Scientific Computing WS 2017/2018

Lecture 6

Jürgen Fuhrmann

juergen.fuhrmann@wias-berlin.de

Recap: C++

Generic programming: templates

- Templates allow to write code where a data type is a parameter
- We want do be able to have vectors of any basic data type.
- We do not want to write new code for each type

```
template <typename T>
class vector
{
  private:
      T *data=nullptr;
      int size=0;
  public:
      int get_size( return size);
      T & operator[](int i) { return data[i]; };
      vector( int new_size) { data = new T[new_size];
                              size = new_size;};
      ~vector() { delete [] data;};
};
...
{
  vector<double> v(5);
  vector<int> iv(3);
}
```

C++ template libray

- The standard template library (STL) became part of the C++11 standard
- Whenever you can, use the classes available from there
- For one-dimensional data, std::vector is appropriate
- For two-dimensional data, things become more complicated
  - There is no reasonable matrix class
    - std::vector< std::vector> is possible but has to allocate each matrix row and is inefficient
  - it is hard to create a std::vector from already existing data

Vector operations

- So far we are able to perform vector operations by explicitely writing loops over the length of the vector
- Generally, C++ allows to *overload* operators like +,-,*,/,= etc. allowing to use vector expressions (like in matlab, python/numpy, Julia)

```
   inline const vector
   operator+( const vector& a, const vector& b )
   {
     vector tmp( a.size() );
     for( std::size_t i=0; i<a.size(); ++i )
     tmp[i] = a[i] + b[i];
     return tmp;
   }
   ...

 vector a,b,c;
 c=a+b;
```

- *But* this involves the creation of a temporary object for each operation in an expression
- Temporary object creation is prohibitively expensive for large objects

Expression templates I

C++ technique which allows to implement expressions of vectors while avoiding introduction and copies of temporary objects.

- Expression class definition:

```
template< typename A, typename B >
class Sum {
  public:
  Sum(const A& a, const B& b): a_( a ), b_( b ){} // Constructor from two vectors
  std::size_t size() const { return a_.size(); }  // Delegate size() to argument
  double operator[]( std::size_t i ) const        // Access operator
  { return a_[i] + b_[i]; }
  private:
  const A& a_;    // Reference to the left-hand side operand
  const B& b_;    // Reference to the right-hand side operand
};
```

- Overloaded + operator:

```
template< typename A, typename B >
const Sum<A,B> operator+( const A& a, const B& b )
{
  return Sum<A,B>( a, b ); // Return instance of Sum<A,B>
}
```

- a,b can be vectors, other expressions or any object which implements `size()` and `[]`.

Expression templates II

- Method in vector class to copy vector data from expression:

```
class vector
{
  public:
  ...
  template< typename A >
  vector& operator=( const A& expr )
  {
    for( std::size_t i=0; i<expr.size(); i++ )
        v_[i] = expr[i];
    return *this; // Return reference to target vector
  }
  ...
};
```

- Usage:

```
vector a,b,c;
c=a+b;
```

- After template instantiation and inlining, the compiler will generate code without temporary vector objects:

```
for( std::size_t i=0; i<a.size(); i++ )
   c[i] = a[i] + b[i];
```

- Large potential for optimization for more complex expressions

Vector classes for linear algebra

- Expression templates and overloading of component access allow to implement classes for linear algebra which are almost as easy to use as in matlab or python/numpy
- These techniques are used by libraries like
  - Eigen `http://eigen.tuxfamily.org`
  - Armadillo `http://arma.sourceforge.net/`
  - Blaze `https://bitbucket.org/blaze-lib/blaze/overview`
- Regrettably, none of this is standardized in C++ ...
- During the course, we will use our own, small and therefore hopefully easy to understand library named numcxx

numcxx

numcxx is a small C++ library developed for and during this course which implements the concepts introduced

- Shared smart pointers vs. references
- 1D/2D Array class
- Matrix class with LAPACK interface
- Expression templates
- Interface to triangulations
- Sparse matrices + UMFPACK interface
- Iterative solvers
- Python interface

numcxx classes

- `TArray1<T>`: templated 1D array class
  `DArray1`: 1D double array class
- `TArray2<T>`: templated 2D array class
  `DArray2`: 2D double array class
- `TMatrix<T>`: templated dense matrix class
  `DMatrix`: double dense matrix class
- `TSolverLapackLU<T>`: LU factorization based on LAPACK
  `DSolverLapackLU`: Specialization for double
- `TSparseMatrix<T>`: Sparse matrix class
  `DSparseMatrix`: Specialization for double
- `TSolverUMFPACK<T>`: Sparse LU factuorization based on UMFPACK
  `DSolverUMFPACK`: Specialization for double

