C++ for Numerical Programming - lectures 11-13

Martin Robinson

2016

Recall that you can write generic functions using templates

```
template <typename T>
T get_min (T a, T b) {
    if (a<b) {
        return a;
    }
    return b; // When a>=b
}

void main() {
   std::cout << get_min<int>(10,-2) << "\n";
   double ans = get_min<double>(22.0/7.0, M_PI);
}
```

Template specialisation

You can provide specialisations for template arguments. Explicit (full) specialisation is when *all* template arguments are specialised

```
template <typename T>
bool is_int(T a) {
    return false;
}

template <>
bool is_int<int> (T a) {
    return true;
}
```

Class Template

It is possible to template both functions and classes

```
template<unsigned DIM>
class DoubleVector {
   double mData[DIM];
   public:
   double& operator[](int pos) {
      assert(pos<DIM);
      return(mData[pos]);
   }
};

int main() {
   DoubleVector<5> a;
   a[0] = 10;
   a[1] = 11;
   std::cout << a[0]+a[1] << "\n";
   a[5] = 0; //Trips assertion
```

Partial specialisation

Class templates can also be *partially* specialised

```
// primary template
template<class T1, class T2, int I>
class A {};

// partial specialization where T2 is a pointer to T1
template<class T, int I>
class A<T, T*, I> {};

// partial specialization where T1 is a pointer
template<class T, class T2, int I>
class A<T*, T2, I> {};

// partial specialization where T1 is int, I is 5,
// and T2 is a pointer
template<class T>
class A<int, T*, 5> {};
```

For example: using partial specialisation to determine if class `T` is a pointer

```
template <typename T>
struct is_pointer { static const bool value = false; };

template <typename T>
struct is_pointer<T*> { static const bool value = true; };

void main(void) {
    assert(is_pointer<int*>::value);
}
```

This, and many more, already implemented for you in C++ `type_traits` standard library

Template instantiation

Before you use, or *instantiate*, a templated class the compiler has nothing to compile. The templated class is simply a generic *template*.

```
template <typename T>
struct Vect3 {
    T x,y,z;
    T norm();
};
```

For a normal class you would probably define the `norm` function in a `.cpp` file which is compiled separately. For a templated class the definition must occur in a header (e.g. `.hpp`) file

```
template <typename T>
T Vect3<T>::norm() {
    return sqrt(pow(x,2) + pow(y,2) + pow(z,2));
}
```

When you instantiate a class, say in a main .cpp file, the compiler fills in the template arguments and can then compile the class

```
void main(void) {
    Vect3<double> v;
    Vect3<float> v;
    Vect3<int> v;
}
```

Note, the above will generate three different classes.

More on the STL

This is a set of commonly used patterns which can be re-used for different types of objects

- Containers. e.g. random access vectors, linked lists
- Algorithms. e.g. sorting
- Iterators
- Special containers e.g. Queues and maps

Containers

Sequence containers:

- `std::array`
- `std::vector`
- `std::deque`
- `std::forward_list`
- `std::list`

Container adaptors:

- `std::stack`
- `std::queue`
- `std::priority_queue`

Associative containers (unordered versions of each as well):

- `std::set`
- `std::multiset`
- `std::map`
- `std::multimap`

std::map example

- The `map` template class provides the machinery to make a mathematical map
- This lets us recall the value to which a particular key maps, rapidly
- The internal organisation of a map relies on the ability to compare the values of keys
- Many plain data types (`int`, `double`) have obvious comparison functions. `std::string` types can be compared lexicographically.
- For more complicated keys, you need to write and add a definition of the 'less than' operator

```
std::map<std::string, int> Phonebook;

Phonebook["Joe"] = 83511;
Phonebook["Sandy"] = 15208;
Phonebook["Sam"] = 10666;

std::cout << "Phonebook[Joe]=" << Phonebook["Joe"] << "\n\r
std::cout << "Map size: " << Phonebook.size() << "\n";

for( auto ii=Phonebook.begin(); ii!=Phonebook.end(); ii++)
{
   std::cout << (*ii).first << ": " << (*ii).second << "\n'
}
assert(Phonebook.count("Laura") == 0);
```

std::set example

This class lets us do comparison (lexicographical) on 2D points

```
class Point2d
{
   int x, y;
public:
   Point2d(int xval, int yval)
   {
      x=xval; y=yval;
   }
   bool operator<(const Point2d& rOther) const
   {
      if (x < rOther.x) return true;
      if (x > rOther.x) return false;
      return (y < rOther.y);
   }
};
```

