CSCI251/CSCI851 Spring-2021
Advanced Programming **(S6c)**

Generic Programming III: Containers and iterators

Outline

- What is a container?
- Why use containers?
- A first better Array?
- Accessing elements in a container.
- Iterators.

What is a container?

- Containers are objects that store a collection of other objects.
  - Arrays are a familiar example, the only built-in container.
- We can use class templates to define more complicated container classes.
  - We develop containers to manage collections of objects of other classes in particular ways, tailored to our needs.

Why use container classes beyond array?

- There are a few advantages over arrays:
  - Subscript bound checking.
  - Memory gets tidied up automatically.
  - Inserting elements anywhere may be made easy, depending on the context.
  - Pass by reference or value, arrays passed by reference only.
  - Non-static local arrays cannot be returned, objects of container classes can be.
- Different container classes may be more appropriate in different circumstances:
  - String is effectively a container class and is generally better than an array of chars.

Queues, stacks, and linked lists

- We may want to store and interact with data in a particular way.
  - **Queues** are data structures in which elements are removed in the same order they were entered (FIFO).
  - **Stacks** are data structures in which elements are removed in the reverse order from which they were entered (LIFO).
  - **Linked lists** provide a method of organizing stored data based on a logical order of the data.
- And the methods needed differ between these.

- For linked lists for example, the objects in it contain data and references to other data.
- We need to have methods to establish and manage the links.
- We need to be able to:
  - Insert a new element.
  - Remove an element.
  - Reorder the list.
  - Retrieve and display the objects.

A first better Array …?

- Here goes our new `Array` container, …
- … and we need some classes to test it with.

```
template<typename T>
class Array {
    private:
        T *data;
        int size;
    public:
        Array(T *, int);
        ~Array();
        void display();
};

template<typename T>
Array<T> :: Array(T *d, int s) {
    size = s;
    data = new T [size];
    for( int i=0; i<size; i++ )
        data[i] = d[i];
}
```

```
template<typename T>
Array<T> :: ~Array() {
    if( data != NULL )
        delete [] data;
}

template<typename T>
void Array<T> :: display() {
    for(int i=0; i<size; i++)
        cout << ' ' << data[i];
    cout << endl;
}
```

A Book class

```
class Book {
    friend ostream& operator<<(ostream&, const Book &);
    private:
        string title;
        double price;
    public:
        void setBook(string, double);
};

void Book :: setBook(string t, double p) {
    title = t;
    price = p;
}

ostream& operator<<(ostream& out, const Book &b) {
  out << "title: " << b.title << ", price: " << b.price << endl;
  return out;
}
```

A `Client` class

```
class Client {
    friend ostream& operator<<(ostream&, const Client &);
    private:
        string name;
        double balance;
    public:
        void setClient(string, double);
};

void Client :: setClient(string n, double b) {
    name = n;
    balance = b;
}

ostream& operator<<(ostream& out, const Client &c) {
  out << "name: " << c.name << ", balance: " << c.balance <<
endl;
  return out;
}
```

