∼

C++ roundup + NUMA recap

Scientific Computing Winter 2016/2017

Lecture 4

Jürgen Fuhrmann

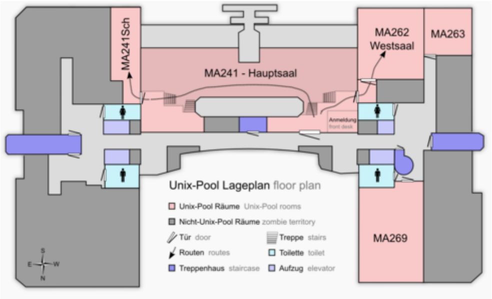
juergen.fuhrmann@wias-berlin.de

With material from from `http://www.cplusplus.com/` and from “Introduction to High-Performance Scientific Computing” by Victor Eijkhout (`http://pages.tacc.utexas.edu/~eijkhout/istc/istc.html`)

made wit pandoc

Admin

- Starting Oct. 31, on Mondays we will try out UNIX pool room **MA 269**
- Computer work in groups of two. Need list of names
- Homework will be this afternoon on homepage
  `http://www.wias-berlin.de/people/fuhrmann/teach.html`
- Consulting for first steps on UNIX on Monday



$\sim$

Recap from last time

The Preprocessor

- Before being sent to the compiler, the source code is sent through the *preprocessor*
- It is a legacy from C which is slowly being squeezed out of C++
- Preprocessor commands start with `#`
- Include contents of file `file.h` found on a default search path known to the compiler:

```
#include <file.h>
```

- Include contents of file `file.h` found on user defined search path

```
#include "file.h"
```

- Define a piece of text (mostly used for constants in pre-C++ times), Avoid! Use `const` instead.

```
#define N 15
```

- Define preprocessor macro for inlining code.
  Avoid! Use inline functions instead

```
#define  MAX(x,y) (((x)>(y))?(x):(y))
```

Why macros are evil ?

(Argumentation from stackoverflow)

- You can not debug macros.
  - a debugger allows to execute the the program statement by statement in order to find errors. Within macros, this is not possible
- Macro expansion can lead to strange side effects.

```
#define  MAX(x,y) (((x)>(y))?(x):(y))
auto a=5, b=4;
auto c=MAX(++a,b);        // gives c=7
auto d=std::max(++a,b);   // gives d=6
```

- Macros have no “namespace”, so it is easy to “replace” functions without notification. If one uses a function, the compiler would issue a warning.
- Macros may affect things you don’t realize. The semantics of macros is completely arbitrary and not detectable by the compiler

Emulating modules

- Until now C++ has no well defined module system.
- A module system usually is emulated using the preprocessor and namespaces. Here we show the ideal way to do this
- File `mymodule.h` containing interface declaratiions

```
#ifndef MYMODULE_H
#define MYMODULE_H
namespace mymodule
{
     void my_function(int i, double x);
}
#endif
```

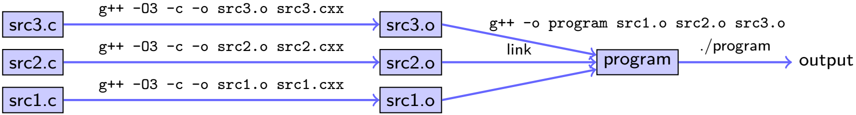
- File `mymodule.cpp` containing function definitions

```
#include "mymodule.h"
namespace mymodule
{
      void my_function(int i, double x)
      {
          ...body of function definition...
      }
}
#endif
```

- File using `mymodule`:

```
#include "mymodule.h"
...
mymodule::my_function(3,15.0);
```

Compiling...




```
$ g++ -O3 -c -o src3.o src3.cxx
$ g++ -O3 -c -o src2.o src2.cxx
$ g++ -O3 -c -o src1.o src1.cxx
$ g++ -o program src1.o src2.o src3.o
$ ./program
```

Shortcut: invoke compiler and linker at once

```
$ g++ -O3 -o program src1.cxx src2.cxx src3.cxx
$ ./program
```

Arrays

- Focusing on numerical methods for PDEs results in work with finite dimensional vectors which are represented as *arrays* - sequences of consecutively stored objects
- Stemming from C, in C++ array objects represent just the fixed amount of consecutive memory. No size info or whatsoever
- No bounds check
- First array index is always 0

```
double x[9]; // uninitialized array of 9 elements
double y[3]={1,2,3}; // initialized array of 3 elements
double z[]={1,2,3};  // Same
double z[]{1,2,3};   //Same
```

- Accessing arrays
  - [] is the array access operator in C++
  - Each element of an array has an index

```
double a=x[3];  // undefined value because x was not initialized
double b=y[12]; // undefined value because out of bounds
y[12]=19;       // may crash program ("segmentation fault"),
double c=z[0];  // Acces to first element in array, now c=1;
```

Arrays, pointers and pointer arithmetic

- Arrays are strongly linked to pointers
- Array object can be treated as pointer

```
double x[]={1,2,3,4};
double b=*x; // now x=1;
double *y=x+2; // y is a pointer to third value in arrax
double c=*y;   // now c=3
ptrdiff_t d=y-x; // We can also do differences between pointers
```

- Pointer arithmetic is valid only in memory regions belonging to the same array

Memory: stack and heap

- Stack: pre-allocated memory where `main()` and all functions called from there put their data.
  - All data declared in {} blocks are placed on the stack
  - Stack size is limited
  - Handling stack memory is cheap

```
{
  double a[10000];
  for (int i=0;i<10000;i++) a[i]=0.0;
  // stack memory implicitely freed  at end of block
}
```

