CE221
Programming in C++
Part 6
Templates and Containers, the vector class, Iterators, STL algorithms
Template Functions 1

Consider the following function.

```c
void swap(int &a, int &b)
{
    int temp = a;
    a = b;
    b = temp;
}
```

This can be used to swap the values in two variables of type `int`; if we wished to swap the values of two variables of type `float` or of type `string` we would have to write extra versions of the function with different argument types.

It is possible to avoid the need to write overloaded versions of functions by using a `template` function.
Here is a template version of the function from the previous slide:

```cpp
template <class T>
void swap(T &a, T &b)
{
    T temp = a;
    a = b;
    b = temp;
}
```

The template declaration says that \( T \) is a type parameter that can be instantiated with any type (it doesn’t have to be a class).

If we make a call `swap(x, y)` where \( x \) and \( y \) are variables of type `float`, the compiler will generate a version of the function with \( T \) instantiated to `float`. 
The compiler will generate a separate instance of the template function for each type for which a call is made, so if we made calls \( \text{swap}(x, y) \) and \( \text{swap}(a, b) \) where \( x \) and \( y \) are of \texttt{float} and \( a \) and \( b \) are of \texttt{int} two separate versions of the function will be created. The machine code would hence be the same as if the programmer had written overloaded versions of the function.

Note that when matching argument types with template types the compiler will not attempt to perform any type conversions apart from implicit referencing and dereferencing, so the function on the previous slide cannot be called with two arguments of different types even if a conversion between the two types exists.
We could write a template function to print the contents of an array in a desired format:

```cpp
template <class T>
void printArray(const T array[], int count)
{
    for (int i = 0; i<count; i++)
        cout << array[i] << "  ";
}
```

If a call is made to this function with the first argument being an array of objects of some class for which the `<<` operator has not been overloaded the compiler will report an error.
A template function can have more than one type parameter, e.g.

```cpp
template <class S, class T>
void printPair(const S &s, const T &t)
{ cout << '<' << s << ', ' << t << '>'; }
```

Here \( S \) and \( T \) may be instantiated with the same type or with different types.
A template class is a class in which one or more of the members has a parameterised type, e.g.

The following class can store pairs of objects.

```cpp
template <class S, class T>
class Pair
{
private:
    S s;
    T t;
public:
    Pair(S a, T b): s(a), t(b) { };
    S getFirst() const { return s; };
    T getSecond() const { return t; }
};
```
A variable to hold an object of this class would be declared using the syntax

```cpp
Pair<string, int> a("fred", 35);
```

Note that if we write a template class in a file of its own we must place the complete function definitions in the header file. If the complete definitions of `getFirst` and `getSecond` were in a file called `Pair.cpp` the compiler when compiling this file would not know what instantiations of `S` and `T` are required by code in other files that uses the class and hence cannot generate appropriate versions of the functions.
The Standard Template Library 1

The *standard template library* (STL) provides a number of template classes for various kinds of collections of objects, and algorithms that can be applied to these classes. Each of these template classes is known as a *container*.

There are three *sequence containers* to store sequential data: *vector*, *deque* and *list*. The *string* class, which we have already encountered, is also regarded as a sequence container (although it is not a template class).

In addition there are three *container adaptors*: *stack*, *queue* and *priority_queue*; these do not provide the full functionality of containers because the operations that may be performed on them are restricted (e.g. we are not allowed to access the middle of a stack).
There are four *associative containers* to store non-sequential data with rapid searching: *set*, *multiset*, *map* and *multimap*. The are two other non-template classes that have most of the functionality of containers: *bitset* and *valarray*. All of the container classes and container adaptors in the STL have a no-argument constructor to initialise a collection to be empty, a copy constructor to initialise a collection to contain copies of the objects in an existing collection, an assignment operator and a destructor. Most containers also have other constructors.
All container classes and container adaptors have member functions called `size` and `empty`. These have no arguments; the former returns the number of elements in the collection and the latter returns a boolean result.

There is also a function `max_size` which returns the maximum possible size for the container. (This will be implementation-dependent.)

