Part X

C++ Standard Template Library

Outline

Typedef

- Within C++ type-names can get very long
- We can use `typedef` to create a shorthand

```
typedef std::vector<double> RealVec;
RealVec a; // same as std::vector<double> a
```

- Essentially creating an alias for other types
- This is a C++ construct, not done by preprocessor

Typedefs in classes

- You can define **typedef**s within a class as well:

```
class MyVector{
public:
  typedef double value_type;
private:
  double* data;
};
```

- You can then chain **typedef**s together as:

```
typedef MyVector V;
V v;
V::value_type a = v[0];
```

- The **typedef** does not create a new type, merely an alias (shorthand).

More typedefs

- Another reason for using `typedef` is for maintainability
- Suppose you had a code in which the number of widget types was known to be less than 255:
- You could use `char` to identify a widget.
- If the number of widget types increased to 300, would need to replace all instances of `char` by `int`, unless you had:

```
typedef char WidgetId;
```

in which case changing it to

```
typedef int WidgetId;
```

would work.
This also leads to more readable code: Which is more readable:

```
void sellWidgets(int, int);
```

or

```
void sellWidgets(WidgetId, int);
```

Outline

STL

- The Standard Template Library is one of the most important parts of C++
- The principle of this is to implement support for generic data-structures and algorithms, which can be applied to any (sensible) data-type
- Various functions have guaranteed computational complexity.
- Examples are finding, sorting, summing (accumulating), and iterating

Data-structures

- Firstly, we need to cover some standard containers:
- pair, tuple, vector, map, list, unordered_map
- These are standard constructs from computer science
- Each of these (and some others) is present as a generic form in the STL.
- We have already seen the fixed-size `std::array` and hints of `std::vector`.

pair

- A `std::pair` is a pair of values of any type:

```
std::pair<int, double> a(42, 3.1415926535);
a = std::make_pair(3, 2.71828182843);
int t = a.first;
double e = a.second;
```

tuple

- A `std::tuple` is a set of values of any type:

```
std::tuple<int, double, std::string> a(42, 3.1415926535,
    "Hello!");
int t = std::get<0>(a);
```

- ... and can be used to return multiple values from a function:

```
std::tuple<int, bool> findElt(){
  return std::make_tuple(42, true);
}

int val; bool found;
std::tie(val, found) = findElt();
```

Outline

vector

- A `vector` is an ordered set of items of the same type.
- It allows for random access to the elements in constant time

```
#include <vector>
std::vector<int> a;
a.resize(5);
a[0] = 3;
a[6] = 4; // Will cause undefined behaviour
for(size_t i=0 ; i < a.size() ; i++){
  a[i] = i;
}
```

- So `a` is a `vector` of integers and knows about its own size
- It can be resized at will (unlike basic arrays)
- If enlarged, it may result in copying of data to a new region of memory
- It handles memory allocation and freeing automatically, and is destroyed when it goes out of scope
- The object is stored on the stack, but its data is (probably) stored on the heap.

Vector constructors

- The easiest way to initialize a small vector is to use an initializer list:

```
std::vector<int> aVec{1,4,9,16};
```

- You can also construct a `std::vector` by copying from an array:

```
std::array<int,4> a = {1,4,9,16};
std::vector<int> aVec(&a[0], &a[4]);
```

  Note that `a[4]` is one after the end of the array. The copy is done up to, but not including, this element, so `a[4]` is never read.

- Note that `sizeof` does not take into account the run-time size of a `vector`:

```
std::vector<int> v(10);
std::cout << sizeof(v) << std::endl;
```

  outputs 24 on my computer, but may be wildly different on yours.

- Use `v.size()` instead.

Vector assignments

- Complete assignments can be made later to a vector:

```
std::vector<int> v;
std::array<int,4> a = {1,4,9,16};
v.assign(a, &a[4]);
```

- Or to set `v` to be of size 4 all with value 42:

```
v.assign(4, 42);
```

- A `vector` is guaranteed to have its data held contiguously in memory (i.e. `a[N]` is followed directly by `a[N+1]` in memory).
- Other containers do not have this property.
- The following is therefore valid:

```
std::vector<int> v(10);
int* vStart = &v[0];
int v5 = *(vStart + 5); // same as v[5]
```

Vector operations

```
std::vector<int> a;
a.at(i) = 5;// Same as a[i] but with bounds checking
bool isEmpty = a.empty();
a.clear(); // Clears contents of a
a.push_back(5); // Inserts "5" as extra element at end
int first = a.front(); // Gets first element of a
int last = a.back(); // Gets last element of a
```

Type conversion of containers

- There is no implicit conversion between containers of different types.
- For example:

```
std::vector<int> a(5, 1); // Length 5, all elts=1
std::vector<double> b = a; // Compile-time failure
```

- Even though there is a known type conversion between `int` and `double`, this does not translate into a type conversion between containers of these types.
- If you wish to do the conversion, you have to do it element by element, using a loop.

