# Introduction

Scientific Computing WS 2017/2018

Lecture 1

Jürgen Fuhrmann

juergen.fuhrmann@wias-berlin.de

## 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

## 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 ≈ 80% of lectures
  - Return assignments (≈ 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

## Literature

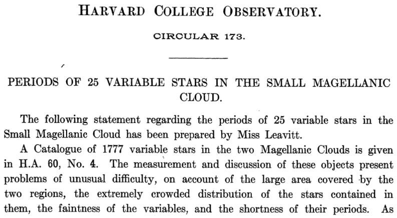
- Numerical methods
  - Y. Saad: Iterative methods for sparse linear systems
    `http://www-users.cs.umn.edu/~saad/IterMethBook_2ndEd.pdf`
  - V. Eijkhout: Introduction to High-Performance Scientific Computing
    `https://bitbucket.org/VictorEijkhout/hpc-book-and-course/`
- C/C++: look for resources on the new standard C++11
  - B. Stroustrup: The C++ Programming Language, **4th Edition**
  - P. Gottschling: Discovering Modern C++: An Intensive Course for Scientists, Engineers, and Programmers (C++ In-Depth)
  - `http://www.cplusplus.com/`
  - `https://isocpp.org/get-started`
  - `http://en.cppreference.com/w/`
- Python: look for resources on Python3
  - `https://www.python.org/`
  - `https://docs.python.org/3/tutorial/`
  - H.P. Langtangen ( † 2016): A Primer on Scientific Programming with Python
    `https://hplgit.github.io/primer.html/doc/pub/half/book.pdf`

# There was a time when “computers” were humans

Harvard Computers, circa 1890

By Harvard College Observatory - Public Domain

https://commons.wikimedia.org/w/index.php?curid=10392913



HARVARD COLLEGE OBSERVATORY.

CIRCULAR 173.

PERIODS OF 25 VARIABLE STARS IN THE SMALL MAGELLANIC CLOUD.

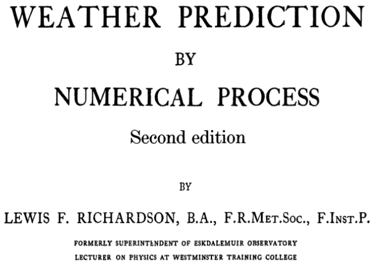
The following statement regarding the periods of 25 variable stars in the Small Magellanic Cloud has been prepared by Miss Leavitt.

A Catalogue of 1777 variable stars in the two Magellanic Clouds is given in H.A. 60, No. 4. The measurement and discussion of these objects present problems of unusual difficulty, on account of the large area covered by the two regions, the extremely crowded distribution of the stars contained in them, the faintness of the variables, and the shortness of their periods. As

It was about science – astronomy

## Does this scale ?

64000 computers predicting weather (1986 Illustration of L.F. Richardson's vision by S. Conlin)



WEATHER PREDICTION

BY

NUMERICAL PROCESS

Second edition

BY

LEWIS F. RICHARDSON, B.A., F.R.MET.SOC., F.INST.P.

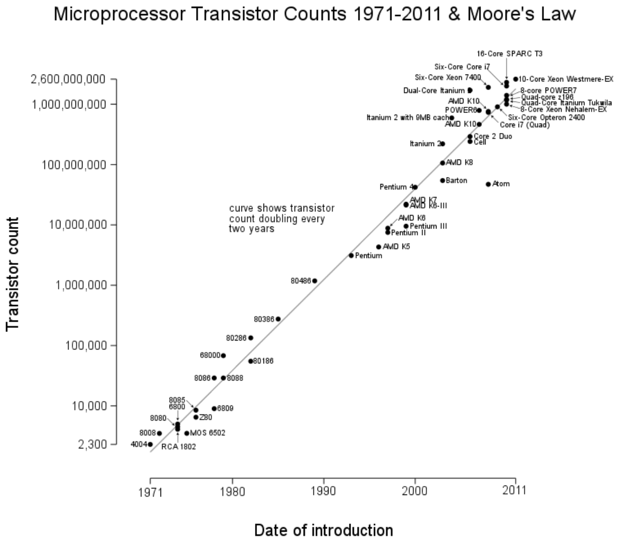
FORMERLY SUPERINTENDENT OF ESKDALEMUIR OBSERVATORY

LECTURER ON PHYSICS AT WESTMINSTER TRAINING COLLEGE

L.F.Richardson 1922

- This was about weather, not science in the first place
- Science *and* Engineering need computing

