Scientific Computing WS 2017/2018

Lecture 4

Jürgen Fuhrmann

juergen.fuhrmann@wias-berlin.de

Recap from last time

Classes and members

- Classes are data types which collect different kinds of data, and methods to work on them.

```
class class_name
{
  private:
    private_member1;
    private_member2;
    ...
  public:
    public_member1;
    public_member2;
    ...
};
```

- If not declared otherwise, all members are private
- `struct` is the same as `class` but by default all members are public
- Accessing members of a class object:

```
    class_name x;
    x.public_member1=...
```

- Accessing members of a pointer to class object:

```
    class_name *x;
    (*x).public_member1=...
    x->public_member1=...
```

Example class

- Define a class `vector` which holds data and length information and thus is more comfortable than plain arrays

```
class vector
{
  private:
      double *data;
  public:
       int size;
      double get_value( int i) {return data[i];};
      void set_value( int i, double value); {data[i]=value;};
};

...

{
  vector v;
  v.data=new double(5);  // would work if data would be public
  v.size=5;
  v.set_value(3,5);

  b=v.get_value(3); // now, b=5
  v.size=6;  // size changed, but not the length of the data array...
             // and who is responsible for delete[] at the end of scope ?
}
```

Constructors and Destructors

```
class vector
{ private:
      double *data=nullptr;
      int size=0;
  public:
      int get_size(){ return size;};
      double get_value( int i ) { return data[i]; };
      void set_value( int i, double value ) { data[i]=value; };
      Vector( int new_size ) { data = new double[new_size];
                              size=new_size; };
      ~Vector() { delete [] data; };
};
...
{ vector v(5);
  for (int i=0;i<5;i++) v.set_value(i,0.0);
  v.set_value(3,5);
  b=v.get_value(3); // now, b=5
  v.size=6;  // Size is now private and can not be set;
  vector w(5);
  for (int i=0;i<5;i++) w.set_value(i,v.get_value(i));
  // Destructors automatically called at end of scope.
}
```

- Constructors are declared as `classname(...)`
- Destructors are declared as `~classname()`

Interlude: References

- C style access to objects is direct or via pointers
- C++ adds another option - references
  - References essentially are alias names for already existing variables
  - Must always be initialized
  - Can be used in function parameters and in return values
  - No pointer arithmetics with them
- Declaration of reference

```
double a=10.0;
double &b=a;

b=15; // a=15 now as well
```

- Reference as function parameter: no copying of data!

```
void do_multiplication(double x, double y, double &result)
{
    result=x*y;
}
...
double x=5,y=9;
double result=0;
do_multiplication(x,y,result) // result now contains 45
```

Vector class again

- We can define () and [] operators!

```
class vector
{
  private:
      double *data=nullptr;
      int size=0;
  public:
      int get_size( return size);
      double & operator()(int i) { return data[i]; };
      double & operator[](int i) { return data[i]; };
      vector( int new_size) { data = new double[new_size];
                              size=new_size;}
      ~vector() { delete [] data;}
};
...
{
  vector v(5);
  for (int i=0;i<5;i++) v[i]=0.0;
  v[3]=5;
  b=v[3]; // now, b=5
  vector w(5);
  for (int i=0;i<5;i++) w(i)=v(i);
}
```

Matrix class

- We can define `(i,j)` but not `[i,j]`

```
class matrix
{ private:
      double *data=nullptr;
      int size=0;  int nrows=0;
      int ncols=0;
  public:
      int get_nrows( return nrows);
      int get_ncols( return ncols);
      double & operator()(int i,int j) { return data[i*nrow+j]);
      matrix( int new_rows,new_cols)
      { nrows=new_rows; ncols=new_cols;
        size=nrows*ncols;
        data = new double[size];
      }
      ~matrix() { delete [] data;}
};
...
{
  matrix m(3,3);
  for (int i=0;i<3;i++)
    for (int j=0;j<3;j++)
        m(i,j)=0.0;
}
```