Each class has a `swap` function, which takes as an argument a reference to another container of the same type: after a call to `c1.swap(c2)`, `c1` will contain the previous contents of `c2` and `c2` will contain the previous contents of `c1`. 
Each class other than `priority_queue` also has the usual six comparison operators, although the behaviour of operators such as `<` depends on the container.

The sequence container and associative container classes have some additional member functions.

The `clear` function (which takes no argument and returns no result) resets the container to be empty.

There are also functions `begin`, `end`, `rbegin` and `rend` which can be used to obtain `iterators` to traverse through the elements of a collection and `erase` to remove one or more elements (using an iterator).
The `vector` template class provides an implementation of arrays with range-checking and should be used when we want to access elements by position. Programs that use this class should contain the line `#include <vector>`.

The no-argument constructor will initialise a vector to have no contents and a default capacity. There is also a constructor with an argument of type `int` which initialises the vector to contain a fixed number of elements, each initialised to the same value – this value may be specified in a second argument which defaults to 0 (or the equivalent for any other type):

```cpp
vector<string> v1;
vector<int> v2(20); // will contain 20 zeroes
vector<char> v3(10, 'x'); // will contain 10 'x's
```
The vector Class 2

Note that for a vector of class objects the use of the single-argument `vector` constructor will cause initialisation of the objects in the vector to be performed by the class’s no-argument constructor, so its use will be rejected by the compiler if no such constructor exists. In addition some other `vector` methods make use of the no-argument constructor so when writing a class that is expected to be used in collections a no-argument constructor should normally be provided.
The vector Class 3

It is possible to initialise the contents of a vector to be a copy of the contents of an array. This is done using a constructor with two arguments – the first is the address of the beginning of the array and the second is the address of the memory location immediately after the end of the array.

Using address arithmetic the appropriate value for the second argument is calculated by adding the length of the array to its start address; we can use a smaller second argument or a larger first argument to copy part of the array into the vector.

```cpp
int a[] = { 1, 2, 3, 4, 5, 6, 7 };
vector<int> v4(a, a+7);
vector<int> v5(a, a+3); // will contain 1,2,3
vector<int> v6(a+4, a+7); // will contain 5,6,7
```
As with the `string` class elements can be accessed using a member function `at` or via subscripting – only the `at` function provides range-checking so for safe access to the elements of a vector an expression such as `v.at(n)` should be used.

There are member functions `front` and `back` which return references to the first and last element; these can be used as the left-hand operand of an assignment statement:

```cpp
v2.front() = 7;
cout << v2.back();
```

If the vector is empty a call to either of these two functions will throw an exception.
There are member functions `push_back` which will append an element to the end of the vector and `pop_back` which will remove the last element (or throw an exception if the vector is empty).

If a call to `push_back` results in the size of the vector becoming greater than the current capacity the capacity will be doubled – this involves dynamic allocation of a new larger block of memory and copying the existing elements into this block. Hence if we know that many calls to `push_back` will be made it can sometimes be more efficient to increase the capacity once using `reserve` before making any of the calls:

```cpp
v.reserve(100);  // increase capacity to 100
```
The vector Class 6

The `reserve` function can also be useful if the size of a large vector is about to reach its capacity and we know that we are going to add only a small number of new elements. We can prevent the automatic doubling of the capacity (which would result in a large amount of unused space being allocated) by reserving enough space for the extra elements, e.g.

```cpp
v.reserve(v.size() + 10);
```
The vector Class 7

It is possible to change the size of a vector using `resize`. If the new size is smaller than the original size the vector will be truncated; if it is greater multiple copies of an extra value will be appended to the end of the vector. The second argument, if provided, specifies this value; the default is 0 or its equivalent, (so we cannot use this method with a vector of objects of a class without a no-argument constructor):

```cpp
v2.resize(5);
// elements after v2[4] will be deleted
v1.resize(10);
// empty strings appended to vector
// if its size is less than 10
v1.resize(15, "hello")
// 5 copies of "hello" appended to vector
```
It is possible to insert values into locations other than the end and remove values from arbitrary locations – performing either of these tasks requires the use of an iterator. These operations involve shifting of the existing elements to the right of the insertion or deletion point and are hence rather inefficient; hence if they are to be performed frequently it would be more appropriate to use a different container class such as list or deque.
An **iterator** can be regarded as a smart pointer that points to each element in a collection in turn; programs that use iterators should contain the line `#include <iterator>`.