Thus we get key comparison (two-dimensional point comparison) for use in a set

```
int main()
{
   std::set<Point2d> points;
   Point2d origin(0,0);
   points.insert(origin);
   points.insert(Point2d(0,1));
   points.insert(Point2d(1,0));
   points.insert(Point2d(0,0)); //No different from origin
   std::cout<<points.size()<<"\n";
   std::cout<<points.count(Point2d(0,0))<<"\n";
}
```

Algorithms

- The STL provides a large list (I stopped counting at 80) of algorithms that operate on all or some of the containers.
- Example using `std::sort`:

```
#include <algorithm>
...
std::vector<int> squares_mod_10;
for (int i=0; i<10; i++){
   squares_mod_10.push_back( (i*i) % 10 );
}
// [0, 1, 4, 9, 6, 5, 6, 9, 4, 1]

std::sort (squares_mod_10.begin(), squares_mod_10.end());
// [0, 1, 1, 4, 4, 5, 6, 6, 9, 9]

for (auto i: squares_mod_10) {
   std::cout << i << "\t";
}
```

Can use algorithms for many of your loops (if you wish).

```
#include <algorithm>
#include <numeric>
#include <iterator>
...
std::vector<int> squares_mod_10(10);
std::iota (squares_mod_10.begin(), squares_mod_10.end(), 0
std::transform (squares_mod_10.begin(), squares_mod_10.end
                squares_mod_10.begin(),
                [](int i) { return (i*i) % 10; });
// [0, 1, 4, 9, 6, 5, 6, 9, 4, 1]

std::sort (squares_mod_10.begin(), squares_mod_10.end());
// [0, 1, 1, 4, 4, 5, 6, 6, 9, 9]

std::ostream_iterator<int> out_it (std::cout,", ");
std::copy (squares_mod_10.begin(), squares_mod_10.end(), ou
```

Summary

- STL concentrates on frequently used design patterns and it's good to know the patterns
- Using algorithms, even simple ones like `std::accumulate` and `std::transform`, can make your code more readable and easy to understand
- STL functionality is highly-optimised by compiler writers to give complexity assurances and a low memory-footprint
- C++17 gives parallel functionality to many of the standard algorithms (much easier that writing your own parallel version of `std::accumulate`!)

Lecture 12 — Linking Third-party Libraries

One of the advantages to using C++ is the wide availability of high performance libraries for scientific computing

Assuming you have installed a third-party library on your system, you can use it by including and linking against the relevant files

The `-I` flag for `g++` specifies the directory where the header files (`.h`) are installed. These are the files that you include in your code using `#include`.

```
g++ -I/usr/include program.cpp
```

The `-L` flag specifies where the library files (`.a`, `.so` or `.dll`) are installed. Use the '-l' flag to specify a library to link against

```
g++ -I/usr/include -L/usr/lib -llibrary program.cpp
```

The above example could link against '/usr/lib/liblibrary.a'.

For example, you might have a `Makefile` that looks like:

```
INCLUDE_DIRECTORY = /usr/include
LIBRARY_DIRECTORY = /usr/lib

program: program.o
    g++ -o program  program.o -L$(LIBRARY_DIRECTORY) \
                -lthird_party_library
program.o: program.cpp
    g++ -I$(INCLUDE_DIRECTORY) -c program.cpp
```

Example: Boost MPI (`boost.org`)

- Boost is a collection of useful C++ libraries to make your life easier.
- It has libraries for special math functions, ODE integration, linear algebra, random number sampling, to name a few.
- Very well regarded, many Boost libraries make their way into the C++ Standary Library

For example, Boost has a user-friendly wrapper for parallel programming using Message Passing Interface (MPI).

```
#include <boost/mpi/environment.hpp>
#include <boost/mpi/communicator.hpp>
#include <iostream>
namespace mpi = boost::mpi;

int main()
{
  mpi::environment env;
  mpi::communicator world;
  std::cout << "I am process " << world.rank()
    << " of " << world.size()
    << "." << std::endl;
  return 0;
}
```

