Scientific Computing WS 2017/2018

Lecture 2

Jürgen Fuhrmann

juergen.fuhrmann@wias-berlin.de

Recap from last time

Me

- Name: Dr. Jürgen Fuhrmann (no, not Prof.)
- Contact: **juergen.fuhrmann@wias-berlin.de**, `http://www.wias-berlin.de/people/fuhrmann/teach.html`
- Affiliation: Weierstrass Institute for Applied Analysis and Stochastics, Berlin (WIAS);
  Deputy Head, *Numerical Mathematics and Scientific Computing*
- Experience/Field of work:
  - Numerical solution of partial differential equations (PDEs)
  - Development, investigation, implementation of finite volume discretizations for nonlinear systems of PDEs
  - Ph.D. on multigrid methods
  - Applications: electrochemistry, semiconductor physics, groundwater...
  - Software development:
    - WIAS code pdelib (`http://pdelib.org`)
    - Languages: C, C++, Python, Lua, Fortran
    - Visualization: OpenGL, VTK

Slide 2

Admin stuff

- Lectures: **Tue 8-10 FH 311**, **Thu 10-12 MA269**
- Consultation: **Thu 12-13 MA269**, more at WIAS on appointment
- There will be coding assignments, mostly in C++
  - Unix pool
  - Linux, MacOSX+ Homebrew, Windows+Cygwin on your own PC/laptop
- Access to examination
  - Attend $\approx$ 80% of lectures
  - Return assignments ($\approx$ 4-5, but yet to be determined)
  - General activity during course
- Course material will be online: slides for sure, I intend to develop a script

UNIX Pool

https://www.math.tu-berlin.de/iuk/lehrrechnerbereich/v_menue/lehrrechnerbereich/

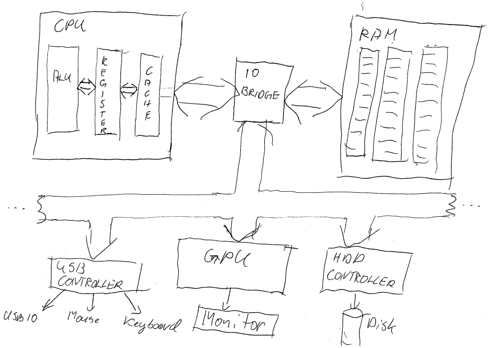
- Working groups of two students per account/computer
- All examples during this course will be available on UNIX pool systems
- All homework can be done on UNIX pool machines as well (Room MA241 outside of course hours)
- Please find yourself in groups of two and fill in the list of accounts for the unix pool.
- Once the administrators open the accounts, you will be able to log in and enter a new password

Recap from last time

Intended aims and topics of this course

- Indicate a reasonable path within this labyrinth
- Relevant topics from numerical analysis
- Introduction to C++ ($\approx$ 3 lectures) and Python (short, mostly for graphics purposes)
- Provide technical skills to understand a part of the inner workings of the relevant tools
- Focus on partial differential equation (PDE) solution
  - Finite elements
  - Finite volumes
  - Mesh generation
  - Nonlinear if time permits – so we can see some real action
  - Parallelization
  - A bit of visualization
- Tools/Languages
  - C++, Python
  - Parallelization: Focus on OpenMP, but glances on MPI, C++ threads
  - Visualization: Python, VTK

von Neumann Architecture



- Data and instructions from same memory
  - Instruction decode: determine operation and operands
  - Get operands from memory
  - Perform operation
  - Write results back
  - Continue with next instruction

Memory Hierachy

- Main memory access is slow compared to the processor
  - 100–1000 cycles latency before data arrive
  - Data stream maybe 1/4 floating point number/cycle;
  - processor wants 2 or 3
- Faster memory is expensive
- *Cache* is a small piece of fast memory for intermediate storage of data
- Operands are moved to CPU *registers* immediately before operation
- Memory hierarchy:

