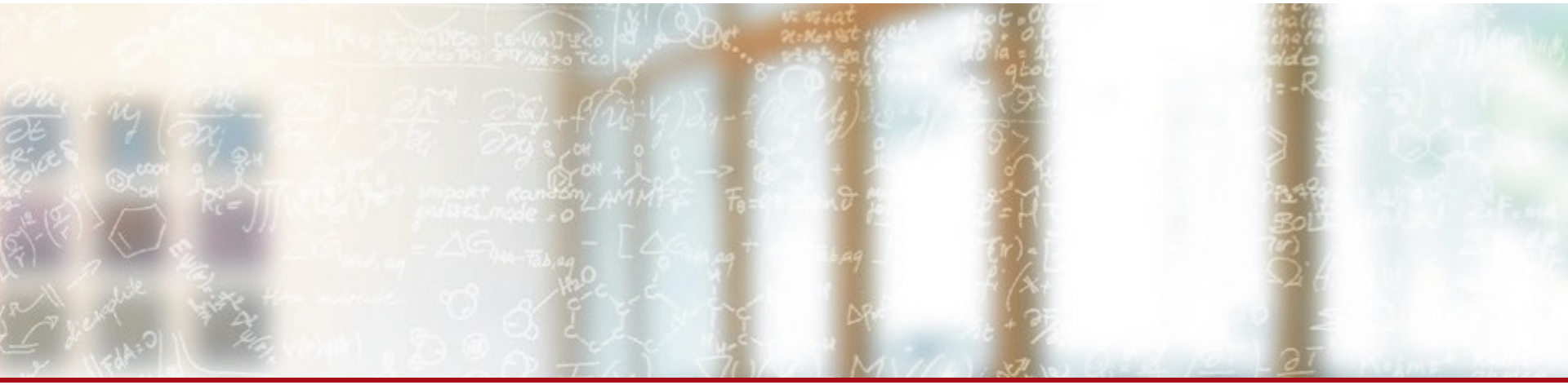
CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich



# Advanced C++ for HPC

## Advanced C++ for HPC

CSCS, Lugano, CH

# Welcome to the Advanced C++ Course

- This year we are fully online!
- Approximate times
  - 9:00 – 10:30
  - 10:45 – 12:00
  - 13:00 – 14:30
  - 14:45 – 16:15
- During the breaks
  - https://spatial.chat/s/CSCSch?sp=NetworkingSpace2021

# Practicals

- Examples and exercises can be found at

  **https://github.com/eth-cscs/examples_cpp**

- Slides can be found at

  **See event page at cscs later during the course**

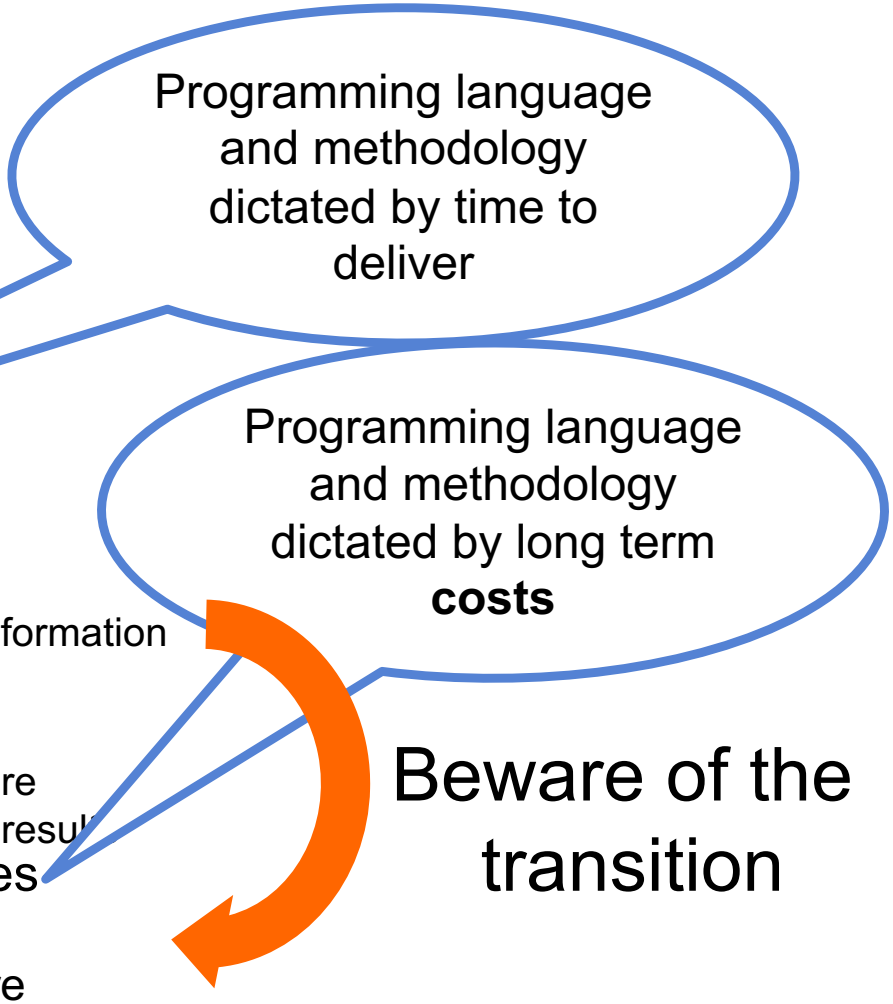
- To compile
  - `` `cd examples_cpp; mkdir build; cd build` ``
  - Run `` `cmake ../Code` ``
    - Optionally `-DCMAKE_BUILD_TYPE=debug|release`
  - Then `` `make` ``
    - `` `make help` ``
    - `` `make` ``