Example Makefile for Boost MPI

```
BOOST_INC = /usr/include
BOOST_LIB = /usr/lib

printRank: printRank.o
    /usr/bin/mpic++ -o printRank  printRank.o \
        -L$(BOOST_LIB) -lboost_mpi -lboost_serialization
printRank.o: printRank.cpp
    /usr/bin/mpic++ -I$(BOOST_INC) -c printRank.cpp
```

Suppose we have lines of code that read

```
bigger_vector = a_vector;
smaller_vector = a_vector;
```

where `smaller_vector`, `a_vector` and `bigger_vector` are vectors that have been declared as having 1, 2 and 3 elements respectively.

The assignment operator is expecting that the size of its input vector (on the right-hand side) matches the object which it is assigning to.

There are clearly errors here – the current implementation will attempt to add too much data into `smaller_vector`.

What should the program do when the sizes do not match?

The answer is – “It depends”.

It’s good to have a hierarchy of errors

Level 1: If the error can be fixed safely, then fix it. If need be, warn the user.

Level 2: If the error could be caused by user input then throw exception up to calling code, since the calling code should have enough context to fix the problem.

Level 3: If the error should not happen under normal circumstances then trip an assertion.

Exceptions are a compromise between carrying on regardless and stopping completely.

Exceptions require use of the keywords `try`, `throw` and `catch`

`try` tells the code to execute some statements

`throw` identifies an error

`catch` attempts to fix the error

We will use the example of assigning to a vector of the wrong length using the overloaded = operator for vectors

When assigning to a longer vector we will treat it as a *Level 1* error - pad the extra entries with zeroes and warn the user. When assigning to a shorter vector we will treat it as a *Level 2* error and throw an exception, because data would be lost otherwise.

When an error occurs we want the code to “throw” two pieces of information

1. A summary
2. A description of the error

We will write a class `Exception` to store these two pieces of information, and with the ability to print this information when required

The file `Exception.hpp` may be written

```
#ifndef EXCEPTIONDEF
#define EXCEPTIONDEF
#include <string>
class Exception
{
public:
    std::string problem, summary;
    Exception(std::string sum, std::string prob);
    void DebugPrint();
};
#endif
```

The file `Exception.cpp` may be written

```
#include "Exception.hpp"
Exception::Exception(std::string sum, std::string prob)
{
    problem = prob;
    summary = sum;
}

void Exception::DebugPrint()
{
    std::cerr << "** Exception ("<<summary<<") **\n";
    std::cerr << "Problem: " << problem << "\n\n";
}
```

Here's the new assignment operator (2 slides)

```
Vector& Vector::operator=(const Vector& rVec)
{
    // if rhs vector is too long then throw
    // if rhs vector is too short, assume missing entries a
    if (rVec.mSize == mSize)
    {
        for (int i=0; i<mSize; i++)
            mData[i] = rVec.mData[i];
    }
    else if (rVec.mSize > mSize)
    {
        throw Exception("length mismatch",
    "vector assignment operator - vectors have different le
}
```

```
    else //if (rVec.mSize < mSize)
    {
        for (int i=0; i<rVec.mSize; i++)
        mData[i] = rVec.mData[i];
        for (int i=rVec.mSize; i<mSize; i++)
            mData[i] = 0.0;
        std::cout << "vector assignment - copied vector too
        std::cout << " and has been extended with zeroes\n"
    }
    return *this;
}
```

We may now test the exception written in our overloaded assignment operator for vectors

```
Vector smaller_vector(1);
Vector a_vector(2);
Vector bigger_vector(3);
//This produces a warning
bigger_vector = a_vector;
//This produces an exception
try
{
    smaller_vector = a_vector;
}
catch (Exception& err)
{
    err.DebugPrint();
}
```

Tip: test first

- Test driven development means that you always start with the code for a test (not the code itself)
- Choose the simplest piece of functionality which you want to implement first
- Make a test before you make the implementation (it won't compile and it won't pass)
- Write the missing functionality until the test passes
- Always check that all tests pass as you add new functionality (then you know as soon as the program gives different behaviour)