Obtaining and compiling the examples

- Copy files, creating subdirectory part2
  - the . denotes the current directory

```
$ ls /net/wir/numxx/examples/10-numcxx-basicx/*.cxx
$ cp -r /net/wir/examples/10-numcxx-basicx/numcxx-expressions.cxx .
```

- Compile sources (for each of the .cxx files) (integrates with codeblocks)

```
$ numcxx-build -o example numcxx-expressions.cxx
$ ./example
```

C++ code using vectors, C-Style, with data on stack

File /net/wir/numcxx/examples/00-cxx-basics/01-c-style-stack.cxx

```
#include <cstdio>
void initialize(double *x, int n)
{
    for (int i=0;i<n;i++) x[i]= 1.0/(double)(1+n-i);
}
double sum_elements(double *x, int n)
{
    double sum=0;
    for (int i=0;i<n;i++) sum+=x[i];
    return sum;
}
int main()
{
    const int n=12345678;
    double x[n];
    initialize(x,n);
    double s=sum_elements(x,n);
    printf("sum=%e\n",s);
}
```

- Large arrays may not fit on stack
- C-Style arrays do not know their length

C++ code using vectors, C-Style, with data on heap

File /net/wir/numcxx/examples/00-cxx-basics/02-c-style-heap.cxx

```
#include <cstdio>
#include <cstdlib>
#include <new>
void initialize(double *x, int n)
{    for (int i=0;i<n;i++) x[i]= 1.0/(double)(1+n-i);
}
double sum_elements(double *x, int n)
{   double sum=0;
    for (int i=0;i<n;i++) sum+=x[i];
    return sum;
}
int main()
{   const int n=12345678;
    try  {  x=new double[n]; // allocate memory for vector on heap }
    catch (std::bad_alloc)  { printf("error allocating x\n");
                              exit(EXIT_FAILURE); }
    initialize(x,n);
    double s=sum_elements(x,n);
    printf("sum=%e\n",s);
    delete[] x;}
```

- Arrays passed in a similar way as in previous example
- Proper memory management is error prone

C++ code using vectors, (mostly) modern C++-style

File /net/wir/numcxx/examples/00-cxx-basics/03-cxx-style-ref.cxx

```
#include <cstdio>
#include <vector>
void initialize(std::vector<double>& x)
{ for (int i=0;i<x.size();i++) x[i]= 1.0/(double)(1+n-i);
}
double sum_elements(std::vector<double>& x)
{   double sum=0;
    for (int i=0;i<x.size();i++)sum+=x[i];
    return sum;}
int main()
{   const int n=12345678;
    std::vector<double> x(n); // Construct vector with n elements
                              // Object "lives" on stack, data on heap
    initialize(x);
    double s=sum_elements(x);
    printf("sum=%e\n",s);
    // Object destructor automatically called at end of lifetime
    // So data array is freed automatically
}
```

- Heap memory management controlled by object lifetime

C++ code using vectors, C++-style with smart pointers

File
/net/wir/numcxx/examples/00-cxx-basics/05-cxx-style-sharedptr.cxx

```
#include <cstdio>
#include <vector>
#include <memory>
void initialize(std::vector<double> &x)
{  for (int i=0;i<x.size();i++) x[i]= 1.0/(double)(1+n-i);}
double sum_elements(std::vector<double> & x)
{    double sum=0;
    for (int i=0;i<x.size();i++)sum+=x[i];
    return sum;
}
int main()
{   const int n=12345678;
    // call constructor and wrap pointer into smart pointer
    auto x=std::make_shared<std::vector<double>>(n);
    initialize(*x);
    double s=sum_elements(*x);
    printf("sum=%e\n",s);
    // smartpointer calls destructor if reference count reaches zero
}
```

- Heap memory management controlled by smart pointer lifetime
- If method or function does not store the object, pass by reference ⇒ API stays the same as for previous case.

CMake

What is behind `numcxx-build`?