# Feedback

- Do you use STL?
- Do you use C++ threads?
- Do you use templates?
- Do you use constexpr?
- Do you use patterns?

# What is HPC?

- What is performance?
  - Efficiency of resource utilization?
  - Time to solution?
  - Energy to solution?
  - Algorithmic complexity?
- HPC modes
  - Research codes
    - Prototyping oriented
      - Short runs
      - Results are architecture related information
    - Scientific results
      - Long runs
      - Results independent from hardware
      - Lifetime depend on publication of resul
  - Production applications and libraries
    - Long runs
    - Results independent from hardware
    - Lifetime spans multiple hardware generations



# Programmers, CSs and SWEs

- Programmer
  - Knows how to program a computer
  - Knows at least one programming language well
- Computer Scientist
  - Understands algorithms, algorith design, data structures
  - Understands computing machineries (at least in principle)
  - Can reason about complexity, performance, information theory…
- Software Engineer
  - Values the code as a self standing product to craft
  - Worries about maintenance and refactorying cost
  - Focuses on interefaces, reusability, composability
  - Enjoy clean code and receiving reviews for improving its quality

# High Performance Code

```
for (int i = 0; i < n; ++i)
  for (int j = 0; j < m; ++j)
    for (int k = 0; k < l; ++k)
      c[i][j] += a[i][k] * b[k][j];
```
(a)

```
for (int i = 0; i < n; ++i)
  for (int k = 0; k < l; ++k)
    for (int j = 0; j < m; ++j)
      c[i][j] += a[i][k] * b[k][j];
```
(b)

- Which is faster?
  1. (a)
  2. (b)
  3. The same (the order would not matter)
  4. The same (the C++ compiler understands)

Bottom line: use BLAS!

C++ is not making writing HPC code easier!

# HPC Focuses on Hardware

- C++ started providing system related information
  - **`std::thread::hardware_concurrency`**
    - Minimum offset between two objects to avoid false sharing
  - **`std::hardware_destructive_interference_size`**
    - Maximum size of contiguous memory to promote true sharing
  - **`std::hardware_constructive_interference_size`**
    - Number of concurrent threads supported

- Not a programming model yet

# What is C++

- Programming language that allows
- Overhead-free abstractions
  - Or minimal overhead given the constraints
- Development of safe and robust code
  - Mostly by allowing good methodologies (such as RAII)
  - Constrained by performance requirements

# Why C++?


- Supported by hardware
- Integrated in established IDEs
- Many available compilers
- Huge online community
- Scales with application complexity
  - Control investment by isolating components
  - Allows API design and implementation
    - Thanks to overhead free abstractions