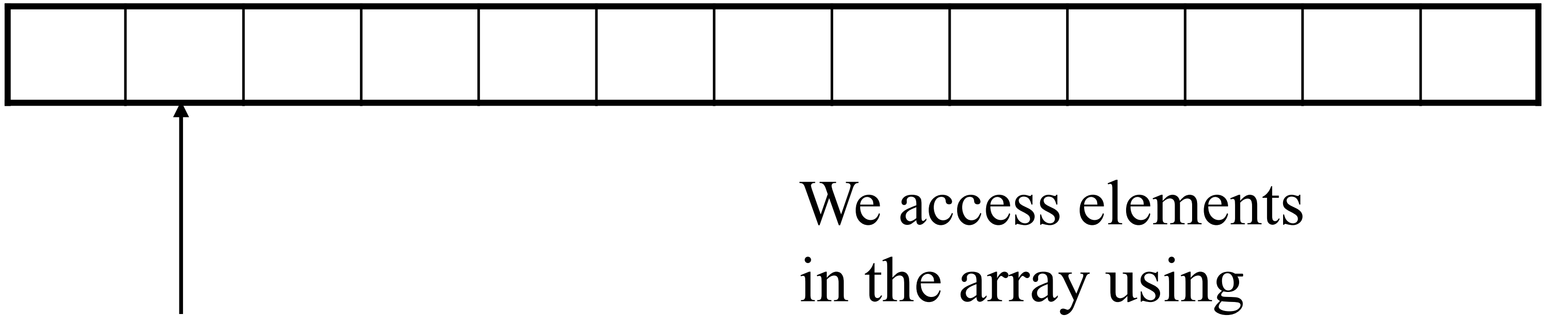
```
int main() {

    int intData[] = {12,34,55};
    double doubleData[] = {11.11, 23.44, 44.55, 125.66};

    Book bookRecs[2];
    bookRecs[0].setBook("Ants", 8.90);
    bookRecs[1].setBook("Solar system", 8.69);

    Client custRecs[3];
    custRecs[0].setClient("Boris Bluenose", 123.45);
    custRecs[1].setClient("Alice and Bob", 23.5);
    custRecs[2].setClient("Oscar the Grouch", 321.5);

    Array<int> intArray( intData, 4 );
    intArray.display();

    Array<double> dArray( doubleData, 3 );
    dArray.display();

    Array<Book> books( bookRecs, 2 );
    books.display();

    Array<Client> clients( custRecs, 3 );
    clients.display();
    return 0;
}
```

Extending the `Array` template…

- We might want to add functions to the `Array` class that:
  - Displays the last element in an array.
  - Displays an element in a specified position in an array.
  - Reverses the order of elements in an array.
  - Sort the elements in an array in ascending or descending order.
  - Remove duplicate elements from an array.
  - …

Accessing elements in a container…

- For an array we have something like …
- … so we access elements via the pointer.



P

We access elements in the array using pointer arithmetic:

```
*P
*P++
*P--
*(P+i)
```

- But this won’t work for double linked lists where the next element isn’t contiguous in memory.
  – And generally accessing the elements of a container is going to require tailoring to the container.
- So, how do reliably and consistently access elements in a container?
  – Reliably also meaning doing better than our classical array where we can walk off the end ☹
- We use iterators!

Iterators

- An **iterator** is an object that moves through a container of other objects and selects them one at a time **without providing direct access to the implementation of that container**.
- Iterators provide a **standard way** to access elements, whether or not the container provides a way to access the elements directly.

Using iterators

- With pointers we use the address-of operator to point at something in particular.
  - For types supporting iterators we instead have members returning particular iterators.
- For an object `v` of a type with iterators …

```
auto b=v.begin();
auto e=v.end();
```

- … `begin` is an iterator denoting the first element, and `end` the off-the-end iterator denoting the position one past the last element.

auto (C++)

- the **auto** keyword directs the compiler to use the initialization expression of a declared variable, or lambda expression parameter, to deduce its type.

```
int a = 10;
auto au_a = a;//自动类型推断，au_a为int类型
cout << typeid(au_a).name() << endl;
```

```
int
```

Iterator operations

- Following the textbook, we can look at the standard operations with an iterator.

Operation	Meaning
`*iter`	Returns a reference to the element denoted by `iter`.
`iter->mem` `(*item).mem`	Deferences `iter` and fetches the member named mem from the underlying element.
`++iter` `--iter`	Increments `iter` to refer to the next element. Decrements `iter` to refer to the previous element.
`iter1 == iter2` `iter1 != iter2`	Compares two iterators for equality or inequality. Equal if they denote the same element or are both the off-the-end iterator for the same container.

Example: Iterator for a `string`

- Here goes an example that illustrates the use of an iterator, in this case for a `string`.
  - The `string` class is not actually a container class but it has a lot of functionality in common with containers and we have at least seen it before.

```
string s("this is a string");
if (s.begin() != s.end()){
     auto it = s.begin();
     *it = toupper(*it);
}
```

```
CC -std=c++11 test.cpp
```

- To step through elements in our string we can do the following …

```
for (auto it=s.begin(); it != s.end(); ++it)
     cout << *it << " ";
```

- The `"  "` highlights the element by element.