An iterator is not actually a pointer but the unary `*` operator has been overloaded so it can be used to refer to the element of the collection that the iterator "points" to. In addition the `++` operator has been overloaded to allow the "pointer" to be advanced to the next element in the collection. Some iterators also have overloaded version of operators such as `--` and `+=`.

An **iterator** is declared using syntax such as `vector<int>::iterator it`. 
There are three different types of iterator: unidirectional (with just `++`), bidirectional (with `++` and `--`) and random-access (with full "pointer" arithmetic); which iterators are available depends on the container class. There are also constant versions which can be used to traverse a collection but not modify its contents.

To obtain an iterator that starts at the beginning of a collection the `begin` function from the container class should be used, e.g.

```cpp
    vector<int>::const_iterator it = myvec.begin();
```

The `end` function returns an iterator whose "pointer" position is just after the last element of the collection so we can compare the current position of our iterator with `myvec.end()` to determine whether all items have been traversed.
Iterators 3

We can use the following loop to print the contents of a vector $v$ with element type `int` one item per line.

```cpp
vector<int>::const_iterator it;
for (it = v.begin(); it != v.end(); it++)
    cout << *it << endl;
```

The following loop will replace all values greater than 100 in the vector with the value 0. (This time we cannot use a constant iterator.)

```cpp
vector<int>::iterator it;
for (it = v.begin(); it != v.end(); it++)
    if (*it > 100)
        *it = 0;
```
If we wish to traverse a collection in reverse order and the container supports only unidirectional iterators we need to use a reverse iterator. The following loop will print the contents of \texttt{v} in reverse order.

```cpp
vector<int>::const_reverse_iterator rit;
for (rit = v.rbegin(); rit != v.rend(); rit++)
    cout << *rit << endl;
```

Observe that we need to use the functions \texttt{rbegin} and \texttt{rend} to obtain reverse iterators, and \texttt{++} (not \texttt{--}) is used to move to the previous element.
The `vector` and `string` classes support random-access iterators so it is not in fact necessary to use a reverse iterator to access the contents in reverse order; we could use `--` to step through the elements from `v.end()` to `v.begin()`; however we cannot use a for loop since we need to decrement the "pointer" before accessing each element in the vector.

```cpp
vector<int>::const_iterator it = v.end();
while (it != v.begin())
{  it--;
   cout << *it << endl;
}
```
The erase Function 1

The `erase` function of a container class may be used to remove one or more elements from a collection. The function takes one or two arguments which must be non-constant iterators for the appropriate container class. The single-argument version simply removes the element currently "pointed" to by the iterator; an exception will be thrown if the iterator does not "point" to an element of the collection.

We can remove the first element from a vector `v` (or any other container) using `v.erase(v.begin())`. To remove the element at location `n` we could use `v.erase(v.begin()+n)`; this technique can be used only with containers that have random-access iterators.
The erase Function 2

We could use the following code to locate and remove the first occurrence of 0 in the vector v.

```cpp
vector<int>::iterator it = v.begin();
while (it != v.end() && *it != 0)
    it++;
if (it != v.end())
    v.erase(it);
```

Note that after use of the `erase` function the value of the iterator is unspecified. However the function returns a new iterator object that "points" to the next element after the removed element(s) so we can use `it = v.erase(it)` to obtain an iterator with which we can continue iterating.
The two-argument version of \texttt{erase} will remove a sequence of elements starting with the element "pointed" to by the first argument and ending immediately before the element "pointed" to by the second argument so to remove the elements at locations 3, 4 and 5 from the vector \texttt{v} we could use \texttt{v.erase(v.begin()+3, v.begin()+6)}. and to remove all but the first six elements we could use \texttt{v.erase(v.begin()+6, v.end())}.

