# Part XII

## Expression templates and more programming comments

# Outline

# Expression templates - overview

- To demonstrate the power of templates within C++, we will look at an advanced example of their use
- Very little in the way of complete code will be given. For more details, see C++ Templates (Vandevoorde and Josuttis)

# Array operations

- Within C++ it is fairly easy to define your own `vector` class that allows for arithmetic operations:

```
Vector a(10), b(10), c(10);
/* Fill a and b with suitable data */
c = 2.3*a + 4.5*b + a*b; // Assume elt-wise multiplication
```

- However, the preceding is not efficient. It effectively does:

```
Vector tmp1(10) = 2.3*a;
Vector tmp2(10) = 4.5*b;
Vector tmp3(10) = tmp1 + tmp2;
Vector tmp4(10) = a*b;
Vector tmp5(10) = tmp3 + tmp4;
c = tmp5;
```

## What do we want?

- Ideally, the line

```
c = 2.3*a + 4.5*b + a*b;
```

should be replaced by something equivalent to

```
for(size_t i=0 ; i < a.size() ; i++){
  c[i] = 2.3*a[i] + 4.5*b[i] + a[i]*b[i];
}
```

- This can be arranged if we use expression templates, which effectively encode an entire expression as a template parameter.

# Generic Vector

- We create a vector class

```
template<typename T, int SIZE,
         typename Internal=SimpleVector<T,SIZE> >
class Vector{
public:
  T operator[](size_t i){return data[i];}
  const Internal& internalType()const{return data;}
private:
  Internal data;
};
```

whose internal storage is generic, but is a simple C-array wrapper by default.

- The internal type must have an `operator[]` but can otherwise be anything.

## Adding two Vectors

- If we want to add two Vectors, we can make the result another class:

```
template<typename T, typename Op1, typename Op2>
class VectorSum{
public:
  VectorSum(Op1 a, Op2 b) : op1(a), op2(b) {}
  T operator[](size_t i)const{
    return op1[i] + op2[i];
  }
private:
  const Op1& op1;
  const Op2& op2;
};
```

# Overloading the + operator

```
template<typename T, int SIZE, typename X, typename Y>
Vector<T,SIZE,VectorSum<T,X,Y> > operator+(
    const Vector<T,SIZE,X>& a,
    const Vector<T,SIZE,Y>& b){
return Vector<T,SIZE,VectorSum<T,X,Y> >
   (VectorSum<T,X,Y>(a.internalType(), b.internalType()));
}
```

# Assignment constructor for Vector

Now we make sure that a Vector can be assigned from any Vector type:

```
template<int SIZE, typename T, typename InternalType>
template<typename S>
Vector<SIZE,T,InternalType>&
    Vector<SIZE,T,InternalType>::operator=(const
    Vector<SIZE,T,S>& s){
   for(int i=0 ; i < SIZE ; i++){
     data[i] = s[i];
   }
}
```

# And it all works...

```
Vector<3,double> a,b,c,d,e;
a = b + c + d + e;
```

will not allocate any temporaries, and will be nearly as efficient as plain C code.

# Doing it properly...

- We also need to overload all other operators - * / += -= etc.
- Need to allow for scalars as appropriate, possibly creating a trivial scalar wrapper that behaves like a Vector.
- Overload `operator*` for all combinations of `Vector` and `Scalar`.
- Can even overload `sin`, `cos` etc.
- We now get a Vector class that can evaluate expressions such as:

```
Vector<3,double> a = sin(b) + 6.7*c + d/9 - pow(e,2);
```

as efficiently as if you'd written an explicit loop over all elements.

# Outline

# Namespaces

- A namespace is just another way of lumping a set of functions/classes together under a general heading
- The only one you've seen so far is `std::`
- External libraries tend to use namespaces to separate their functions from other libraries/users

```
namespace MyMatrixLibrary{
  Matrix transpose(const Matrix&);
}
```

- This distinguishes this `transpose` function from others since you now have to refer to it as `MyMatrixLibrary::transpose`.

# Importing namespaces

- It is possible to import a namespace into the global namespace:

```
using namespace MyMatrixLibrary;
```

- Now, the function can be referred to as `transpose`
- Since this could cause clashes of functions/classes, I recommend not using `using`
- This extends to never using `using namespace std;` which also reminds you that these functions/classes are contained in `std::`
- It also saves clashes if you want to call a variable `vector` for example...

# Outline

# Exceptions

- In C and Fortran, indicating that an error occured within a function often requires an error code:

```
int invertMatrix(const Matrix& A, Matrix& AT);
```

- or even setting a global variable `err`
- We would prefer a basic call to look like:

```
Matrix invertMatrix(const Matrix& A);
```

- The approach used in C++ involves exceptions:
- The exception is a simple struct/class
- It can contain information about the type of error if necessary.
- The principle behind these is that the calling function has more information about the situation than the inversion function, and can therefore deal more appropriately with it.
- The exception will propagate up the call-chain until it finds a matching `catch()`
- NOTE: C++ exceptions are NOT the same as floating-point exceptions.

