << *it << "\n";
  }
}
```

- In most algorithms you specify the start and end iterators for the algorithm to apply over
- This means that you can run them on **part** of a container

std::sort

```cpp
#include <vector>
#include <iostream>

int main(){
  std::vector<int> v;
  //Store numbers 1 to 10
  for (int i=1;i<=10;i++) v.push_back(i);

  //Sort the vector
  //For a normal ascending order sort you should return
  //Whether i1 is < i2
  //Here, we do the opposite so the sort is in descending order
  std::sort(v.begin(),v.end(),[](int i1,int i2){return i1>i2;});
  //Print the result (will be descending order)
  for (auto it = v.begin(); it!=v.end();it++){
    std::cout << *it << "\n";
  }
}
```

- This is another lambda
- It deliberately does the comparison backwards so that it sorts in descending order
- With no comparison function it would just use the **<** and **>** operators

How does vector grow?

- We know that vector contains an arbitrary number of items
- We know that when it grows it may have to move the items to a new bit of memory to accommodate the new items
- Because it is contiguous - all of the items follow one another in memory
- But what actually happens when we add an item? Does it grow by one item?
- **NO**

How does vector grow?

- When vector grows it grows by more items that it immediately needs to add
- Generally as a multiple of the number of elements already in the vector
  - Generally either 2x or about 1.6x (the golden ratio, in particular)
- This means that a vector has two related but different concepts
  - **size** - The number of elements stored in the vector
  - **capacity** - The number of elements that **could** be stored in the vector without having to reallocate memory
- There are methods of a vector with these names to check these values

How does vector grow?

- To go with the concepts of size and capacity, there are methods to set both the size and capacity of a vector

- **resize(N)** - Set the vector to hold N items. The items are immediately created and initialised and can be accessed by index or iterator. i.e change the **size** of the vector

- **reserve(N)** - Set the vector to be **able** to hold N items. The items are not created and are not available, but memory is set aside to hold them. i.e. change the **capacity** of the vector

- **resize**'s use is obvious, but **reserve** is commonly used to give a good first estimate of how many elements ***might*** go into a vector when using **push_back** or **push_front**

Vector Conclusions

- Vector is basically an array but “better”
  - You can add items to it
  - You can remove items from it
- Thanks to **algorithm** there are also many useful functions that you can to do things like sort elements in a vector etc.
  - If you want to do something with your data, look at what is already there!

Pair

std::pair

- **std::pair** is a class that joins together two values that may be of different types
- It isn't really an STL container in itself, but it is used by various other STL containers
- It does have some uses in your own code, but not very commonly
- There is a generalisation of **std::pair** to an arbitrary (but known at compile time) number of connected types called a **std::tuple**
- More useful in places, but not generally needed except for advanced features

std::pair

```cpp
#include <utility>
#include <iostream>
#include <string>

int main(){

  std::pair<int,std::string> i_s_pair;
  i_s_pair.first = 123;
  i_s_pair.second = "Hello world!";

  std::cout << i_s_pair.first << " : " << i_s_pair.second << "\n";

}
```

std::pair

```cpp
#include <utility>
#include <iostream>
#include <string>

int main(){

  std::pair<int,std::string> i_s_pair;
  i_s_pair.first = 123;
  i_s_pair.second = "Hello world!";

  std::cout << i_s_pair.first << " : " << i_s_pair.second << "\n";

}
```

std::pair

```cpp
#include <utility>
#include <iostream>
#include <string>

int main(){

  std::pair<int,std::string> i_s_pair;
  i_s_pair.first = 123;
  i_s_pair.second = "Hello world!";

  std::cout << i_s_pair.first << " : " << i_s_pair.second << "\n";

}
```

std::pair

```cpp
#include <utility>
#include <iostream>
#include <string>

int main(){

  std::pair<int,std::string> i_s_pair;
  i_s_pair.first = 123;
  i_s_pair.second = "Hello world!";

  std::cout << i_s_pair.first << " : " << i_s_pair.second << "\n";

}
```

std::pair

- Quite often if you are working with pairs yourself you want to quickly make a pair from two pieces of data
- You can create the pair and assign the elements as shown above, but there is an easier way
- **auto mypair = std::make_pair(first,second);**
- First and second can be of any type, the compiler will deduce the correct types and create a pair of those types

Map

std::map

- **std::map** is an **associative array** class
- That is it **maps** a key to a value
- You can use a key to store a value
- If you know the key you can retrieve the value
- Each key can have at most one value associated with it
  - There is **std::multimap** that allows more than one value per key

std::map

- For a given **std::map** the key and the value are of a specified type
  - The value can be of almost any type
  - The key must be **orderable** i.e. there must exist < and > operators for the key
- Once again, possible but very difficult to store different types as values
  - **STILL A VERY BAD IDEA**

std::map

```
#include <map>
#include <iostream>
#include <string>

int main(){
  std::map<std::string,int> age_map;

  age_map["William"]=24;
  age_map["David"]=27;
  age_map["Albert"]=67;

  std::cout << "Age of David is " << age_map["David"] << "\n";

