Part XI

Algorithms and Templates

Outline

Computational Complexity

- Various operations have computational complexity guaranteed by the standard
- These are usually upper-bounds on the maximum complexity required for that operation
- For example, a sort algorithm on a `vector` can be carried out in approximately $N \log(N)$ operations.
- This is guaranteed by the standard, so any C++ implementation must implement algorithms with at most this complexity
- However, implementations may vary as to how efficiently they carry out the actual sort.

More algorithms

- The STL has various sorting algorithms which can act on suitable containers:

```
std::vector<int> a(4);
a[0] = 1; a[1] = 3; a[2] = 4; a[3] = -9;
std::sort(myVector.begin(), myVector.end());
// Now a contains (-9,1,3,4)
```

  would sort the whole of `myVector`.
- This takes on average $N \log N$ comparisons, and in the (rare) worst case: N^2

Sorting ctd

- If you want guaranteed complexity $N \log N \log N$ then use `stable_sort`
- This also preserves the order of elements that compare to be equal.
- This may appear irrelevant but if your elements are `std::pair<int, double>` and you order based on the first integer, then the associated `double` may not be the same for identical integers.

String algorithms

- Since strings are also containers (of `char`s), many algorithms can be applied to these as well

```
std::string a("The quick brown fox jumped swiftly
              over the lazy dog.");
std::sort(a.begin(), a.end());
std::cout << a << std::endl;
```

will output

`.Tabcddeeeeffghhiijkllmnoooopqrrsttuuvwwxyyz`

Member algorithms

- Some algorithms are implemented as member functions of their containers:

  ```
  std::list<int> a;
  a.remove(10);
  a.sort();
  ```

  would remove all 10s from the list, and then sort it.
- This is a member function because the `sort` operation for a `list` can be written in terms of modifying the list elements' next/previous pointers, rather than copying/moving the elements themselves.
- The `std::sort` function requires random-access iterators, which a `list` does not have.

Copying algorithms

- You can copy from one container to another:

```
std::vector<int> a(9);
std::vector<double> b(9);
std::copy(a.begin(), a.end(), b.begin());
```

  which assumes that `b` has at least as many elements as `a`
- In order to copy part of a container, you could use:

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

- would copy all elements up to (but not including) the first occurence of 10 from `a` into `b`

Transformation algorithm

- You may wish to create a new container by applying a function to another one:

```
std::vector<int> a;
std::vector<double> b;
std::transform(a.begin(), a.end(), b.begin(), sqrt);
```

- This assumes that `b` is at least as big as `a`.

find_if and predicate

- We can use the **find_if** algorithm with a function:

```
bool lessThan10(int x){ return (x < 10); }
aIter = std::find_if(a.begin(), a.end(), lessThan10);
```

  will return an iterator pointing to the first element less than 10.
- However, this is wasteful in writing functions with different names.
- C++ allows us to define a function inline:

```
aIter = std::find_if(a.begin(), a.end(),
                     [](int x){return (x < 10);}  );
```

  where the [] syntax indicates a "lambda" function.
- The lambda can also pick up variables from the current scope:

```
int valueToFind = getIntegerFromUser();
aIter = std::find_if(a.begin(), a.end(),
          [valueToFind](int x){return (x < valueToFind);});
```

- Variables that need to be captured by the lambda are put in the square brackets.

Lambda functions ctd

- We could also use a different algorithm:

```
std::transform(a.begin(), a.end(), b.begin(),
  [cutoff](int x){return (x < cutoff) ? x : 0;}
  );
```

  which copies `a` into `b`, except that it replaces values larger than `cutoff` with zeros.

- Or:

```
auto sortUnit = [](int a, int b){return (a % 10 < b % 10);};
std::sort(a.begin(), a.end(), sortUnit);
```

  to sort `a` according to the units-digits of its elements.

- In each case, the arguments and return-type of the lambda must be as expected by the algorithm in which it is being used.

Lambda functions ctd

- In some cases lambda functions can make the code more compact and easy to read.
- In some cases they can make it substantially more complicated to read or possibly less efficient.
- A few extra syntax notes:
  - The capture list can be given as
    - `[&]`: all variables captured by reference, or
    - `[&, a, b]`: captures all local variables other than `a` and `b` by reference, or
    - `[=]`: all variables captured by value, or
    - `[=, &a, &b]`: captures all local variables by value except for `a` and `b` which are captured by reference.
  - If there are no parameters to pass to the lambda function, the `()` can be omitted.
  - Parameter values are captured at the point where the lambda function is created.

Yet more algorithms

- Other useful algorithms available in C++ include:
  - `count_if`: count all sequence elements that satisfy a condition
  - `equal`: Compare two sequences
  - `merge`: Merge sequences
  - `max_element`: Find maximum element in a sequence
- If you need to use these, then any decent C++ reference manual should give you the relevant syntax
- There are many more algorithms than I have listed here. However, they all apply to a wide range of containers, and are fully user-customisable (e.g. sorting criterion).

Outline

Templates

- You may have wondered: Why the Standard *Template* Library?
- Many of the algorithms can be applied to `vectors`, `lists`, `maps`
- As long as a container has certain properties, then a generic algorithm can be applied to it
- We can write a general piece of code that applies to any container, and just needs the actual container type to be specified.
- This is essentially what a C++ template is.