Inheritance

- Classes in C++ can be extended, creating new classes which retain characteristics of the base class.
- The *derived class* inherits the members of the *base class*, on top of which it can add its own members.

```
class vector2d
{  private:
     double *data;
     int nrow, ncol;
     int size;
   public:
     double & operator(int i, int j);
     vector2d(int nrow, ncol);
     ~vector2d();
}
class matrix: public vector2d
{ public:
     apply(const vector1d & u, vector1d &v);
     solve(vector1d &u, const vector1d &rhs);
}
```

- All operations which can be performed with instances of `vector2d` can be performed with instances of `matrix` as well
- In addition, `matrix` has methods for linear system solution and matrix-vector multiplication

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);
}
```

Smart pointers

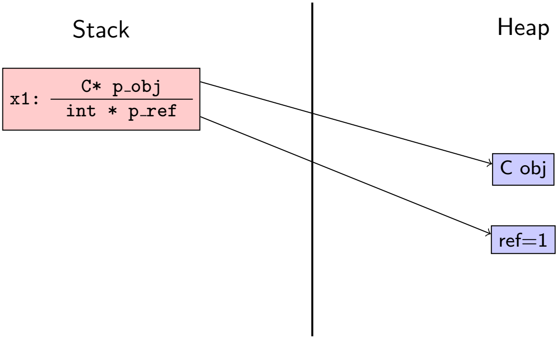
... with a little help from Timo Streckenbach from WIAS who introduced smart pointers into our simulation code.

- Automatic book-keeping of pointers to objects in memory.
- Instead of the meory addres of an object aka. pointer, a structure is passed around *by value* which holds the memory address and a pointer to a *reference count* object.
- It delegates the member access operator `->` and the address resolution operator `*` to the pointer it contains.
- Each assignment of a smart pointer increases this reference count.
- Each destructor invocation from a copy of the smart pointer structure decreases the reference count.
- If the reference count reaches zero, the memory is freed.
- `std::shared_ptr` is part of the C++11 standard

Smart pointer schematic

(this is one possibe way to implement it)

```
class C;
```

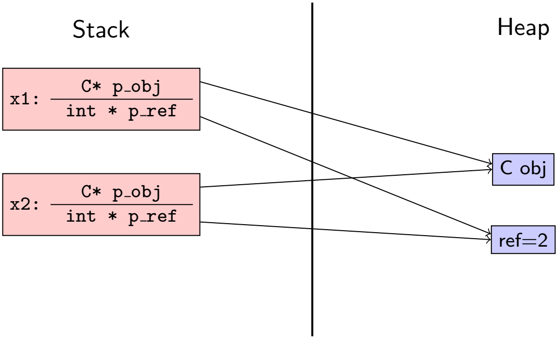



```
std::shared_ptr<C> x1= std::make_shared<C>();
```

Smart pointer schematic

(this is one possibe way to implement it)

```
class C;
```

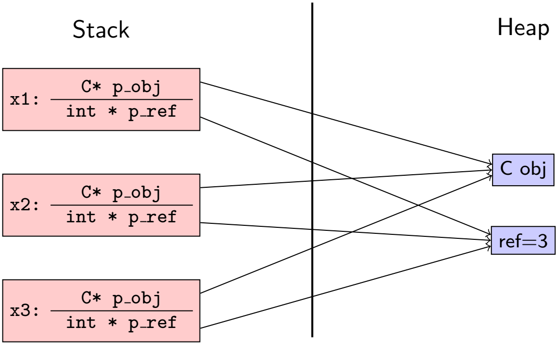



```
std::shared_ptr<C> x1= std::make_shared<C>();
std::shared_ptr<C> x2= x1;
```

Smart pointer schematic

(this is one possibe way to implement it)

```
class C;
```




```
std::shared_ptr<C> x1= std::make_shared<C>();
std::shared_ptr<C> x2= x1;
std::shared_ptr<C> x3= x1;
```

Smart pointers vs. *-pointers

- When writing code using smart pointers, write

```
#include <memory>
class R;
std::shared_ptr<R> ReturnObjectOfClassR(void);
void PassObjectOfClassR(std::shared_ptr<R> pR);
...
{  auto pR=std::make_shared<R>();
   PassObjectOfClassR(pR)
   // Smart pointer object is deleted at end of scope and frees memory
}
```

instead of