# Computing was taken over by machines



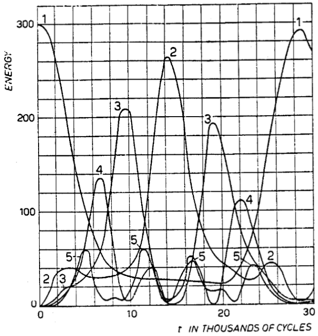
By Wgsimon - Own work, CC BY-SA 3.0 - Public Domain `https://commons.wikimedia.org/w/index.php?curid=15193542`

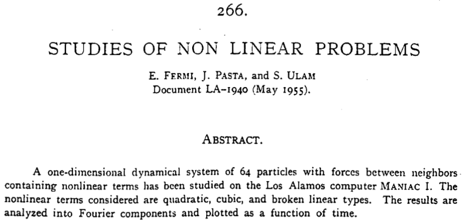
## Computational engineering

- Starting points: Nuclear weapons + rocket design, ballistic trajectories, weather . . .
- Now ubiquitous:
  - Structural engineering
  - Car industry
  - Oil recovery
  - . . .
- Use of well established, verfied, well supported commercial codes
  - Comsol
  - ANSYS
  - Eclipse
  - . . .

# As soon as computing machines became available . . .

. . . Scientists “misused” them to satisfy their curiosity




266.

STUDIES OF NON LINEAR PROBLEMS

E. FERMI, J. PASTA, and S. ULAM
Document LA-1940 (May 1955).

ABSTRACT.

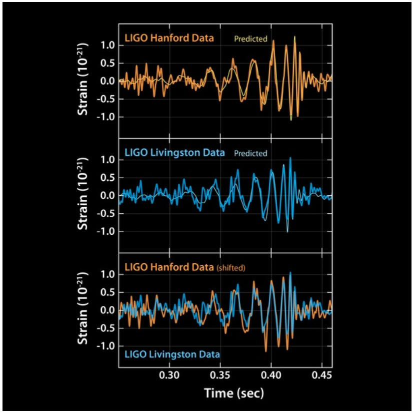
A one-dimensional dynamical system of 64 particles with forces between neighbors containing nonlinear terms has been studied on the Los Alamos computer MANIAC I. The nonlinear terms considered are quadratic, cubic, and broken linear types. The results are analyzed into Fourier components and plotted as a function of time.

“. . . Fermi became interested in the development and potentialities of the electronic computing machines. He held many discussions [. . . ] of the kind of future problems which could be studied through the use of such machines.”

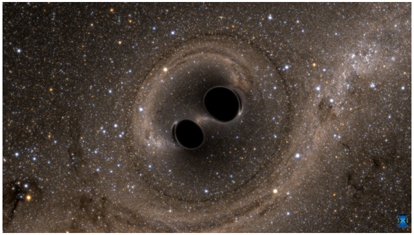
Fermi,Pasta and Ulam studied particle systems with *nonlinear* interactions

Calculations were done on the MANIAC-1 computer at Los Alamos

## And they still do. . .



Caltech/MIT/LIGO Lab



SXS, the Simulating eXtreme Spacetimes (SXS) project

(http://www.black-holes.org)

Verification of the detection of gravitational waves by numerical solution of Einstein's equations of general relativity using the "Spectral Einstein Code"