Outline

Description of problem

- Suppose we want to sum all the elements of a `std::vector`
- (Ignore the fact that this can be done using a built-in algorithm)
- We want to start with zero, and add successive elements to it
- We should not simply cast to a specific default type, because different types will behave differently with regard to overflow, precision, etc.
- We could write a separate version for every type we want to sum, but this is error-prone
- We could use macros (see Practical 6), but we would still need to add an extra line for every type we wanted to use (also, macros are evil)

Template code

```
template<typename T>
T sumVector(const std::vector<T>& v){
  T sum = 0;
  for(size_t i=0 ; i < v.size() ; i++){
    sum += v[i];
  }
  return sum;
}

std::vector<int> a;
int s = sumVector(a);
```

- Whenever a templated-function is used, the compiler checks the types of the parameters and then matches these to the templated definition
- It tries to deduce the template-parameters (`T` in the above)
- When the parameters have been deduced from the calling syntax, a version of the function is *instantiated* using these parameters.
- The generated function is then compiled, and stored in the current object file being generated

Specialisation of templated function

- For floating-point values, the order in which we sum them is important, so we may wish to use a fully specialised version

```
template<>
double sumVector<double>(const std::vector<double>& v){
  // Use Kahan summation to reduce round-off error
  return sum;
}
```

- Now when we pass a `std::vector<double>` to `sumVector`, this version of the function will be used instead of the generic form on the previous slide.
- The `template<>` construct says that this is a templated function, but that there are no free template parameters.
- The `sumVector<double>` says that this is `sumVector` with the template parameter `T = double`.

Template parameters

- In the previous code, the template-parameter `T` was a `typename`
- Template-parameters can also be:
  - Integral types (i.e. int, char, enums, etc.)
  - Function-pointers
  - Templated typenames (advanced users only)
- They cannot be:
  - Floating point values
  - Strings

Integer as template parameter

- Very simple example:

```
template<int V>
int addInt(int a){
  return a + V;
}
```

- This could be of use when using STL algorithms that require a function of the form `int f(int)`

```
std::transform(a.begin(), a.end(), b.begin(), addInt<5>);
```

  to add 5 to every element of `a` and put it into `b`.
- Otherwise, you might have to implement separate `add4`, `add5` functions.
- (Of course, a simpler approach would be to use a lambda function, although perhaps the compiler might not be able to optimize it as well.)

Outline

Templated classes

It is also possible to template classes

```
template<typename T>
class MyArray{
public:
  MyArray(unsigned int);
  T operator[](size_t)const;
private:
  T* data;
};
template<typename T>
MyArray<T>::MyArray(unsigned int s){
  data = new T[s];
}
template<typename T>
T MyArray<T>::operator[](size_t i)const{
  return data[i];
};
```

Within a class definition, or a member function definition, the class-name always refers to the version with the current template arguments, unless otherwise specified.

Template instantiation

- For most uses of templates, you will always need to make sure that a definition of a templated class or function is available within the current source-file being processed.
- Thus, when a compiler decides that it needs a particular version of a class/function with particular template arguments, it can instantiate one immediately.
- This ensures that all necessary code is compiled as necessary.
- Therefore, most template definitions should be put into header `.H` files, and included as necessary.
- If a templated definition were only present in a `.C` file, but used in another source file, the compiler would not be able to see the definition to create the correct version.

Class template default parameters

- Class templates are allowed to have default parameters:

```
template<int SIZE, typename T = double>
class realVector{
  private:
    std::array<T,SIZE> data;
};

realVector<10> v;
realVector<10, float> vSingle;
```

- As with default function parameters, once one is specified, all subsequent parameters must also be specified.
- Templated functions are *not* allowed to have default template parameters as this would end up conflicting with overloading.

Partial specialisation

- It is also permitted to define partial specialisations of classes (not functions), where some template parameters are specified:

  ```
  template<typename X, typename Y>
  class A{ /* Code */ };

  template<typename Z>
  class A<Z,Z>{};
  ```

  defines a generic form for a class with two template parameters, but specialises for the case where the template parameters are the same
- It is not necessary to make different specialisations of the class similar in any way whatsoever, but you would nearly always do so for reasons of clarity.

Templated members of templated classes

- If you have a templated class, you may wish its member functions also to depend on different template parameters:

```
template<int SIZE, typename T = double>
class realVector{
  template<typename S>
  realVector operator*(const S&)const;
};

template<int SIZE, typename T>
template<typename S>
realVector<SIZE,T>
realVector<SIZE,T>::operator*(const S& s)const{
  // Create new vector multiplied by s
}
```

Compile-time computation

- Templates can also be used to carry out computations at compile-time.
- You will see an example of this in the practicals.
- This relies on the fact that in

```
template<int N>
struct Double{
  static const int result = 2*N;
};
```

the value of `result` can be computed at compile-time.

Advanced template constructs

- It is possible to do some very advanced compile-time computations with templates
- These are usually used to allow the compiler to make optimizations which it would have not otherwise had sufficient information to make.
- For example, the Boost libraries have a templated function `boost::math::pow<5>(a);` to compute the fifth power of `a`.
- The templating allows the compiler to see this as `(a*a*a*a*a)` or maybe even `( (a*a) * (a*a) * a )`
  which may allow it to make optimizations that would not have been possible with a function `pow(a, 5)`.