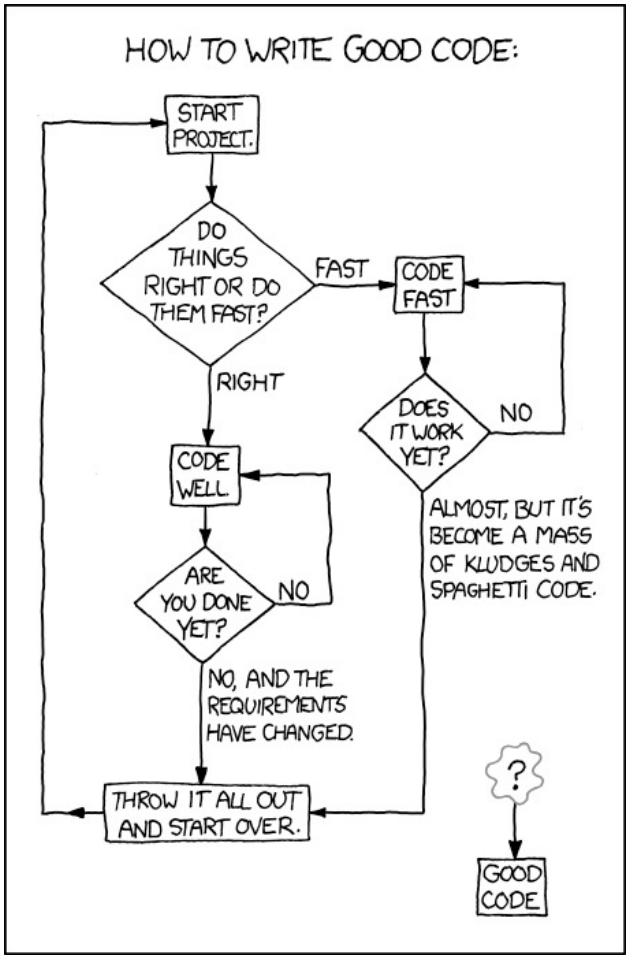
# C++…

- Rise abstraction level
  - Leads to declarative style programming
- Library adaptation/abstraction
- **Separation of concerns**
- **Fast prototyping straight into production**
- **Incremental optimization / Extensibility**
- Downside: **typical HPC tools fails**
  - Mostly due to massive inlining

# There should not be C++ without

- API design
- Unit testing
- Test Driven Development
- Systematic application of methodologies
  - Limit violations and always try to fix them
- Coding standards

- This course is in this context!

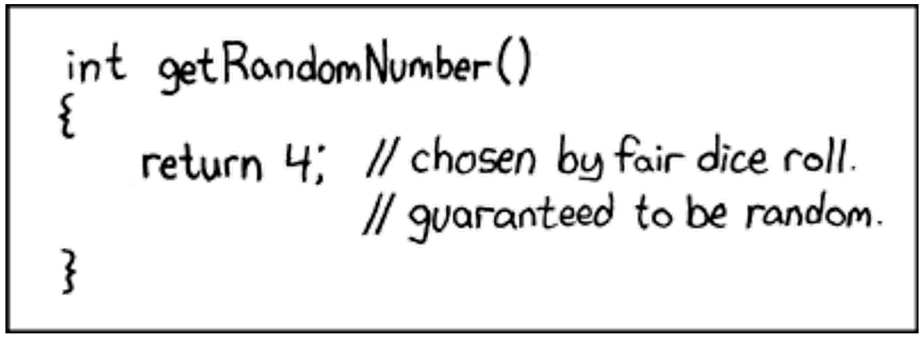


# Unit Testing

- Every component, or sub-component, function, facilty…

**Should be tested!**

- Examples:
  - A function to compute a value
  - A data-member that should satisfy an invariant
  - A value that should always be available at compile time
- Few things can not be unit-tested
  - int main()
    - Keep main at short
  - static_assert
- Use a unit-testing framework!
  - C++ does not offer one



# Test-Driven Design/Development (TDD)

- **Before developing a feature write the code that uses it!**
- Lead to better APIs
  - Think as a user that's not you
- APIs are the most **durable** things you'll develop
  - For the good or for the bad
  - Believe it or not you are API designers
- I also do DDD: Desire-Driven Design
  - Fix interface if my wish is not realizable
- KISS: Keep It Simple St..id