The two arguments must have the same type so we cannot use a combination of a normal iterator and a reverse iterator.
Inserting into Vectors 1

The `vector` class has three `insert` member functions allowing new elements to be inserted at a position specified by an iterator. (Note that unlike `erase` these functions are not provided by all of the container classes.) In all of these the element(s) will be inserted in front of the position "pointed" to be the first argument; an exception will be thrown if the iterator does not "point" either to an element of the collection or to the position immediately after the last element.
Inserting into Vectors 2

The simplest `insert` function has just two arguments, the second being the item to be inserted.

```cpp
v.insert(v.begin()+2, 4);  // inserts 4 in front of v[2]
```

The second `insert` function allows multiple copies of the same element to be inserted. This takes the number of copies as its second argument and the element as its third argument.

```cpp
v.insert(v.begin()+3, 2, 5);  // inserts 2 copies of 5 in front of v[3]
```
The third \texttt{insert} function allows a sequence of elements from another collection (which does not have to be a vector) or an array to be inserted at a position specified by an iterator. The second argument specifies the location of the beginning of the sequence and the third the location immediately after the end of the sequence. If the sequence comes from an array these arguments should be addresses; otherwise they should be two iterators of the same type each of which points to an element in the same collection object.

\begin{verbatim}
int a[] = ..........
v.insert(v.begin(), a, a+5);
// inserts copies of first 5 elements of a
\end{verbatim}
If several functions in a program used constant reverse iterators for `vector<int>` it would be inconvenient have to use `vector<int>::const_reverse_iterator` in many declarations. To make the code more concise we can make use of `typedef`. This allows a name to be given to any type.

After defining the type `CRI` using

```cpp
typedef vector<int>::const_reverse_iterator CRI;
```

we could declare constant reverse iterators by simply using declarations of the form `CRI it;`. 
The position of the name of a type in a `typedef` statement is always the same position as that of a variable of that type in a declaration.

Hence, for example, since `int *p[40];` would define `p` to be an array of 40 pointers to integers,

```plaintext
typedef int *PA[40];
```

would define the type `PA` to be "array of 40 pointers to integers".  

[ We can tell that the variable declaration gives an array of pointers rather than a pointer to an array since the precedence rules state that the expression `*p[39]` means `*(p[39])` so `p` must be an array and `p[39]` must be a pointer. ]
typedef 3

It is possible to include `typedef` statements inside class declarations.

Consider for example

```cpp
class C {
    typedef ...X...
    ....
}
```

Inside the class declaration we can use the name `X` to refer to the type specified by the `typedef` statement; outside the class we would have to use `C::X`.

This is in fact how the writers of the standard template library were able to define types like `vector<int>::iterator`.
Using `auto` in C++11

C++ 11 introduced a facility for the compiler to infer types of variables from their initialisation, avoiding the need to explicitly use lengthy type names in declarations.

In a declaration of the form

```c++
    auto x = e;
```

the type of `x` will be the type of the expression `e`.

Note that if `e` is a call to a function that returns a reference the type of `x` will **not** be a reference; if we want a reference we must explicitly use the `&` symbol:

```c++
    auto &x = myfun(y);
```
Using auto in C++11 2

Using C++ 11 we could have written on slide 25

```cpp
class it = v.end();
```

instead of

```cpp
std::vector<int>::const_iterator it = v.end();
```

However, since the return type of the `end` function is an iterator not a constant iterator, the code on slide 24 invokes an implicit type conversion. The inferred type of `it` would be `std::vector<int>::iterator`, but we may be willing to accept this if we know that our code will not change the contents of `v`, and do not require the compiler to check this.
Using \texttt{auto} in C++11

Note that \texttt{auto} can be used only when the variable is given an initial value in its declaration.

We could not replace
\begin{verbatim}
vector<int>::const_reverse_iterator rit;
\end{verbatim}
from slide 24 with
\begin{verbatim}
auto rit;
\end{verbatim}

However, we could have used
\begin{verbatim}
auto rit = v.begin();
for (; rit != v.rend(); rit++)
    cout \ll *rit \ll endl;
\end{verbatim}
The standard template library has a large collection of functions known as algorithms that can be used to search and manipulate the contents of containers or strings. To allow these algorithms to be implemented without reference to specific container types they access the containers through iterators and when a reference to an item in the collection needs to be returned this is also done via an iterator.