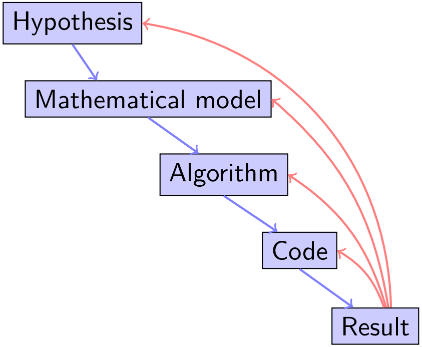
Computations significantly contributed to the 2017 Nobel prize in physics

## Scientific computing

**“The purpose of computing is insight, not numbers.”**
(`https://en.wikiquote.org/wiki/Richard_Hamming`)

- Frontiers of Scientific Computing
  - Insight into complicated phenomena not accessible by other methods
  - Improvement of models to better fit reality
  - Improvment of computational methods
  - Generate testable hypothesis
  - Support experimentation in other scientific fields
  - Exploration of new computing capabilities
  - Prediction, optimization of complex systems
- Good scientifc practice
  - Reproducibility
  - Sharing of ideas and knowledge
- Interdisciplinarity
  - Numerical Analysis
  - Computer science
  - Modeling in specific fields

## General approach



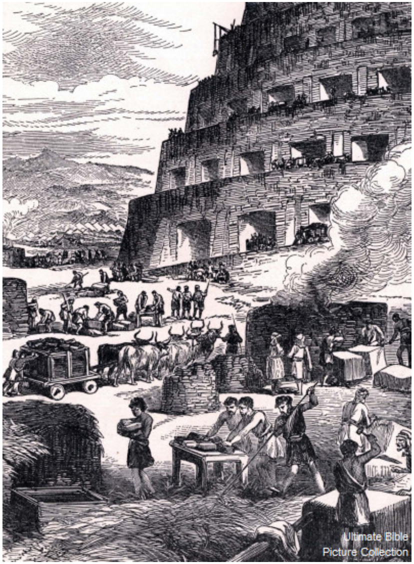
- Possible (probable) involvement of different persons, institutions
- It is important to keep interdisciplinarity in mind

## Scientific computing tools

Many of them are Open Source

- General purpose environments
  - Matlab
  - COMSOL
  - Python + ecosystem
  - R + ecosystem
  - Julia (evolving)
- “Classical” computer languages + compilers
  - Fortran
  - C, C++
- Established special purpose libraries
  - Linear algebra: LAPACK, BLAS, UMFPACK, Pardiso
  - Mesh generation: triangle, TetGen, NetGen
  - Eigenvalue problems: ARPACK
  - Visualization libraries: VTK
- Tools in the “background”
  - Build systems Make, CMake
  - Editors + IDEs (emacs, jedit, eclipse)
  - Debuggers
  - Version control (svn, git, hg)

## Confusio Linguarum



"And the whole land was of one language and of one speech. ... And they said, Go to, let us build us a city and a tower whose top may reach unto heaven. ... And the Lord said, behold, the people is one, and they have all one language. ... Go to, let us go down, and there confound their language that they may not understand one another's speech. So the Lord scattered them abroad from thence upon the face of all the earth." (Daniel 1:1-7)

## Once again Hamming

. . . of “Hamming code” and “Hamming distance” fame, who started his carrier programming in Los Alamos:

“Indeed, one of my major complaints about the computer field is that whereas Newton could say,”If I have seen a little farther than others, it is because I have stood on the shoulders of giants,” I am forced to say, “Today we stand on each other’s feet.” Perhaps the central problem we face in all of computer science is how we are to get to the situation where we build on top of the work of others rather than redoing so much of it in a trivially different way. Science is supposed to be cumulative, not almost endless duplication of the same kind of things.” (1968)

- 2017 this is still a problem

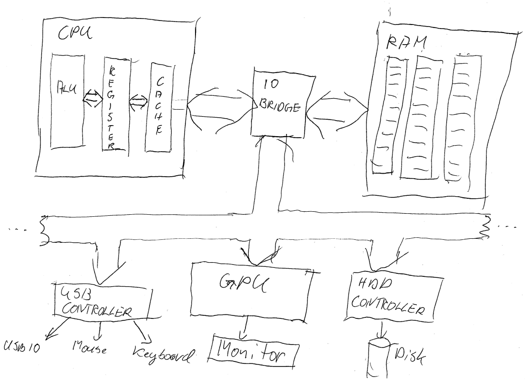
## Intended aims and topics of this course