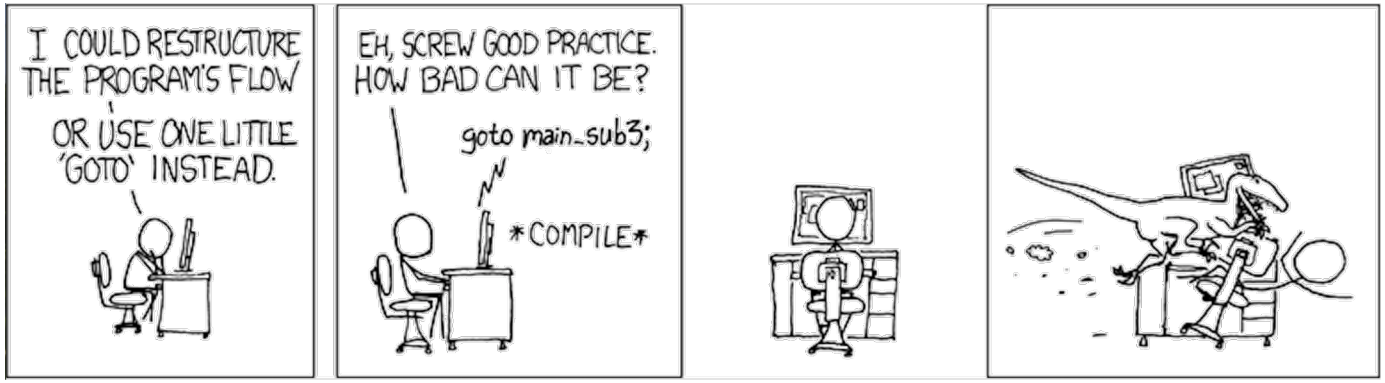
# A Good API Should

- **Be easy to use correctly**
  - Intuitive and self explanatory
    - `list.size()` vs. `list.real_size()`
    - `list.length()` vs. `list.maximum_length()`
  - Check for error conditions
  - Signal the errors appropriately
    - `static_assert`, asserts, exceptions, error codes
- **Difficult to use mistakenly**
  - Using protections to catch common misuses
  - Be simple (see next)

- Documentation is useful if
  - User read it
  - User can remember some of it (API must be simple)

# Methodologies I: Adhere to rules (examples that may apply)

- Always implement classes that provide RAII
  - A user of that class never needs a **`new`** statement
- Every function should be at most *n* lines long
- A class should do at most *n* things
- Write code that comment itself (comment the rest)
- Review code upon pull-requests
- Run automatic tests suites
- And more…



# Hello world

```
#include <iostream>

using namespace std;

/*
 * main function
 */
int main() {
  cout << "Hello world!" << endl; // Prints hello world
  return 0;
}
```

Using at global scope

Useless comment

Useless comment

Superfluous and not portable

Use '\n'

Global variable (ok in main)

# Methodologies II: Function Design (examples that may apply)

- Function names should be meaningful
- Number of arguments (< 4)
- Don't pass `const& int`
- **`set_time(int year, int month, int day)`**
  - `set_time(10,7,2012)`: what is the meaning?
- **`set_time(Year year, Month month, Day day)`**
  - `set_time(Day(10), Month(7), Year(2012))`
  - Don't provide conversions
- Call it **`set_date`** maybe?

# Methodologies III: Class Design

- Implement “Concrete Types”
  - Behaves like an int
- Rule of 0
  - A class that does not need special constructors or destructors does not need to specify any
- Rule of 3
  - If your class needs a user defined copy, copy-assign or a destructor then it probably needs all three
- Rule of 5
  - If an object needs to be moved, then all 5 are needed: copy, copy-assign, move, move-assign, destructor

# Automatic generation of special member functions

- Default constructor if no other constructor is explicitly declared
- Copy constructor if no move constructor and move assignment operator are explicitly declared
- If a destructor is declared generation of a copy constructor is deprecated
- Move constructor if no copy constructor, copy assignment operator, move assignment operator and destructor are explicitly declared
- Copy assignment operator if no move constructor and move assignment operator are explicitly declared
- Move assignment operator if no copy constructor, copy assignment operator, move constructor and destructor are explicitly declared
- Destructor

# Methodologies IV: RAII

- Encapsulate each resource into a class, where
- Constructor acquires the resource
  - Also establishes all class invariants or throws
- Destructor releases the resource and never throws
- Always use the resource via an instance of a RAII-class
- RAII-class either
  - Automatic or temporary storage duration
  - Lifetime bounded by that of an automatic or temporary object
- Well defined by initialization order, stack-unwinding…
- Made effective by move semantics

# Coding Conventions: Religion at test

- **Needed to make code readable**
  - Reduce time waste on small details
    - You'll now the kind of symbol by the syntax, example:
      - `'m_name'` is a data member
      - `'s_name'` is a static member
      - `'Name'` is a template argument
  - Common spacing across the project
- There are many coding conventions out there
- Find consensus in your team
  - Then cut it off
- Try not to be an extremist
  - The aim is to make code readable!

# Design Patterns/Idioms

- C++ provide many patterns few are relevant for HPC
  - CRTP
  - Type Erasure
  - Proxy
  - Visitor

# What We Cover… more or less

- Object Initialization
- Name resolution
- Templates
- Move semantics
- Smart Pointers
- Threads
- Tasking library
- Constexpr
- Lambdas/Functions
- STL
- Generic Programming/Concepts
- Several use cases

# Thinking C++

- Some behaviors in C++ may seems abstruse
- But they –almost always– have a good reason
  - Typically for efficiency!
  - Compatibility with C
- We need a **mental model** for reasoning about code
- Most of this model comes from low level details
  - Function call mechanisms
  - Stack frames
  - Optimization techniques
  - String matching and substitution
  - Maybe even Godelization
- Sometimes we can live with **approximate models**