- Well $<$ is often undefined, while $!=$ will be defined for iterators so the $!=$ form is going to be widely used.

`const` iterators

- The type of iterator pointing to a `const` object is different to that not …

```
#include <iostream>
#include <typeinfo>
using namespace std;

int main()
{
        string s1("this is a string");
        const string s2("this is a string too");

        auto it1=s1.begin();
        auto it2=s2.begin();

        cout << typeid(it1).name() << endl;
        cout << typeid(it2).name() << endl;
}
```

```
$ CC -std=c++11 test.cpp
$ ./a.out
N9__gnu_cxx17__normal_iteratorIPcSsEE
N9__gnu_cxx17__normal_iteratorIPKcSsEE
```

- Sometimes we want to access a container only for reading, and not for writing, even though the container may not itself be `const`.
- C++11 allows us to have an iterator that recognises this.
- We can use `cbegin()` and `cend()` in place of `begin()` and `end()` to get `const` iterators.

More on iterators later …

- We should be able to look at iterators a bit more later.
  - There are different kinds of iterators and various other operations, including arithmetic, that we can sometimes use with them.
- For now we are going to turn to the standard template library to took at how the containers and iterators work with `vector`.
  - We can take vector as a prototype for how we the rest of the standard template library works and as indicative of how templating generally works.

Outline

- What is STL?
- The building blocks.
- Why/when should you use STL?
- Why/when shouldn’t you use STL?

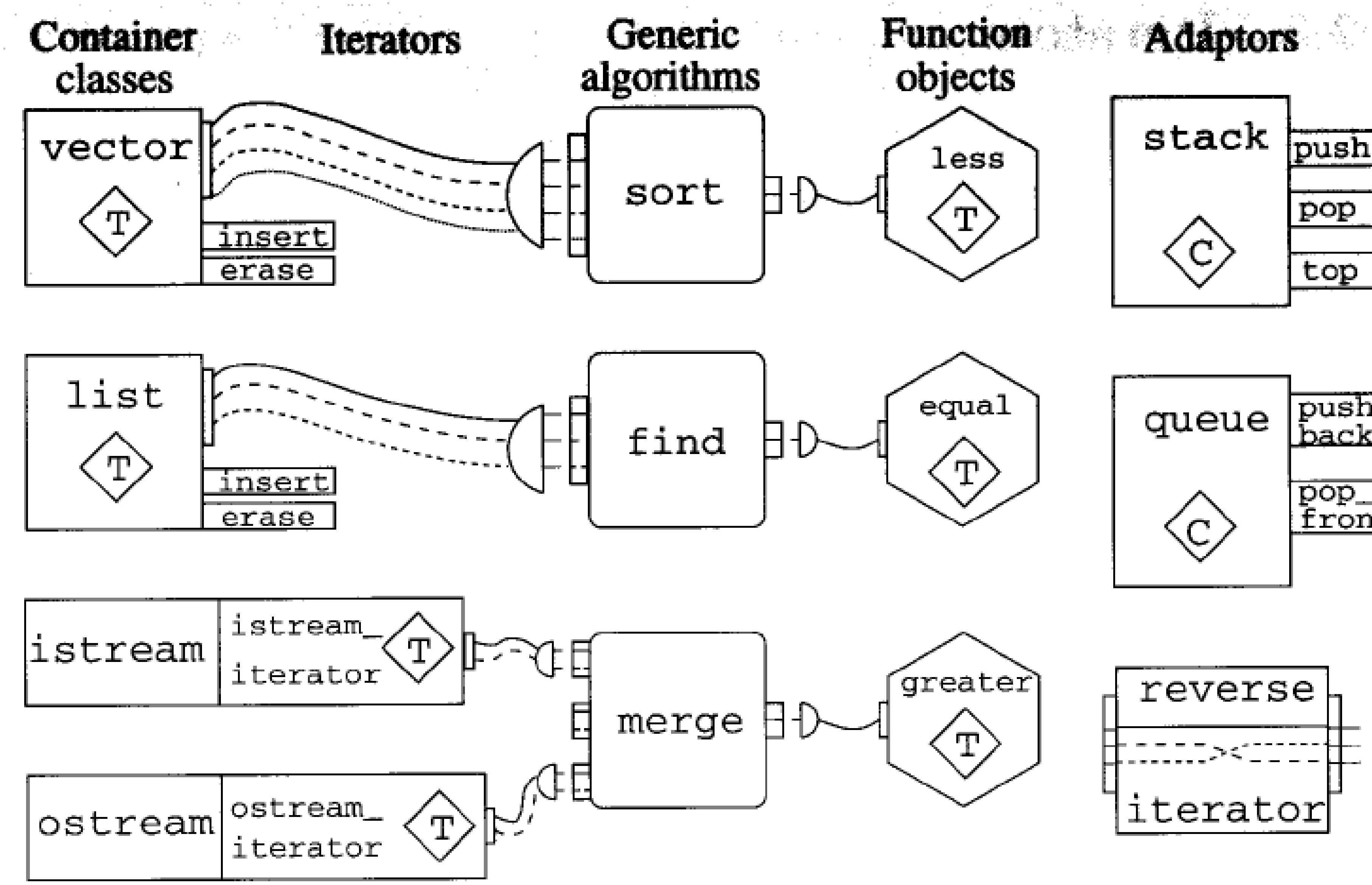
What is STL?