Registers in different cores
Fast on-CPU cache memory (L1, L2, L3)
Main memory

Machine code

- Detailed instructions for the actions of the CPU
- Not human readable
- Sample types of instructions:
  - Transfer data between memory location and register
  - Perform arithmetic/logic operations with data in register
  - Check if data in register fulfills some condition
  - Conditionally change the memory address from where instructions are fetched $\equiv$ "jump" to address
  - Save all register context and take instructions from different memory location until return $\equiv$ "call"
- Instructions are very hard to handle, although programming started this way...

```
534c 29e5 31db 48c1 fd03 4883 ec08 e85d
feff ff48 85ed 741e 0f1f 8400 0000 0000
4c89 ea4c 89f6 4489 ff41 ff14 dc48 83c3
0148 39eb 75ea 4883 c408 5b5d 415c 415d
415e 415f c390 662e 0f1f 8400 0000 0000
f3c3 0000 4883 ec08 4883 c408 c300 0000
0100 0200 4865 6c6c 6f20 776f 726c 6400
011b 033b 3400 0000 0500 0000 20fe ffff
8000 0000 60fe ffff 5000 0000 4dff ffff
```

Assembler code

- Human readable representation of CPU instructions
- Some write it by hand ...
  - Code close to abilities and structure of the machine
  - Handle constrained resources (embedded systems, early computers)
- Translated to machine code by a programm called *assembler*

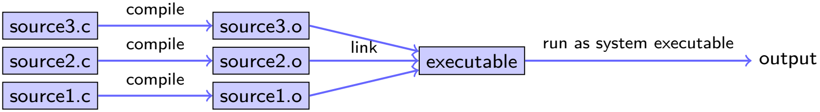
```
    .file   "code.c"
    .section    .rodata
.LC0:
    .string "Hello world"
    .text
    ...
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    subq    $16, %rsp
    movl    %edi, -4(%rbp)
    movq    %rsi, -16(%rbp)
    movl    $.LC0, %edi
    movl    $0, %eax
    call    printf
```

Compiled high level languages

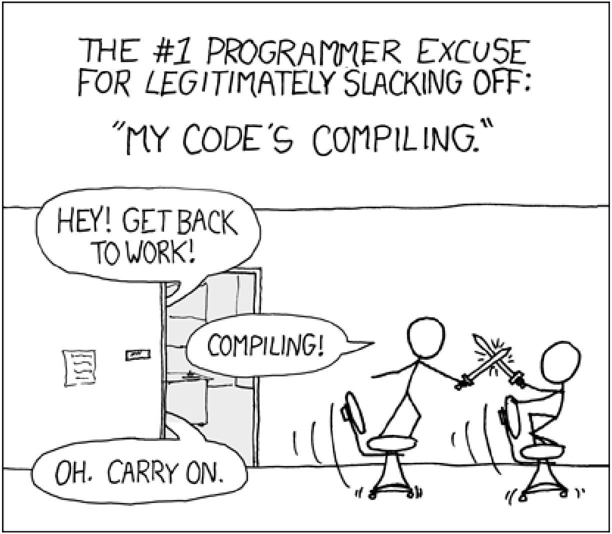
- Algorithm description using mix of mathematical formulas and statements inspired by human language
- Translated to machine code (resp. assembler) by *compiler*

```
#include <stdio.h>
int main (int argc, char *argv[])
{
  printf("Hello world");
}
```

- "Far away" from CPU ⇒ the compiler is responsible for creation of optimized machine code
- Fortran, COBOL, C, Pascal, Ada, Modula2, C++, Go, Rust, Swift
- Strongly typed
- Tedious workflow: compile - link - run



Compiling. . .