# Exception example

```
struct SingularMatrix{
};

Matrix invertMatrix(const Matrix& a){
 if( det(a) == 0 ){
   throw SingularMatrix();
 }
}
int main(void){
try{
  a = invertMatrix(eqnSystem);
}
catch(SingularMatrix& e){
  std::cout << "eqnSystem is singular" << std::endl;
}
}
```

# Standard exceptions

- Various exceptions can be thrown by C++ operators and STL library functions
- For example, when out of memory:

```
try{
  int* a = new int[10000];
}
catch(std::bad_alloc){
  std::cout << "Out of memory" << std::endl;
}
```

- If uncaught, the exception will propagate to the top of the stack and cause the code to abort
- The C++ run-time may give a useful indication of the exception.

# More exceptions

- An exception is caught by the first `catch` construct that matches its type.

```
struct MatrixError {};
struct SingularMatrix : MatrixError{};

try{
  a = invertLargeMatrix(b);
}
catch(std::bad_alloc){
  // Do something
}
catch(MatrixError& e){
  // Some matrix error occured
}
```

- Even if the specialised `SingularMatrix` is thrown, it still matches the general `MatrixError` type due to inheritance.
- Exceptions should only be used to deal with exceptional behaviour; they should not be part of the expected execution of your program.

# Outline

# Who needs const?

- Clearly, `const` is useful, mostly for protecting the programmer from themselves.
- However, sometimes it can be too strict. For example:

```
class calcFn{
  public:
    double doCalc(double x)const;
  private:
    double calcData;
};
```

  where our expensive calculation has associated constant data.
- If we want to introduce a cache for the last value of `x` to be used, and the result for that `x`, then what can we do?
- We would presumably need to set the values of `lastX` and `lastResult` in `doCalc(x)`, but it's a `const` function...

# Slightly hacky approach...

- One possibility would be a pointer to a cache class:

```
class calcFn{
  CacheClass* myCache;
  // Other members here
};
```

- This works because a const member function only guarantees that `myCache` does not change, but we can still change the value of the object pointed to by `myCache`.

# mutable

- The answer is to use **mutable**, which allows values to change, even in a **const** member function:

```
class calcFn{
  private:
    mutable double lastX;
    mutable double lastF;
  // Other members here
};
```

- This feature could obviously be abused horrendously, but allows situations like the current one without breaking encapsulation by having the cache outside the class.

# Input

- Getting input from a file is hard within C++ if you want anything more complex than space-separated numbers.
- One answer is `libconfig`: `www.hyperrealm.com/libconfig`
- Allows for input files like:

```
application:
{
  window:
  {
    title = "My Application";
    size = { /* width */ w = 640; /* height */ h = 480; };
    pos = { x = 350; y = 250; };
  };
}
```

# Outline

# Good programming practice

- Being able to program well is somewhat of an art or skill
- It is best to use relatively simple-looking approaches, rather than obscure C++ constructs
- Names should be descriptive, and use the correct part of speech
  - Variables - usually nouns
  - Functions - usually verbs

# Error handling

- In scientific computing, you should aim to check for errors as soon and as often as possible.
- Some error checking may be computationally expensive, in which case you can skip it (but remember that you have omitted it)
- You could use `#ifdef DEBUG` to omit checking when running optimized code.
- For example, bounds checking can be expensive, but if you remove it remember that you may get seg-faults or odd behaviour if you write outside the array
- In general, if an error occurs, alert the user and abort immediately.

# Debugging

- The GNU debugger **gdb** is capable of understanding C++ classes
- Member data and also that from base-classes is printed in an easy-to-read fashion
- It also (as from version 7.0) understands STL classes such as `vector` etc.

```
(gdb) print myVector
$1 = std::vector of length 10, capacity 10 = {0, 2, 4, 6,
     8, 10, 12, 14, 16, 18}
(gdb) print myMap
$1 = std::map with 3 elements = {
  [1] = 0.90000000000000002,
  [4] = 5,
  [6] = 23.449999999999999
}
```

# C++ 17 / C++ 20

- So far we have been dealing with the C++14 standard (broadly speaking).
- However, the C++17 standard was finalised in December 2017, and the C++20 standard had its final draft in September 2020.
- These introduce several new features to the language
- Many compilers support some of these features, but it may be a year or two before complete support for C++20 is available
- For a rather technical overview of what features were introduced at each standard, see `https://gcc.gnu.org/projects/cxx-status.html`

# Programming Quotes

- "Premature optimization is the root of all evil" - Donald Knuth
- "Everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will you ever debug it?" - Brian Kernighan

# Further reading

- Code Complete (Steve McConnell)
  (Useful for code-designing hints)
- `www.oodesign.com`
- Solid Code (Marshall, Bruno)
- C++ Templates (Vandevoorde & Josuttis)
- Modern C++ Design (Alexandrescu)
- Programming Pearls, (Jon Bentley)