- 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

## Sequential hardware

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

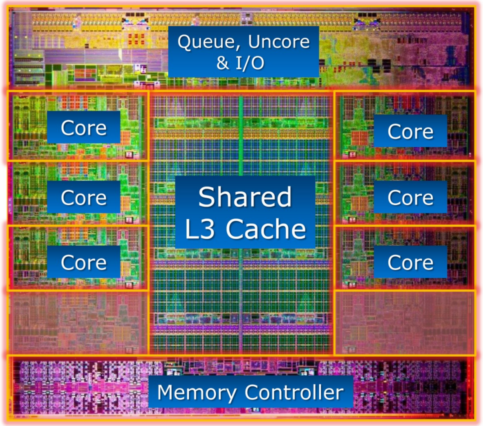
# von Neumann Architecture



- 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

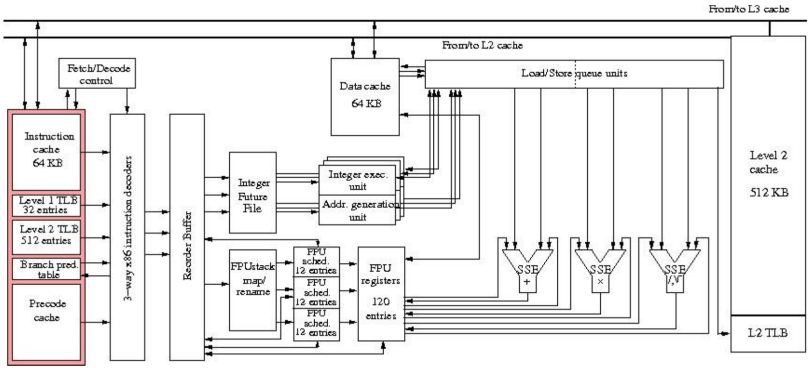
## Contemporary Architecture

- Multiple operations simultaneously “in flight”
- Operands can be in memory, cache, register
- Results may need to be coordinated with other processing elements
- Operations can be performed speculatively



Modern CPU. From: https://www.hartware.de/review_1411_2.html

## What is in a “core” ?



From: Eijkhout

# Modern CPU functionality

- Traditionally: one instruction per clock cycle
- Modern CPUs: Multiple floating point units, for instance 1 Mul + 1 Add, or 1 FMA
  - Peak performance is several operations /clock cycle
  - Only possible to obtain with highly optimized code
- Pipelining
  - A single floating point instruction takes several clock cycles to complete:
  - Subdivide an instruction:
    - Instruction decode
    - Operand exponent align
    - Actual operation
    - Normalize
  - Pipeline: separate piece of hardware for each subdivision
  - Like assembly line

## Data and code

- Stored in the same memory $\Rightarrow$ encoded in the same way
- Stored as binary numbers, most often written in hexadecimal form