. . . from xkcd

Compiled languages in Scientific Computing

- Fortran: FORmula TRANslator (1957)
  - Fortran4: really dead
  - Fortran77: large number of legacy libs: BLAS, LAPACK, ARPACK ...
  - Fortran90, Fortran2003, Fortran 2008
    - Catch up with features of C/C++ (structures,allocation,classes,inheritance, C/C++ library calls)
    - Lost momentum among new programmers
    - Hard to integrate with C/C++
    - In many aspects very well adapted to numerical computing
    - Well designed multidimensional arrays
- C: General purpose language
  - K&R C (1978) weak type checking
  - ANSI C (1989) strong type checking
  - Had structures and allocation early on
  - Numerical methods support via libraries
  - Fortran library calls possible
- C++: *The* powerful object oriented language
  - Superset of C (in a first approximation)
  - Classes, inheritance, overloading, templates (generic programming)
  - C++11: Quantum leap: smart pointers, threads, lambdas, initializer lists in standard
  - With great power comes the possibility of great failure...

First steps with C++

Evolution

- Essentially, C++ started as "C with classes"
- Current standard is C++11, C++14 and C++17 are evolving.
- Almost all of the C language is part of C++
- C standard library is part of C++ standard librar
- As most computer languages, C++ has variables, flow control, functions etc. which will be discussed first

Printing stuff

Printing is not part of the language itself, but is performed via functions from libraries. As we need printing very early in the examples, we show how to do it.

- IOStream library
  - "Official" C++ output library
  - Type safe, easy to extend
  - Clumsy syntax for format control

```
#include <iostream>
...
std::cout << "Hello world" << std::endl;
```

- C Output library
  - Supported by C++-11 standard
  - No type safety, Hard to extend
  - Short, relatively easy syntax for format control
  - Same format specifications as in Python

```
#include <cstdio>
...
std::printf("Hello world\n");
```

C++ : scalar data types

- Store character, integer and floating point values of various sizes
- Type sizes are the “usual ones” on 64bit systems

name	printf	bytes	bits	Minimum value	Maximum value
char	%c (%d)	1	8	-128	127
unsigned char	%c (%d)	1	8	0	255
short int	%d	2	16	-32768	32767
unsigned short int	%u	2	16	0	65535
int	%d	4	32	-2147483648	2147483647
unsigned int	%u	4	32	0	4294967295
long int	%ld	8	64	-9223372036854775808	9223372036854775807
unsigned long int	%lu	8	64	0	18446744073709551615
float	%e	4	32	1.175494e-38	3.402823e38
double	%e	8	64	2.225074e-308	1.797693e308
long double	%Le	16	128	3.362103e-4932	1.189731e4932
bool	%d	1	8	0	1

- The standard only guarantees that

  `sizeof(short ...) <= sizeof(...) <=sizeof(long ...)`
- E.g. on embedded systems sizes may be different
- Declaration and output (example)

```
#include <cstdio>
...
int i=3;
double x=15.0;
std::printf("i=%d, x=%e\n",i,x);
```

Typed constant expressions

- C++ has the ability to declare variables as constants:

```
const int i=15;
i=i+1; // attempt to modify value of const object leads to
     // compiler error
```

Scopes, Declaration, Initialization

- **All variables are typed and must be declared**
  - Declared variables "live" in scopes defined by braces `{ }`
  - Good practice: initialize variables along with declaration
  - "auto" is a great innovation in C++11 which is useful with complicated types which arise in template programming
    - type of *lvalue* (left hand side value) is detected from type of *rvalue* (value at the right hand side)

```
{
   int i=3;
   double x=15.0;
   auto y=33.0;
}
```

Arithmetic operators

- Assignment operator

```
a=b;
c=(a=b);
```

- Arithmetic operators `+`, `-`, `*`, `/`, modulo (`%`)
- Beware of precedence which (mostly) is like in math!
- If in doubt, use brackets, or look it up!