The STL has about 70 algorithms, including **find**, **replace**, **search**, **sort**, **copy**, and **remove**. Not all algorithms work with all container classes; for example a set is an unordered collection of items (\{1, 2, 3\} is the same set as \{3, 1, 2\}) so there is no concept of being sorted for a set.
STL Algorithms 2

All algorithms take two iterator objects as arguments, specifying the start and end of the portion of the collection to which the algorithm is applied. (If this is the whole collection the results returned by `begin` and `end` or `rbegin` and `rend` should be used.) There will often be additional arguments. As usual the end argument should be an iterator that references the element after the last item in the portion.

To use most algorithms a program must contain the line `#include <algorithm>`. However there are some numerical algorithms (which work only with collections of numbers or objects that have appropriate operators, e.g. `operator+`) for which the line `#include <numeric>` should be used instead.
The find algorithm will return an iterator that references some occurrence of a specific value, supplied as a third argument. In the case of a sequence container or string this will be the first occurrence (or last occurrence if a reverse iterator is being used). If the value does not occur in the collection the end iterator for the collection or portion of the collection (i.e. that supplied as the second argument) will be returned.

The code fragment on the next slide will print the contents of the vector \texttt{v} (of items of type \texttt{int}) from the first occurrence of 0 to the end of the vector and also the substring of the string \texttt{s} extending from the character following the first occurrence of ‘*’ to the end of the string.
typedef vector<int>::const_iterator VecIter;
typedef string::const_iterator StrIter;

VecIter zero = find(v.begin(), v.end(), 0)
for (VecIter it1 = zero; it1!=v.end(); it1++)
    cout << *it1 << ' ';

cout << endl;

StrIter star = find(s.begin(), s.end(), '*')
if (star != s.end())
    for (StrIter it2 = star+1; it2!=s.end(); it2++)
        cout << *it2;

cout << endl;
find 3

The following function will return the number of occurrences of the character `c` in the string `s

```cpp
int count(char c, const string& s) {
    int occs = 0;
    string::const_iterator it =
        find(s.begin(), s.end(), c);
    while (it!=s.end())
    { occs++;
        it = find(it+1, s.end(), c);
    }
    return occs;
}
```
We could also write a template version that will count the number of occurrences of an item in any collection with the correct item type.

```cpp
template <class T, class C>
int count(T val, const C& s)
{
    int occs = 0;
    typename C::const_iterator it =
        find(s.begin(), s.end(), val);

    while (it!=s.end())
    {
        occs++;
        it = find(it+1, s.end(), val)
    }
    return occs;
}
```

The compiler cannot tell that `C::const_iterator` is a type; hence the need for `typename`. 
The two-argument `sort` algorithm will sort the contents of a collection (or part of a collection) into ascending order (i.e. smallest first). It can be used only with containers that support random-access iterators. Comparison of items within the function is performed using the `<` operator so if the items in the collection are objects they must belong to a class that has an `operator<` function. The function changes the order of the contents within the container so it does not make a copy and does not return a result.

If the arguments are reverse iterators the contents of the collection will be sorted into descending order.
// sortdemo.cpp - demonstration of sort
using namespace std;
#include <vector>
#include <iterator>
#include <algorithm>
#include <iostream>
template <class T> void printvec(vector<T> v)
{ cout << '<';
    vector<T>::const_iterator it = v.begin();
    while (it!=v.end())
    { cout << *it;
        if (++it != v.end())
            cout << ',';
    }
    cout << '>' << endl;
}                         // continued on next slide
// sortdemo.cpp continued
int main()
{
    vector<int> v;
    v.push_back(7);
    v.push_back(17);
    v.push_back(3);  // v now holds [7,17,3]
    vector<int> v2(v);  // v2 also holds [7,17,3]
    sort(v.begin(), v.end());
    printvec(v);  // outputs <3,7,17>
    sort(v2.rbegin(), v2.rend());
    printvec(v2);  // outputs <17,7,3>
}
There is a three-argument sort algorithm that will sort the contents using a programmer-supplied comparison function instead of <. This is useful when we wish to sort a sequence of objects and the value to be used for sorting is one of the members of the class but either the operator< function of the class compares a different member or the class has no operator< function.