## 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
```

## 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

## Registers

Processor instructions operate on registers directly

- have assembly language names names like: eax, ebx, ecx, etc.
- sample instruction: addl %eax, %edx
- Separate instructions and registers for floating-point operations

## Data caches

- Between the CPU Registers and main memory
- L1 Cache: Data cache closest to registers
- L2 Cache: Secondary data cache, stores both data and instructions
- Data from L2 has to go through L1 to registers
- L2 is 10 to 100 times larger than L1
- Some systems have an L3 cache, $\approx$10x larger than L2

## Cache line

- The smallest unit of data transferred between main memory and the caches (or between levels of cache)
- N sequentially-stored, multi-byte words (usually N=8 or 16).
- If you request one word on a cache line, you get the whole line
  - make sure to use the other items, you've paid for them in bandwidth
  - Sequential access good, "strided" access ok, random access bad
- Cache hit: location referenced is found in the cache
- Cache miss: location referenced is not found in cache
  - triggers access to the next higher cache or memory
- Cache thrashing
  - Two data elements can be mapped to the same cache line: loading the second "evicts" the first
  - Now what if this code is in a loop? "thrashing": really bad for performance
- Performance is limited by data transfer rate
  - High performance if data items are used multiple times

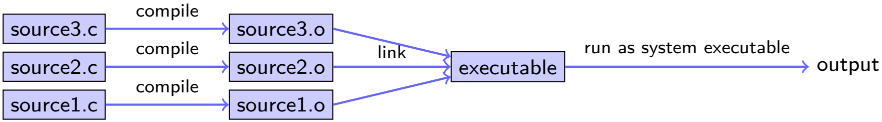
## High level computer languages

# Compiled high level languages

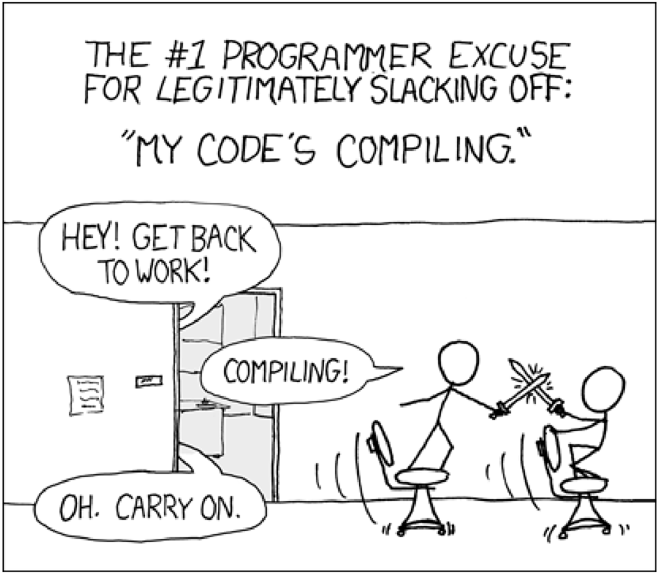
- Algorithm description using mix of mathematical formulas and statements inspired by human language
- Translated to machine code (resp. assembler) by *compiler*

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

- "Far away" from CPU $\Rightarrow$ 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. . .



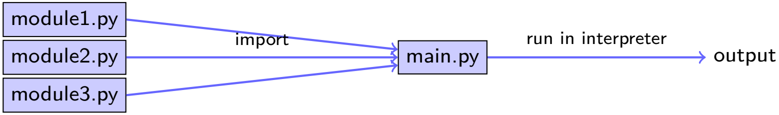
. . . from xkcd

# High level scripting languages

- Algorithm description using mix of mathematical formulas and statements inspired by human language
- Need intepreter in order to be executed

```
print("Hello world")
```

- Very far away from CPU $\Rightarrow$ usually significantly slower compared to compiled languages
- Matlab, Python, Lua, perl, R, Java, javascript
- Less strict type checking, often simple syntax, powerful introspection capabilities
- Immediate workflow: “just run”
  - in fact: first compiled to *bytecode* which can be interpreted more efficiently



## JITting to the future ?

- As described, all modern interpreted language first compile to bytecode wich then is run in the interpreter
- Couldn't they be compiled to machine code instead? – Yes, they can: Use a just in time (JIT) compiler!
  - V8 $\rightarrow$ javascript
  - LLVM Compiler infrastructure $\rightarrow$ Python/NUMBA, **Julia** (currently at 0.5)
  - LuaJIT
  - Java
  - Smalltalk
- Drawback over compiled languages: compilation delay at every start, can be mediated by caching
- Potential advantage over compiled languages: *tracing* JIT, i.e. optimization at runtime
- Still early times, but watch closely. . .

## 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...

## Summary

- Compiled languages important for high performance
- Fortran lost its momentum, but still important due to huge amount of legacy libs
- C++ highly expressive, ubiquitous, significant improvements in C++11