- The Standard Template Library for C++.
- It's an evolving standard containing the result of many years research into generic programming.
  – So making use of compile time polymorphism rather than the run time polymorphism we have in object oriented programming.
- It's used to provide reusable and widely adaptable solutions, that are never-the-less efficient.
  – STL contains various building blocks.

- In object oriented programming we attempt to tightly bind data and the operations on the data.
- In generic programming we attempt to decouple them.
- We put the data into containers and use the iterators as an interface to use standard algorithms on data.
  - The container `vector` is a critical part of modern C++, and is part of STL.

The building blocks

- STL contains six major kinds of components, implemented using templates:
  - Containers.
  - Iterators.
  - Generic algorithms.
  - Function objects.
  - Adaptors.
  - Allocators.



- Not every algorithm and iterator can be connected.

Performance vs memory size

- Since STL is based on function and class templates it can process all built in and user defined data types.
- Each version of a component generated from a template becomes specialized for a particular data type and therefore can be as efficiently as C++ code written directly for this data type.
  - Note that if many different data types are used, a large number of specialized components may result in substantial executable size expansion.

Why/when should you use STL?

- Code reuse, the code has already been written and tested.
- Since it’s in the standard, you can expect portability too.
- You also get guarantees about the performance of particular parts of STL.
  - STL aims to do as well as you could if you crafted non-templated code.
  - There are improvements over C++98/C++03 performance.

STL Containers

- There are two types of containers:
  - Sequential containers, in which the programmer controls the order in which elements are added and accessed, and the position of elements is determined when they are inserted.
    - The elements are organised linearly.
  - Associative containers, in which element are stored and linked on the basis of a key value.
    - The elements are not necessarily organised linearly.

Sequential containers: Section 9.1

Container	Notes
`vector`	Flexible-size array. Supports fast random access. Inserting/deleting other than at the back may be slow.
`deque`	Double-ended queue. Supports fast random access. Fast insert/delete at front or back.
`list`	Doubly linked list. Supports only bidirectional sequential access. Fast insert/delete at any point.
`forward_list`	Singly linked list. Supports only sequential access in one direction. Fast insert/delete at any point. C++11
`array`	Fixed-size array. Supports fast random access. Cannot add or remove elements (positions). C++11
`string`	Specialised, not fully templated. Similar to vector but for characters. Fast random access. Fast insert/delete at the back.

Random access → Access elements in an arbitrary order with similar performance. Contrasts with sequential access where you need to go through other elements to reach the one you want.