The third argument to sort should be a pointer to a boolean function that takes two arguments (constant references to the items to be compared) and returns true if the value of the first argument is less than the value of the second argument using the ordering that we wish to use.
The code below will sort a vector v of type Vector<student> by descending average mark (assuming that aveMark is a public member of the Student class).

[ Recall that to pass a pointer to a function as an argument to another function we simply supply the function's name. ]

```
// sortdemo.cpp continued
bool compareMark(const Student& s1, const Student& s2)
{
    return s1.aveMark < s2.aveMark;
}
// need a reverse iterator for descending order
sort(v.rbegin(), v.rend(), compareMark);
```
The `for_each` algorithm provides in C++ some of the functionality of Java's `for int i:myList` or Python's `for i in myList`. Note that as it is a function rather than a loop construct we have to provide the "loop body" as an argument. This must be a pointer to a function that will be called with each of the elements of the collection in turn as its argument; hence it must have a single parameter whose type should be the type of the items in the collection. Any result returned by the function will be ignored.
We can print the contents of a vector of integers one item per line by supplying a function that will print a single integer and a newline as an argument to `for_each`:

```cpp
void print(int i)
{
    cout << i << endl;
}

vector<int> v;

for_each(v.begin(), v.end(), print);
```
In Java we might write code such as:

```java
int sum = 0;
for (int i:myList) sum += i;
```

Since the "loop body" in C++ has to be written as a function it cannot access any local variables from the function that makes the call to `for_each` so if we wished to write similar code in C++ we would have to use a global variable:

```cpp
int sum;
void addtoSum(int i) { sum += i; }
int main()
{ ......
  sum = 0;
  for_each(v.begin(), v.end(), addtoSum);
}
```
accumulate

There are ways to overcome the need to use a global variable by using class objects but these are complicated, so it may be concluded that `for_each` is not as useful as Java and Python's for loops.

Fortunately the use of `for_each` to obtain results like sums or minimum and maximum values is rarely needed since there is in fact an algorithm available to perform such tasks – it is called `accumulate` (and declared in the `<numeric>` header.) We could obtain the sum of the items in a vector using

```cpp
sum = accumulate(v.begin(), v.end(), 0)
```

The third argument is a starting value. Items from the collection are repeatedly added to this with the final value being returned.
Note that the return type of `accumulate` is determined by the type of the third argument since it is being used to build up the result. If we used

```
sum = accumulate(v.begin(), v.end(), 0.0)
```

the sum would be a real number.

To use `accumulate` to generate anything but the sum we need to supply as a fourth argument a pointer to a two-argument function to be performed instead of addition – its first argument and return type need to have the same type as the third argument to `accumulate` and the type of its second argument should be the type of the items in the collection.
A call to `accumulate(it, end, x, f)` will effectively perform a loop of the form

```c
while(it!=end)
    x = f(x, *it++);
```

and return the final value of `x`.

To use `accumulate` to find the minimum value in a collection of integers we could write a function

```c
int minimum(int m, int n)
{
    return m < n ? m : n;
}
```

The first argument represents the smallest item found so far and the second represents the current item in the collection; the value returned is the updated smallest item found so far.
accumulate 4

The initial value for the smallest item so far needs to be supplied as the third argument to `accumulate` – we should use the largest `int` value supported by the C++ implementation – this can be obtained using `numeric_limits<int>::max()` (declared in the header file `<limits>`).

Assuming our program contains the `minimum` function on the previous slide, a call to `accumulate` to find the minimum value in a vector `v` of integers should be of the form

```c++
min = accumulate(v.begin(), v.end(), numeric_limits<int>::max(), minimum);
```
Suppose we have a collection of objects of type `Student` with a member called `mark` (of type `float`) and wish to find the highest mark in the collection. This time we need to write a function which extracts the mark from the current object and compares it with the maximum so far. Assuming no student can have a negative mark the appropriate start value for the maximum so far is 0.

```c++
float maxMark(float m, const Student &s) {
    return s.mark > m ? s.mark : m;
}
```

```c++
max = accumulate(studs.begin(), studs.end(), 0.0, maxMark);
```