```
class R;
R* ReturnObjectOfClassR(void);
void PassObjectOfClassR(R* o);
...
{  R* pR=new R;
   PassObjectOfClassR(pR);
   delete pR; // never forget this here!!!
}
```

C/C++: Code examples

How to obtain/compile examples ?

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

File /net/wir/examples/part1/01-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/examples/part1/02-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/examples/part1/03-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/examples/part1/05-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.

Working with source code

- Copy the code:

```
$ cp /net/wir/examples/part1/example.cxx .
```

- Editing:

```
$ gedit  example.cxx
```

- Compiling: (-o gives the name of the output file)

```
$ g++ -std=c++11  example.cxx -o  example
```

- Running: (./ means file from current directory)

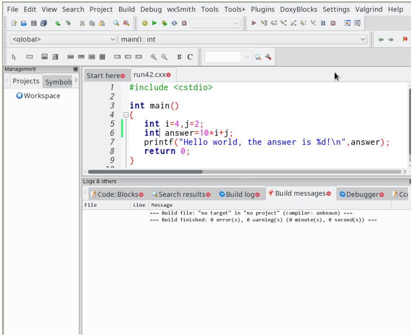
```
$ ./example
```

Alternative:Code::Blocks IDE

- `http://www.codeblocks.org/`
- Open example

```
$ codeblocks example.cxx
```

- Compile + run example: `Build/"Build and Run"` or `F9`
- Switching on C++11 standard: tick `Settings/Compiler/"Have g++ follow the C++11..."`



C/C++: Expression templates

Expression templates

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

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

Code with temporary objects

- Generally, C++ allows to *overload* operators like +,-,*,/,= etc.
- *But* this involves the creation of a temporary object for each operation in an expression

```
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;
}
```

- Temporary object creation is prohibitively expensive for large objects

Code with expression templates I

(K. Iglberger, “Expression templates revisited”)

Expression template:

```
template< typename A, typename B >
class Sum {
public:
    Sum( const A& a, const B& b ) : a_( a ), b_( b )    {}
    std::size_t size() const { return a_.size(); }
    double operator[]( std::size_t i ) const
    { 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 );
}
```

Code with expression templates II

Method 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;
   }
// ...
};
```

After template instantiation, the compiler will use

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

Vector classes for linear algebra

- Expression templates + overloading of component access allow to implement classes for linear algebra which are almost as easy to use as in matlab or python
- 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 rather easy to understand library named numcxx

C++ topics not covered so far

- To be covered later
  - Threads/parallelism
  - Graphics (via library)
- To be covered on occurence (possibly)
  - Character strings
  - Overloading
  - Functor classes, lambdas
  - malloc/free/realloc (C-style memory management)
  - cmath library
  - Interfacing C/Fortran
- To be omitted (probably)
  - optional arguments, variable parameter lists
  - Exceptions
  - Move semantics
  - GUI libraries
  - Interfacing Python/numpy

~

Recap from numerical analysis

Floating point representation

- Scientific notation of floating point numbers: e.g. $x = 6.022 \cdot 10^{23}$
- Representation formula:

$$x = \pm \sum_{i=0}^{\infty} d_i \beta^{-i} \beta^e$$

  - $\beta \in \mathbb{N}, \beta \geq 2$: base
  - $d_i \in \mathbb{N}, 0 \leq d_i \leq \beta$: mantissa digits
  - $e \in \mathbb{Z}$: exponent
- Representation on computer:

$$x = \pm \sum_{i=0}^{t-1} d_i \beta^{-i} \beta^e$$

  - $\beta = 2$
  - t: mantissa length, e.g. $t = 53$ for IEEE double
  - $L \leq e \leq U$, e.g. $-1022 \leq e \leq 1023$ (10 bits) for IEEE double
  - $d_0 \neq 0 \Rightarrow$ normalized numbers, unique representation