Outline

Iterators

- An iterator is a type that points to a single element of a data structure, and allows iteration over that structure
- It dereferences to an element of the data-structure

```
std::vector<int> a;
std::vector<int>::iterator vectorIter;
for( vectorIter = a.begin() ; vectorIter != a.end() ;
    ++vectorIter ){
  (*vectorIter) = 1;
}
```

- This sets all elements of the vector `a` to be 1.
- Note that `a.end()` is one element past the end of `a`, so that `!=` is the correct test to use
- There is usually no ordering on iterators, so `!=` is used, rather than `<`
- The reasons for this slightly verbose notation will become clearer later.

Const-iterators

- It is also possible to have iterators over constant structures:

```
int f(const std::vector<int>& a){
  int sum = 0;
  std::vector<int>::const_iterator aIter;
  for( aIter = a.begin() ; aIter != a.end() ; ++aIter ){
    sum += *aIter;
  }
return sum;
}
```

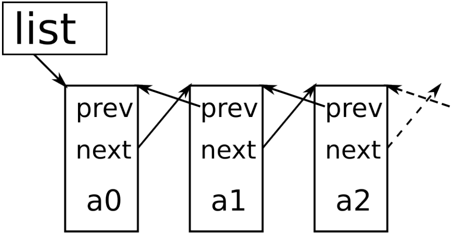
- It is not possible to alter an object through a `const_iterator`.
- It is not possible to take an `iterator` from a constant object.

Outline

Linked List

- A linked-list is an ordered sequence of elements of the same type where each element has a pointer to its immediate neighbouring element(s)

```
struct ListElt{
  int value;
  ListElt* prev;
  ListElt* next;
};
```



- An element can be inserted anywhere in the list by changing only its neighbouring elements, i.e. their `next` and `prev` pointers.
- This is an advantage over a `vector` where insertion requires all the elements after it to be moved in memory.

std::list

```
#include <list>
std::list<int> l;
std::cout << l.size(); // 0
l.push_back(5);
l.push_front(10); // Not possible with a vector
std::cout << l.front(); // 10
std::cout << l.size(); // 2
std::list<double> myReals{3.14159, M_E, -1.7};
```

std::list ctd

- Random-access is no longer allowed, since that would require iterating through the list for n elements
- The iterator approach seen earlier is still valid:

```
int f(const std::list<int>& a){
  int sum = 0;
  std::list<int>::const_iterator aIter;
  for( aIter = a.begin() ; aIter != a.end() ; ++aIter ){
    sum += *aIter;
  }
return sum;
}
```

The similarity of this function to the `std::vector` version will become important later.

Iterators overview

- In general, iterators resemble a safer version of pointers:
  - Dereferencing gives an element in a container
  - Incrementing/decrementing gives the next/previous element in a container
- Iterators only apply to one specific container and, if that container is altered, the iterator is (probably) invalid
- For example, adding an element to a list will probably mean that all previous iterators obtained from the list are now no longer correct
- Typedef can be used to save typing, as in:
  `typedef std::list<std::string>::const_iterator ListIter;`

auto

- Even having to define **typedef**s can become tiresome.
- We therefore have the **auto** keyword, which defines a variable to be whatever type the right-hand side happens to be.
- For example:

```
std::list<std::string> myList;
for( auto listIter = myList.begin() ; listIter != myList.enc
    ; ++listIter ){
  std::cout << *listIter << std::endl;
}
```

More auto and for-loops

- The previous example can be shortened further:

```
std::list<std::string> myList;
for( auto listItem : myList ){
  std::cout << listItem << std::endl;
}
```

- Modifiers can be used with `auto`:

```
std::vector<int> myVector;
for( auto& value : myVector ){
  value *= 2;
}
```

  to double all members of `myVector`

- Or:

```
std::vector<int> myVector;
for( const auto& value : myVector ){
  value *= 2; // Gives compile error
  std::cout << value << std::endl; // OK
}
```

auto notes

- `auto` can be used anywhere that the compiler can deduce the type of the variable.
- In practice, I suggest it is used only for short-lived variables or ones where the developer doesn't really need to know the full gory details of the type.
- In most cases, I find that knowing for certain what type a variable is helps with code reading and debugging.
- For example, where the variable type may affect accuracy (`float`/`double`) or performance (`std::list<int>` versus `std::vector<int>`)
- In most cases, I ignore `auto` as I believe knowing what the compiler is doing for you to be important for scientific computing.