- Compund assignment: `+=`, `-=`, `*=`, `/=`, `%=`

```
x=x+a;
x+=a; // equivalent to =x+a
```

- Increment and decrement:
  `++`,`--`

```
y=x+1;
y=x++; // equivalent to y=x; x=x+1;
y=++x; // equivalent to x=x+1; y=x;
```

Further operators

- Relational and comparison operators `==, !=, >, <, >=, <=`
- Logical operators `!, &&, ||`
  - short circuit evaluation:
    - if `a` in `a&&b` is false, the expression is false and `b` is never evaluated
    - if `a` in `a||b` is true, the expression is true and `b` is never evaluated
- Conditional ternary operator `?`

```
c=(a<b)?a:b; // equivalent to the following
if (a<b) c=a; else c=b;
```

- Comma operator `,`

```
c=(a,b); // evaluates to c=b
```

- Bitwise operators `&, |, ^, ~, <<, >>`
- `sizeof`: memory space (in bytes) used by the object resp. type

```
n=sizeof(char); //  evaluate
```

Functions

- Functions have to be *declared* and given names as other variables:

```
type name(type1 p1, type2 p2,...);
```

- (...) holds parameter list
  - each parameter has to be defined with its type
- type part of declaration describes type of return value
  - void for returning nothing

```
double multiply(double x, double y);
```

- Functions are *defined* by attaching a scope to the declaration
  - *Values of parameters are copied into the scope*

```
double multiply(double x, double y)
{
   return x*y;
}
```

- Functions are *called* by statements invoking the function with a particular set of parameters

```
{
    double s=3.0, t=9.0;
    double result=multiply(s,t);
    printf("s=%e, t=%e, s*t= %e\n",s,t,result); // s and t keep their values
}
```

Functions: inlining

- Function calls sometimes are expensive compared to the task performed by the function
  - Remember: save all register context and take instructions from different memory location until return, restore register context after return
  - The compiler may include the content of functions into the instruction stream instead of generating a call

```
inline double multiply(double x, double y)
{
   return x*y;
}
```

Flow control: Statements and conditional statements

- Statements are individual expressions like declarations or instructions or sequences of statements enclosed in curly braces:

```
    { statement1; statement2; statement3; }
```

- Conditional execution: `if`

```
  if (condition) statement;
  if (condition) statement; else statement;
```

```
if (x>15)
{
    printf("error");
}
else
{
   x++;
}
```

Equivalent but less safe:

```
if (x>15)
    printf("error");
else
   x++;
```

Flow control: Simple loops

- While loop:
  `while (condition) statement;`

```
i=0;
while (i<9)
{
  printf("i=%d\n",i);
  i++;
}
```

- Do-While loop: `do statement while (condition);`

Flow control: for loops

- This is the most important kind of loops for numerical methods.
  `for (initialization; condition; increase) statement;`

  1. initialization is executed. Generally, here, one declares a counter variable and sets it to some initial value. This is executed a single time, at the beginning of the loop.
  2. condition is checked. If it is true, the loop continues; otherwise, the loop ends, and statement is skipped, going directly to step 5.
  3. statement is executed. As usual, it can be either a single statement or a block enclosed in curly braces { }
  4. increase is executed, and the loop gets back to step 2.
  5. The loop ends: execution continues at the next statement after it.

- All elements (initialization, condition, increase, statement) can be empty

```
for (int i=0;i<9;i++)   printf("i=%d\n",i); // same  as on previous slide
for(;;);  // completely valid, runs forever
```

Flow control: break, continue

- break statement: "premature" end of loop

```
for (int i=1;i<10;i++)
{
   if (i*i>15) break;
}
```

- continue statement: jump to end of loop body

```
for (int i=1;i<10;i++)
{
   if (i==5) continue;
   else do_someting_with_i;
}
```

Flow control: switch

```
switch (expression)
{
  case constant1:
     group-of-statements-1;
     break;
  case constant2:
     group-of-statements-2;
     break;
  ...
  default:
     default-group-of-statements
}
```

equivalent to

```
if      (expression==constant1) {group-of-statements-1;}
else if (expression==constant2) {group-of-statements-2;}
...
else                            {default-group-of-statements;}
```