- CMake - the current best way to build code
- Describe project in a file called `CMakeLists.txt`

```
cmake_minimum_required(VERSION 2.8.12)
PROJECT(example C CXX)
find_package(NUMCXX REQUIRED)
include_directories("${NUMCXX_INCLUDE_DIRS}")
link_libraries("${NUMCXX_LIBRARIES}")
add_executable(example example.cxx)
```

- Set up project (only once)

```
$ mkdir builddir
$ cd buildir
$ cmake ..
$ cd ..
```

- build code

```
$ cmake --build builddir
```

- run code

```
$ ./builddir/example
```

Numcxx with CodeBlocks

- CodeBlocks support has been added to numcxx-build:
  - `numcxx-build --codeblocks hello.cxx` creates a subdirectory `hello.codeblocks` which contains the codeblocks project file `hello.cbp`
  - Configure and then start codeblocks:

```
$ numcxx-build --codeblocks hello.cxx
$ codeblocks hello.codeblocks/hello.cbp
```

  - Or start codeblocks immediately after configuring

```
$ numcxx-build --codeblocks --execute hello.cxx
```

  - In Codeblocks, instead of "all" select target "hello" or "hello/fast", then Build & Run as usual.

Let's have some naming conventions

- lowercase letters: scalar values
  - `i,j,k,l,m,n` standalone or as prefixes: integers, indices
  - others: floating point
- uppercase letters: class objects/references

```
std::vector<double> X(n);
numcxx::DArray1<double> Y(n);
```

- pUpper_case_letters: smart pointers to objects

```
auto pX=std::make_shared<std::vector<double>>(n);
auto pY=numcxx::TArray1<double>::create(n);
auto pZ=numcxx::TArray1<double>::create({1,2,3,4});

// getting references from smart pointers
auto &X=*pX;
auto &Y=*pY;
auto &Z=*pZ;

auto W=std::make_shared<std::vector<double>>({1,2,3,4}); // doesn't work...
```

C++ code using numcxx with references

File /net/wir/examples/10-numcxx-basicx/numcxx-ref.cxx

```
#include <cstdio>
#include <numcxx/numcxx.hxx>
void initialize(numcxx::DArray1 &X)
{   const int n=X.size();
  for (int i=0;i<n;i++) X[i]= 1.0/(double)(1+n-i);
}
double sum_elements(numcxx::DArray1 & X)
{   double sum=0;
  for (int i=0;i<X.size();i++)sum+=X[i];
  return sum;
}
int main()
{   const int n=12345678;
  numcxx::TArray1<double> X(n);
  initialize(X);
  double s=sum_elements(X);
  printf("sum=%e\n",s);
}
```

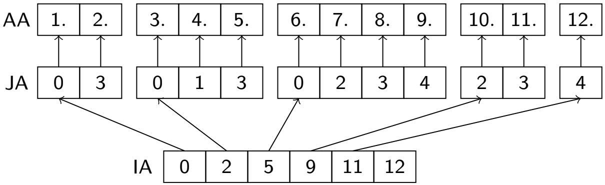
C++ code using numcxx with smart pointers

File /net/wir/examples/10-numcxx-basics/numcxx-sharedptr.cxx

```
#include <cstdio>
#include <memory>
#include <numcxx/numcxx.hxx>
void initialize(numcxx::DArray1 &X)
{   const int n=X.size();
  for (int i=0;i<n;i++) X[i]= 1.0/(double)(1+n-i);
}
double sum_elements(numcxx::DArray1 & X)
{   double sum=0;
  for (int i=0;i<X.size();i++)sum+=X[i];
  return sum;
}
int main()
{   const int n=12345678;
  // call constructor and wrap pointer into smart pointer
  auto pX=numcxx::TArray1<double>::create(n);
  initialize(*pX);
  double s=sum_elements(*pX);
  printf("sum=%e\n",s);
}
```

CRS format with zero array offsets . . .

$$A = \begin{pmatrix} 1. & 0. & 0. & 2. & 0. \\ 3. & 4. & 0. & 5. & 0. \\ 6. & 0. & 7. & 8. & 9. \\ 0. & 0. & 10. & 11. & 0. \\ 0. & 0. & 0. & 0. & 12. \end{pmatrix}$$



- some package APIs provide the possibility to specify array offset
- index shift is not very expensive compared to the rest of the work

numcxx Sparse matrix class

`numcxx::TSparseMatrix<T>`

- Class characterized by IA/JA/AA arrays
- How to create these arrays ?
- Common way (e.g. Eigen) : from a list triples i, j, a_{ij}. In practice, this can be expensive because in FEM assembly (especially in 3D) we will have many triplets with the same i, j but different a_{ij}
- Remedy:
  - Internally create and update an intermediate data structure which maintains information on already available entries
  - Hide this behind the facade $A(i, j) = x$
  - In order to switch between intermediate and final internal data structure, we need a `flush` method which triggers the rebuild.