}
```

std::map

```
#include <map>
#include <iostream>
#include <string>

int main(){
  std::map<std::string,int> age_map;

  age_map["William"]=24;
  age_map["David"]=27;
  age_map["Albert"]=67;

  std::cout << "Age of David is " << age_map["David"] << "\n";

}
```

- First element of the template is the type of the key
- Most “simple” types will work

std::map

```
#include <map>
#include <iostream>
#include <string>

int main(){
  std::map<std::string,int> age_map;

  age_map["William"]=24;
  age_map["David"]=27;
  age_map["Albert"]=67;

  std::cout << "Age of David is " << age_map["David"] << "\n";

}
```

- Second element of the template is the type of the value
- This can be any constructible type

std::map

```cpp
#include <map>
#include <iostream>
#include <string>

int main(){
  std::map<std::string,int> age_map;

  age_map["William"]=24;
  age_map["David"]=27;
  age_map["Albert"]=67;

  std::cout << "Age of David is " << age_map["David"] << "\n";

}
```

- Access an element of a map with **[]** just like an array or vector
- Type of value in **[]** is now the type of the key

std::map

```
#include <map>
#include <iostream>
#include <string>

int main(){
  std::map<std::string,int> age_map;

  age_map["William"]=24;
  age_map["David"]=27;
  age_map["Albert"]=67;

  std::cout << "Age of David is " << age_map["David"] << "\n";

}
```

- You can both access and set the value
- You don’t have to create a key specially when you first use it

std::map

- That actually causes one of the problems with **std::map**
- If you try to **read** from a map element that hasn't already been set then it is silently created and set to a default value (technically it is **value initialized**)
- If you want to check whether a key is already in the map then you have to test for it
  - Use the **find** method to find the key
  - Use the **count** method to count how often the key appears (it will only ever by 1 or 0 in **std::map** since each key is unique)

std::map

```
#include <map>
#include <iostream>
#include <string>

int main(){
  std::map<std::string,int> age_map;

  age_map["William"]=24;
  age_map["David"]=27;
  age_map["Albert"]=67;

  bool is_William = (age_map.count("William")!=0);
  bool is_David = (age_map.find("David")!=age_map.end());
  bool is_Alice = (age_map.count("Alice")!=0);

  std::cout << "Is \"William\" in the map : " << is_William << "\n";
  std::cout << "Is \"David\" in the map : " << is_David << "\n";
  std::cout << "Is \"Alice\" in the map : " << is_Alice << "\n";

}
```

std::map

- This is actually a lot of how maps are used
- The real power of a map is being able to store and retrieve data based on the key
- Map is a fast container for random access (O(Ln N))
- Sometimes you want to iterate through your map and access the elements
- This is similar, but different, to what you do for vector

Iterating over **std::map**

```
#include <map>
#include <iostream>
#include <string>

int main(){
  std::map<std::string,int> age_map;

  age_map["William"]=24;
  age_map["David"]=27;
  age_map["Albert"]=67;


  for (auto it = age_map.begin(); it!=age_map.end(); ++it){
   std::cout << "Key is   :" << (*it).first << "\n";
    std::cout << "Value is :" << (*it).second << "\n";
  }

}
```

Iterating over **std::map**

```
#include <map>
#include <iostream>
#include <string>

int main(){
  std::map<std::string,int> age_map;

  age_map["William"]=24;
  age_map["David"]=27;
  age_map["Albert"]=67;


  for (auto it = age_map.begin(); it!=age_map.end(); ++it){
    std::cout << "Key is   :" << (*it).first << "\n";
     std::cout << "Value is :" << (*it).second << "\n";
  }

}
```

- You can't iterate by number like you can with a **std::vector**
- Have to use an iterator
- Loop from begin() to end() an increment the iterator

Iterating over **std::map**

```
#include <map>
#include <iostream>
#include <string>

int main(){
  std::map<std::string,int> age_map;

  age_map["William"]=24;
  age_map["David"]=27;
  age_map["Albert"]=67;


  for (auto it = age_map.begin(); it!=age_map.end(); ++it){
   std::cout << "Key is   :" << (*it).first << "\n";
    std::cout << "Value is :" << (*it).second << "\n";
  }

}
```

- As before you **dereference** the iterator with * to get the value

Iterating over **std::map**

```
#include <map>
#include <iostream>
#include <string>

int main(){
  std::map<std::string,int> age_map;

  age_map["William"]=24;
  age_map["David"]=27;
  age_map["Albert"]=67;


  for (auto it = age_map.begin(); it!=age_map.end(); ++it){
   std::cout << "Key is   :" << (*it).first << "\n";
    std::cout << "Value is :" << (*it).second << "\n";
  }

}
```

- Unlike with **std::vector** the value that you get from the iterator **isn't** just the value
- It is a **std::pair** of the key and the value

Iterating over **std::map**