Floating point limits

- symmetry wrt. 0 because of sign bit
- smallest positive normalized number: $d_0 = 1, d_i = 0, i = 1 \dots t-1$
  $x_{min} = \beta^L$
- smallest positive denormalized number:
  $d_i = 0, i = 0 \dots t-2, d_{t-1} = 1$
  $x_{min} = \beta^{1-t}\beta^L$
- largest positive normalized number: $d_i = \beta - 1, 0 \dots t-1$
  $x_{max} = \beta(1-\beta^{1-t})\beta^U$

Machine precision

- Exact value x
- Approximation $\tilde{x}$
- Then: $|\frac{\tilde{x}-x}{x}| < \epsilon$ is the best accuracy estimate we can get, where
  - $\epsilon = \beta^{1-t}$ (truncation)
  - $\epsilon = \frac{1}{2}\beta^{1-t}$ (rounding)
- Also: ϵ is the smallest representable number such that $1 + \epsilon > 1$.
- Relative errors show up in partiular when
  - subtracting two close numbers
  - adding smaller numbers to larger ones

Matrix + Vector norms

- Vector norms: let $x = (x_i) \in \mathbb{R}^n$
  - $||x||_1 = \sum_i =^n |x_i|$: sum norm, l_1-norm
  - $||x||_2 = \sqrt{\sum_{i=1}^n x_i^2}$: Euclidean norm, l_2-norm
  - $||x||_\infty = \max_{i=1\dots n} |x_i|$: maximum norm, l_∞-norm
- Matrix $A = (a_{ij}) \in \mathbb{R}^n \times \mathbb{R}^n$
  - Representation of linear operator $\mathcal{A} : \mathbb{R}^n \to \mathbb{R}^n$ defined by $\mathcal{A} : x \mapsto y = Ax$ with

$$y_i = \sum_{j=1}^n a_{ij} x_j$$

  - Induced matrix norm:

$$\begin{aligned} ||A||_\nu &= \max_{x \in \mathbb{R}^n, x \neq 0} \frac{||Ax||_\nu}{||x||_\nu} \\ &= \max_{x \in \mathbb{R}^n, ||x||_\nu = 1} \frac{||Ax||_\nu}{||x||_\nu} \end{aligned}$$

Matrix norms

- $||A||_1 = \max_{j=1 \dots n} \sum_{i=1}^{n} |a_{ij}|$ maximum of column sums
- $||A||_\infty = \max_{i=1 \dots n} \sum_{j=1}^{n} |a_{ij}|$ maximum of row sums
- $||A||_2 = \sqrt{\lambda_{max}}$ with λ_{max}: largest eigenvalue of $A^T A$.

Matrix condition number and error propagation

Problem: solve $Ax = b$, where b is inexact.

$$A(x + \Delta x) = b + \Delta b.$$

Since $Ax = b$, we get $A\Delta x = \Delta b$. From this,

$$\left\{ \begin{array}{ll} \Delta x & = A^{-1}\Delta b \\ Ax & = b \end{array} \right\} \Rightarrow \left\{ \begin{array}{rl} ||A|| \cdot ||x|| & \geq ||b|| \\ ||\Delta x|| & \leq ||A^{-1}|| \cdot ||\Delta b|| \end{array} \right.$$

$$\Rightarrow \frac{||\Delta x||}{||x||} \leq \kappa(A) \frac{||\Delta b||}{||b||}$$

where $\kappa(A) = ||A|| \cdot ||A^{-1}||$ is the *condition number* of A.

Approaches to linear system solution

Solve $Ax = b$

Direct methods:

- Deterministic
- Exact up to machine precision
- Expensive (in time and space)

Iterative methods:

- Only approximate
- Cheaper in space and (possibly) time
- Convergence not guaranteed

Homework assignment

- Please find the first homework assignmet on the course homepage https://www.wias-berlin.de/people/fuhrmann/teach.html.
- Due Nov. 8.
- Can be finshed in Unix Pool or on your on computer

500 years ago

- Martin Luther
- 95 theses "Disputation on the Power of Indulgences"
- Wittenberg Church portal

500 years ago

- Martin Luther
- 95 theses "Disputation on the Power of Indulgences"
- Wittenberg Church portal

No lecture on Tue Oct 31 "Reformation Day"