Execution of switch statement can be faster than the hierarchy of if-then-else statement

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))
```

Conditional compilation and pragmas

- Conditional compilation of pieces of source code, mostly used to dispatch between system dependent variant of code. Rarely necessary nowadays…

```
#ifdef MACOSX
statements to be compiled only for MACOSX
#else
statements for all other systems
#endif
```

- There can be more complex logic involving constant expressions
- A pragma gives directions to the compiler concerning code generation:

```
#pragma omp parallel
```

Headers

- If we want to use functions from the standard library we need to include a *header file* which contains their declarations
  - The `#include` statement invokes the C-Preprocessor and leads to the inclusion of the file referenced therein into the actual source
  - Include files with names in < > brackets are searched for in system dependent directories known to the compiler

```
#include <iostream>
```

Namespaces

- Namespaces allow to prevent clashes between names of functions from different projects
  - All functions from the standard library belong to the namespace `std`

```
namespace foo
{
  void cool_function(void);
}

namespace bar
{
  void cool_function(void);
}

...

{
  using namespace bar;
  foo::cool function()
  cool_function() // equivalent to bar::cool_function()
}
```

Modules ?

- Currently, C++ has no well defined module system.
- A module system usually is emulated using the preprocessor and namespaces.

Emulating modules

- File `mymodule.h` containing interface declarations

```
#ifndef MYMODULE_H // Handle multiple #include statements
#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);
```

main

Now we are able to write a complete program in C++

- `main()`
  is the function called by the system when running the program.
  Everything else needs to be called from there.

Assume the follwing content of the file `run42.cxx`:

```
#include <cstdio>

int main()
{
   int i=4,j=2;
   int answer=10*i+j;
   printf("Hello world, the answer is %d!\n",answer);
   return 0;
}
```

Then the sequence of command line commands

```
$ g++ -o run42 run42.cxx
$ ./run42
```

gives the right answer to (almost) anything.

Command line instructions to control compiler

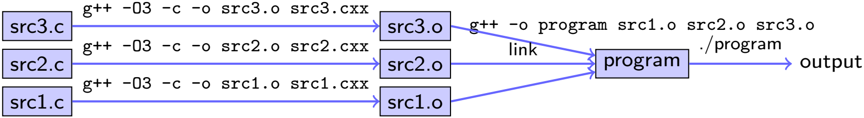
- By default, the compiler command performs the linking process as well
- Compiler command (Linux)

```
g++       GNU C++ compiler
g++-5     GNU C++ 5.x
clang++   CLANG compiler from LLVM project
icpc      Intel compiler
```

- Options (common to all of those named above, but not standardized)

```
-o name             Name of output file
-g                  Generate debugging instructions
-O0, -O1, -O2, -O3  Optimization levels
-c                  Avoid linking
-I<path>            Add <path> to include search path
-D<symbol>          Define preprocessor symbol
-std=c++11          Use C++11 standard
```

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
```

Some shell commands in the terminal window

```
ls -l             list files in directory
                  subdirectories are marked with 'd'
                  in the first column of permission list
cd  dir           change directory to dir
cd  ..            change directory one level up in directory hierachy
cp  file1 file2   copy file1 to file2
cp  file1 dir     copy file1 to directory
mv  file1 file2   rename file1 to file2
mv  file1 dir     move file1 to directory
rm  file          delete file
[cmd] *.o         perform command on all files with name ending with .o
```

Editors & IDEs

- Source code is written with text editors
  (as compared to word processors like `MS Word` or `libreoffice`)
- Editors installed are
  - `gedit` - text editor of gnome desktop (recommended)
  - `emacs` - comprehensive, powerful, a bit unusual GUI (my preferred choice)
  - `nedit` - quick and simple
  - `vi`, `vim` - the UNIX purist's crowbar
    (which I avoid as much as possible)
- Integrated development environments (IDE)
  - Integrated editor/debugger/compiler
  - eclipse (need to get myself used to it before teaching)