```
#include <map>
#include <iostream>
#include <string>

int main(){
  std::map<std::string,int> age_map;

  age_map["William"]=24;
  age_map["David"]=27;
  age_map["Albert"]=67;


  for (auto it = age_map.begin(); it!=age_map.end(); ++it){
   std::cout << "Key is   :" << (*it).first << "\n";
    std::cout << "Value is :" << (*it).second << "\n";
  }

}
```

- Get the key with .first
- Get the value with .second

(*it). vs **it->**

- You will have noticed that slightly inelegant syntax there
  - **(*it).first** and **(*it).second**
- You get that syntax because you want to dereference the iterator to get the pair and then access a member of the pair
- This type of syntax is so common in C++ (and C where it originated) that there is a special syntax for it **->**
- So **(*it).first** is exactly equivalent to **it->first**
- This is true for all uses of the dereference operator in C and C++

Other loops

- There is one other type of loop that makes it simpler to access elements of a map
- That is the **range based for loop**
- These are loops that iterate through the elements of a container directly, giving you access element by element
- Can be used on any STL container, just a different syntax for the same thing

Range based loop

```
#include <map>
#include <iostream>
#include <string>

int main(){
  //Create the map
  std::map<std::string,int> age_map;

  //Populate the map
  age_map["William"]=24;
  age_map["David"]=27;
  age_map["Albert"]=67;

  //Loop over the map using iterators
  for (auto element:age_map){
    std::cout << "Key is   :" << element.first << "\n";
    std::cout << "Value is :" << element.second << "\n";
  }

}
```

Range based loop

```
#include <map>
#include <iostream>
#include <string>

int main(){
  //Create the map
  std::map<std::string,int> age_map;

  //Populate the map
  age_map["William"]=24;
  age_map["David"]=27;
  age_map["Albert"]=67;

  //Loop over the map using iterators
  for (auto element:age_map){
    std::cout << "Key is   :" << element.first << "\n";
    std::cout << "Value is :" << element.second << "\n";
  }

}
```

- Specify the loop variable first (this use of auto is by far the most common way of using these loops)

Range based loop

```cpp
#include <map>
#include <iostream>
#include <string>

int main(){
  //Create the map
  std::map<std::string,int> age_map;

  //Populate the map
  age_map["William"]=24;
  age_map["David"]=27;
  age_map["Albert"]=67;

  //Loop over the map using iterators
  for (auto element:age_map){
    std::cout << "Key is   :" << element.first << "\n";
    std::cout << "Value is :" << element.second << "\n";
  }

}
```

- Then put a **:** and the name of the container to loop over
- The loop variable will be assigned the value of each element of the container in turn
- For a map, the elements are still **std::pairs** of keys and values

Range based loop

```
#include <map>
#include <iostream>
#include <string>

int main(){
  //Create the map
  std::map<std::string,int> age_map;

  //Populate the map
  age_map["William"]=24;
  age_map["David"]=27;
  age_map["Albert"]=67;

  //Loop over the map using iterators
  for (auto element:age_map){
    std::cout << "Key is   :" << element.first << "\n";
    std::cout << "Value is :" << element.second << "\n";
  }

}
```

- Note that as written here, I do mean that **element** is given the **value** of each element of the container

- Try using it to change values

  - Nothing will happen

Range based loop

```
#include <map>
#include <iostream>
#include <string>

int main(){
  //Create the map
  std::map<std::string,int> age_map;

  //Populate the map
  age_map["William"]=24;
  age_map["David"]=27;
  age_map["Albert"]=67;

  //Loop over the map using iterators
  for (auto &element:age_map){
    std::cout << "Key is   :" << element.first << "\n";
    std::cout << "Value is :" << element.second << "\n";
  }

}
```

- Solution is as simple as putting **&** before the name of your loop variable
- Makes it a reference again
- Can now change the elements of the container

Further auto

- This shows an important element of the **auto** keyword
- **auto** picks up **most** but not **all** elements of the inferred type automatically
- It picks up the type (**int** vs. **float** vs. **std::string** for example), and it picks up whether something is a pointer or not
- It **doesn't** pick up whether something is a reference or a handful of other properties
- If a function returns a reference then you have to use **auto &var** to store the result as a reference variable, otherwise it makes a copy

Structured Bindings

- One of the more useful recent additions to C++ (in C++17) are **structured bindings**
- They are rather like the tuple unpacking in Python
- If you have something that returns a **std::pair** or a **std::tuple** then you can unpack it directly to normal variables rather than having to access the pair using **.first** and **.second**
- Put the variables that should hold the answers into **auto [var1,var2]**
- They can get very complex in more powerful applications but they are useful here

Structured Binding

```cpp
#include <map>
#include <iostream>
#include <string>

int main(){
  //Create the map
  std::map<std::string,int> age_map;

  //Populate the map
  age_map["William"]=24;
  age_map["David"]=27;
  age_map["Albert"]=67;

  //Loop over the map using structured binding
  for (auto [key,value]:age_map){
    std::cout << "Key is   :" << key << "\n";
    std::cout << "Value is :" << value << "\n";
  }

}
```