Section 9.1: Choosing a sequential container …

- Default: `vector`.
  - Use it unless you have a good reason not to.
- Lots of small elements and space overhead matters? Avoid `list` and `forward_list`.
- Need random access to elements: `vector` or `deque`.
- Insert or delete elements in the middle: `list` or `forward_list`.
- Insert or delete elements only at the front and back: `deque`.
- More complex scenarios:
  - Insert elements in the middle only while reading input, and subsequently needing random access.
    - Possibly just use `vector` anyway with elements to be added at the end, following by a call to `sort` it once the input is finished.
    - Or, consider using a `list` for the input phase and then copy the list into a `vector`.
  - …

```
#include <iostream>
#include <vector>
using namespace std;

int main()
{
   size_t size;
   cout << "Enter the size of the container: ";
   cin >> size;

   // get space for size integers and initialize them to 0
   vector<int> intArray( size );

   for(int i=0; i<size; ++i)
      intArray[i] = i;
}
```

```
intArray vector<int>
```

- The variable size is taken care of.
- No need to use dynamic memory allocation.

Associative containers

- Sorted: Elements ordered by key:

Container	Notes
`map`	Associate array; holds key-value pairs.
`set`	Container in which the key is the value.
`multimap`	`map` but with a key that can appear multiple times.
`multiset`	`set` but with a key that can appear multiple times.

- Unordered Collections (C++11):

Container	Notes
`unordered_map`	`map` organised by a hash function.
`unordered_set`	`set` organised by a hash function.
`unordered_multimap`	Hashed `map`; keys can appear multiple times.
`unordered_multiset`	Hashed `set`; keys can appear multiple times.

Iterators

- The iterators that satisfy the requirements of output iterators are sometimes referred to as mutable iterators, while those that don’t are referred to as constant iterators.
- C++17 adds a sixth kind of iterator, the Contiguous Iterator.
  - I think it’s supposed to provide some optimisations above random access iterators in the case where the container elements are stored contiguously.

```
#include <iostream>
#include <vector>
using namespace std;

int main()
{
  vector<char> c;
  c.push_back('A');
  c.push_back('B');
  c.push_back('C');
  c.push_back('D');
  for (int i=0; i<4;++i)
    cout << "c[" << I << "]=" << c[i] << endl;
  vector<char>::iterator p = c.begin();
  cout << "The third entry is " << c[2] << endl;
  cout << "The third entry is " << p[2] << endl;
  cout << "The third entry is " << *(p+2) << endl;

  cout << "Back to c[0].\n";
  p = c.begin();
  cout << "which has value " << *p << endl;
```

`c[2]` is just for `vector`, `array` and arrays...

```
 cout << "Two steps forward and one step back:\n";

  p++;
  cout << *p << endl;
  p++;
  cout << *p << endl;
  p--;
  cout << *p << endl;
  return 0;
}
```

```
$ ./a.out
c[0]=A
c[1]=B
c[2]=C
c[3]=D
The third entry is C
The third entry is C
The third entry is C
Back to c[0].
which has value A
Two steps forward and one step back:
B
C
B
```

More examples:
https://www.learncpp.com/cpp-tutorial/stl-iterators-overview/

Reverse iterators …

- Use `rbegin()` and `rend()`.

```
#include <iostream>
#include <vector>
using namespace std;

int main()
{
  vector<char> c;
  c.push_back('A');
  c.push_back('B');
  c.push_back('C');
  cout << "Forward:\n";
  vector<char>::iterator p;

  for (p=c.begin(); p!=c.end(); p++) cout<< *p << " ";
  cout << endl;

  cout << "Reverse:\n";
  vector<char>::reverse_iterator rp;

  for(rp=c.rbegin(); rp!=c.rend(); rp++) cout<< *rp <<" ";
  cout << endl;

  return 0;
}
```

Adaptors

- A container adapter is a variation of a sequence or associative container that restricts the interface for simplicity and clarity. Container adapters don't support iterators.
- There are three sequential container adaptors:
  – Stack: FIFO.
  – Queue: LIFO.
  – Priority_queue: Prioritises based on, by default, <.

https://docs.microsoft.com/en-us/cpp/standard-library/stl-containers?view=msvc-160

- As an example; the stack adaptor turns a sequential container, other than `array` or `forward_list`, and makes it operate like a stack.