Distance between iterators

- It is possible to determine the distance between two iterators as:

```
std::list<int>::iterator first10 = std::find(a.begin(),
    a.end(), 10);
size_t posn = std::distance(a.begin(), first10)
```

- However, for iterators other than those into containers where random access is allowed (e.g. `std::vector`), this is implemented using `iter--`, and is therefore slow $O(N)$.
- For random-access iterators, subtraction is overloaded, i.e.:

```
std::vector<int>::iterator first10 = std::find(a.begin(),
    a.end(), 10);
size_t posn = first10 - a.begin();
```

- Subtraction is not overloaded for non-random-access so that you don't try to use an expensive operation by mistake.

Outline

Algorithms

- There are many operations that are applicable to many different container types: `find`, `equal`, `count`, `sort`
- These exist as generic algorithms within C++

```
#include <list>
#include <algorithm>
std::list<int> myList;
std::list<int>::iterator posnOf10 =
     std::find(myList.begin(), myList.end(), 10);
if( posnOf10 != myList.end() ){
    *posnOf10 = 11;
}
```

- The preceding replaces the first occurence of 10 (if any) in the list by 11.
- An invalid iterator is usually represented by `myContainer.end()`, being one element past the end of the container
- This algorithm can be applied to any suitable container, e.g. `vector`, `deque` etc.

Algorithms ctd

- The iterator points to a single element of the container.
- The returned iterator does *not* contain any information about the algorithm that produced it.
- In order to find the next element equal to 10, the following is required:

```
posnOf10 = std::find(++posnOf10, myList.end(), 10);
```

  which advances `posnOf10` to the next element, and calls another `find` starting from that element.
- Advancing `posnOf10` off the end of the container makes it equal to `myList.end()`

General algorithms

- A sort algorithm could also be applied to a `vector`:

```
std::vector<unsigned int> a;
std::sort(a.begin(), a.end());
```

- By default, this will use the $<$ comparison.
- To use a different comparison, you can supply a different function:

```
bool sortByLastDigit(unsigned int i, unsigned int j){
  return ( (i % 10) < (j % 10) );
}

std::sort(a.begin(), a.end(), sortByLastDigit);
```

- Many algorithms within the STL are customisable in this fashion.

Ordering on classes

- If you want to apply find and sort algorithms to containers of your own classes, you will need to define an ordering.
- This only requires overloading `operator<`

```
class Rational{
  bool operator<(const Rational& a)const{
    return (num/(double)denom) < (a.num / (double)a.denom);
  }
};

std::vector<Rational> a;
std::sort(a.begin(), a.end());
```

will sort the `vector` of Rationals correctly.

More algorithms

Counting the number of 'e's in a string:

```
std::string gadsby;
size_t numberEs = std::count(gadsby.begin(), gadsby.end(),
    'e');
if( numberEs != 0 ){
    std::cout << "Wright did fail." << std::endl;
}
```

- The preceding example works because a **std::string** is a container of **char**s

Outline

Map

- A map is a mapping from one set of values to another.
- The types of the key (from) and value (to) can be anything (so long as the key-type has an ordering)

```
#include <map>
#include <string>
std::map<std::string, int> age;

age["Fred"] = 5;
age["Simon"] = 10;
std::cout << age["Fred"] << std::endl; // outputs 5
std::map<std::string, int>::const_iterator tom =
    age.find("Tom");
// Now tom == age.end()
tom = age.find("Simon");
std::cout << tom->second << std::endl; // Outputs 10
```

- This works because there is an ordering on `std::string`
- The elements of a `std::map` are ordered by their key.
- This only usually matters when using iterators of maps.

Map

A map can be initialized using an initializer list:

```
std::map<int,double> b{ {1, M_PI}, {2, M_E}, {6, 9.80665} };
```

which is using uniform initialization for individual `std::pair` elements, and an initializer list overall.

Map ctd

- Note that element access for a map is not a constant member fn:

```
std::map<std::string, int> myMap;
int a = myMap["Fred"];
```

will result in `a` being undefined, and `myMap` has a new element `Fred`

- In general, the element is created using the default constructor, but for an `int` this does not do anything.
- If you wish to test whether a particular value is in the `map`, use:

```
if( myMap.find("Fred") != myMap.end() )
```

or

```
if( myMap.count("Fred") == 1 )
```

- If you have a `const std::map<>` then you are prevented from using `[]` access on it by const-ness.

Unordered map

- An `std::unordered_map` uses a hash function to map a `Key` type to a `size_t` value.
- Items with identical hashes are placed into a single bucket (although different keys are still kept distinct).
- This results in (average) constant time complexity for search, removal, and insertion of items.
- (as compared to a `std::map` having logarithmic complexity for these).

More container types

- Other container types in C++ are:
  - `deque`: Double-ended queue: Allows efficient `push_front` and `push_back`, and efficient subscripting
  - `set`: Set of values (similar to map where the value type is irrelevant)
  - `multimap`: Multi-valued map, i.e. mapping from a single key to a set of values