- Heap: Additional memory available from system on request
  - Mix between array and pointer arithmetic allows to access stack and heap allocated arrays in the same way.
  - only the pointer is placed on the stack
  - `new/delete` are expensive operations

```
{
  double *a= new double[10000];
  for (int i=0;i<10000;i++) a[i]=0.0;
  delete[] a; // need to release memory explicitely
}
```

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` data types are defined in the same way as classes, 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=...
```

Templated vector class

- We want to 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;   // Plain C-style pointer to data array
      int _size=0;       // Private size information
  public:
      int size() {return _size;}  // Retrieval of size information
      T & operator[](int i) { return data[i]);  // Array access operator
      vector( int size): _size(size) { data = new T[size];} // Constructor
      ~vector() { delete [] data;} // Destructor
};
...
{
  vector<double> v(5);
  vector<int> iv(3);
}
```

- A vector class like this is available from the C++ standard template library

C++ standard template libray (STL)

- 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 not possible to create an std::vector from already existing data
- STL vector constructors are not able to use already allocated memory, as it becomes available e.g. from mesh generators like TetGen, or when interfacing numpy array from python
- The way forward in projects: existing C++ linear algebra libraries
  - Eigen
  - Armadillo
  - numcpp
- For teaching: develop own small library, explaining all internal mechanisms

∼

Inheritance and smart pointers

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;
    vector2d<int> shape;
    int size
public:
    double & operator(int i, int j);
    vector2d(int nrow, ncol);
    ~vector2d();template <t
}

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

Smart pointers

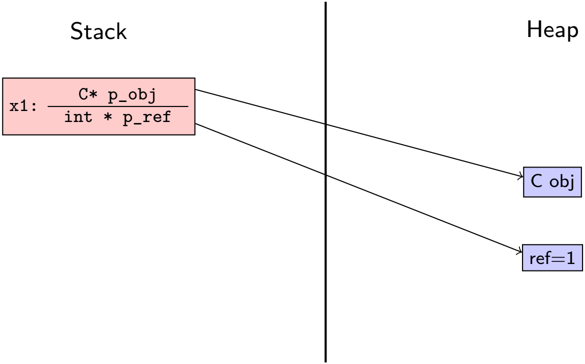
... with a little help from Timo Streckenbach from WIAS who introduced smart pointers into our simulation code.

- 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 decreses 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;
```

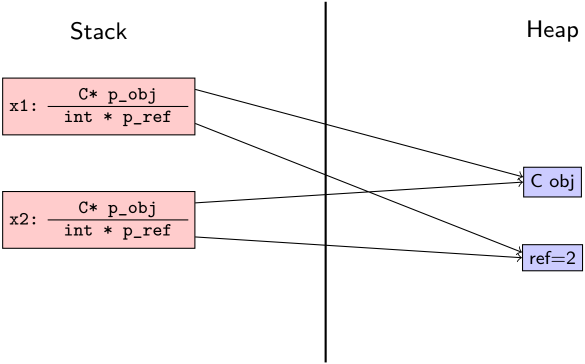


std::shared_ptr<C> x1= std::make_shared<C>();

Smart pointer schematic

(this is one possibe way to implement it)

```
class C;
```

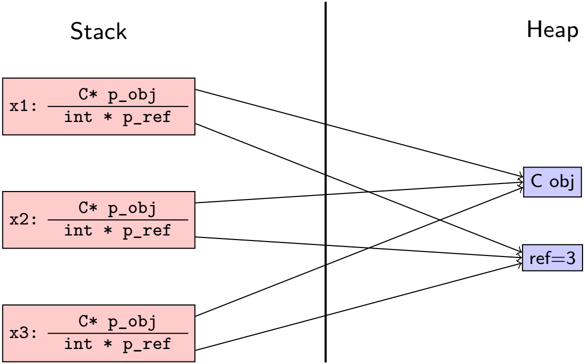



```
std::shared_ptr<C> x1= std::make_shared<C>();
std::shared_ptr<C> x2= x1;
```

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

```cpp
#include <memory>
class R;
std::shared_ptr<R> ReturnObjectOfClassR(void);
void PassObjectOfClassR(std::shared_ptr<R> o);
...
std::shared_ptr<R> o;
o->member=5;
...
{
   auto o=std::make_shared<R>();
   PassObjectOfClassR(o)
   // Smart pointer object is deleted at end of scope and frees memory
}
```

instead of

```cpp
class R;
R* ReturnObjectOfClassR(void);
void PassObjectOfClassR(R* o);
...
R*o;
o->member=5;
...
{
   R* o=new R;
   PassObjectOfClassR(o);
   delete o;
}
```

Smart pointer advantages vs. $*$-pointers

- "Forget" about memory deallocation
- Automatic book-keeping in situations when members of several different objects point to the same allocated memory
- Proper reference counting when working together with other libraries, e.g. numpy

C++ topics not covered so far

- To be covered on occurence
  - character strings
  - overloading
  - optional arguments, variable parameter lists
  - Functor classes, lambdas
  - threads
  - malloc/free/realloc (C-style memory management)
  - cmath library
  - Interfacing C/Fortran
  - Interfacing Python/numpy
- To be omitted (probably)
  - Exceptions
  - Move semantics
  - Expression templates
    - Expression templates allow to write code like `c=A*b` for a matrix `A` and vectors `b,c`.
    - Realised e.g. in Eigen, Armadillo
    - Too complicated for teaching (IMHO)
  - GUI libraries
  - Graphics (we aim at python here)

$\sim$

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