- Iterator adaptors include such things as `reverse_iterator`, `move_iterator`, ...

http://en.cppreference.com/w/cpp/iterator#Iterator_adaptors

And stream iterators …

Allocators

- The library class `allocator` is used to allocate unconstructed memory.
- It provides us with the means to separate the allocation of memory from the construction of objects in that memory.
- From: http://en.cppreference.com/w/cpp/concept/Allocator
- “Encapsulates strategies for access/addressing, allocation/deallocation and construction/destruction of objects.”

A bit more STL

- More on vectors.
- More on allocators.
- Pairs.
- Pair aliasing.

More on `vector<T>`

- The STL container class template `vector<T>` can be taken as a guide for how to do containers.
  - We are not going to dwell on the others in the lectures.
- We go going to look at some examples to review some of the `vector<T>` basics, and introduce some new fundamentals at the same time.
  - Including an output stream iterator.

```
#include <iostream>
#include <vector>
#include <string>
#include <iterator>  // needed for ostream iterator
#include <algorithm>

using namespace std;

int main()
{
    // create empty vector container for strings
    vector<string> sentence;

    // reserve memory for five elements to avoid reallocation
    sentence.reserve(5);

    // append some elements
    sentence.push_back("STL");
    sentence.push_back("C++");
    sentence.push_back("tutorial");
    sentence.push_back("reference");
    sentence.push_back("guide");
```

```
// print elements separated with spaces
for( int i=0; i<sentence.size(); ++i )
    cout<<sentence[i]<< " ";

cout << endl;

// print properties
cout << "  max_size(): " << sentence.max_size() << endl;
cout << "  size():     " << sentence.size()     << endl;
cout << "  capacity(): " << sentence.capacity() << endl;

// swap the first and second elements of the vector
swap (sentence[0], sentence[1]);

// insert a word "and" before the word "reference"
vector<string>::iterator insIt =
          find(sentence.begin(),sentence.end(),"reference");
if( insIt != sentence.end() ) // if found
   sentence.insert( insIt, "and" ); // insert a word

sentence.push_back(".");
```

```
    // print elements separated with spaces
    ostream_iterator<string> outIt( cout," " );
    copy( sentence.begin(), sentence.end(), outIt );

    cout << endl;

    // print properties
    cout << "  max_size(): " << sentence.max_size() << endl;
    cout << "  size():     " << sentence.size()     << endl;
    cout << "  capacity(): " << sentence.capacity() << endl;
}
```

- No visible output loop, it’s in the `copy` algorithm which takes a stream iterator, called `outIt` here.

- As with other containers, `vector<T>` is a class template:

```
template <class T, class Allocator = allocator<T> >
class vector {
      //...
  }
```

- … where `T` is the type of data being stored and `Allocator` specifies an allocator used to implement a dynamic memory allocation strategy for the container.

More on `allocator`

- This came up in the STL introduction briefly.
  - The class `allocator` is defined in the header `memory`.
- Why does it exist?
  - The use of `new` is constrained in that it combines allocating memory with constructing an object, or objects, in that memory.
  - As we stated earlier, an allocator allows these two operations to be separated.

- Instances of the class allocator can be use to provide type-aware allocation of raw, unconstructed, memory.
- We defined an allocator object for objects of type T as follows:

```
allocator<T> a;
```

- The operations for allocation, deallocation, creation and destruction are paired, and we will summarise them on the next slide.
  - We will put them in the order they typically need to be used in.

Operations in an `allocator`

`auto const p = a.allocate(n);`	Allocates raw, unconstructed memory to hold n objects of type `T`, and sets up `p` to point to it.
`a.construct(p, args);`	Calls the constructer for objects of type T, associated with the `T*` pointer `p`.
`a.destroy(p);`	Calls the destructor on the object pointed to by the `T*` pointer `p`.
`a.deallocate(p,n);`	Deallocates the memory of `n` type `T` objects pointed to by `p`.

- Where is the advantage?
  – You only destroy what you construct.

- The arguments on the constructor are, as of C++11, allowed to match any constructor for the relevant class.

```
allocator<string> alloc;
auto const p = alloc.allocate(5);

auto q=p;
alloc.construct(q++);
alloc.construct(q++, 10, 'c');
alloc.construct(q++, "hi");

auto r=p;
do
        cout << *r << endl;
while (++r != q);

while (q != p)
        alloc.destroy(--q);
alloc.deallocate(p, 5);
```

`q` will be used to point to one past the last constructed.

3 constructors …

Displaying …

Destroying…

Deallocating…

Back to `vector<T>`: constructors

- There are a fair few options, listed without the allocator here.
- Default, with an empty vector.

```
vector<int> v0;
```

- Initialisation with values …

```
vector<int> v1(10, -1);  // 10 ints set to -1
```

- Initialisation without values …

```
vector<int> v2(10);  // 10 ints, default set to 0.
```

- By copying from another suitable `vector<T>`.

```
vector<int> v3(v2);
vector<int> v4=v2;  // equiv. to above
```

- Iterator based construction:

```
vector<int> v5(v4.begin(),v4.end());
```

- List initialisation, from C++11.

```
vector<string> words = {"one",
"two", "red", "blue"};
vector<int> numbers{1,2,3,4,5,6,7};
```

A vector of vectors …

- The elements of a vector can be vectors themselves.
- A special notation was needed, the addition of a space between the last >,

```
vector<vector<int> > v6;
```

- … but isn’t from C++11…

```
vector<vector<int>> v6;
vector<vector<vector<vector<vector<int>>>>> v7;
```

- This was also true for some other composites.

Vector operations …

- The usual comparison operators:

  == (equivalence)

  < (less than)

  <= (less than or equal to)

  > (greater than)

  >= (great than or equal to)

  != (not equal to)

- But to store something in a vector we need to have: a default constructor, assignment operator (=), equivalent operator (==) and less then operator (<).

Sequential containers: Adding and removing element …

- Pushing and popping …

```
void push_back(const T& x);
void push_front(const T& x); // for deque
void pop_back();
void pop_front();                    // for deque
```

- The use of pop removes an element but doesn’t free the associated memory.
- We also saw the use of `insert` earlier.

`pair`: A `utility` library type

- This templated type is helpful for understanding associative containers, so maps, sets and their extensions.
- A `pair` is constructed with two type names, and the data elements of the `pair` have the corresponding types.
- For example,

```
pair<string, string> writers;
pair<string, string> musician{"Billy", "Joel"};
```

- Unusually, the data members of a `pair` are public.
  - So you need to be careful where you use these.
- The data members are accessed as `first` and `second`.

```
musician.first;
musician.second;
```

- For the associative containers you might be doing something like counting instances of words using a `pair` like …

```
pair<string,int> wordcount;
```

Constructors …

- These change a fair bit across C++11, C++14, C++17.

http://en.cppreference.com/w/cpp/utility/pair/pair

- See also the function template `make_pair`…

http://en.cppreference.com/w/cpp/utility/pair/make_pair

Aliasing templates

- Sometimes we want a short name to capture a specific instantiation of a class template.

```
typedef Blob<string> StrBlob;
```

- C++11 allows type aliases for a class template, such as pairs.
- Here goes an example …

```
template<typename T> using twin=pair<T, T>;
twin<string> authors;
```

- We use `twin` as a synonym for pairs with the same type.

- This means we can have things like …

```
twin<int> win_loss;
twin<double> dimensions;
```

- … so that `win_loss` is a `pair <int,int>` and `dimensions` a `pair<double, double>`.

```
#include<iostream>
using namespace std;

template<typename T>using twin=pair<T, T>;

int main()
{

twin<string> musician("Billy","Joel");

cout << musician.first << " ";
cout << musician.second << endl;
}
```