

Introduction to Advanced Programming Concepts

Stefan Kluth
MPI für Physik
skluth@mpp.mpg.de

Who are we?

- Introduce ourselves
- Why are we here?
- What do we expect?
- What is our background?
- What do we know already?

APC topics

Principles of object oriented design

Advanced C++ concepts

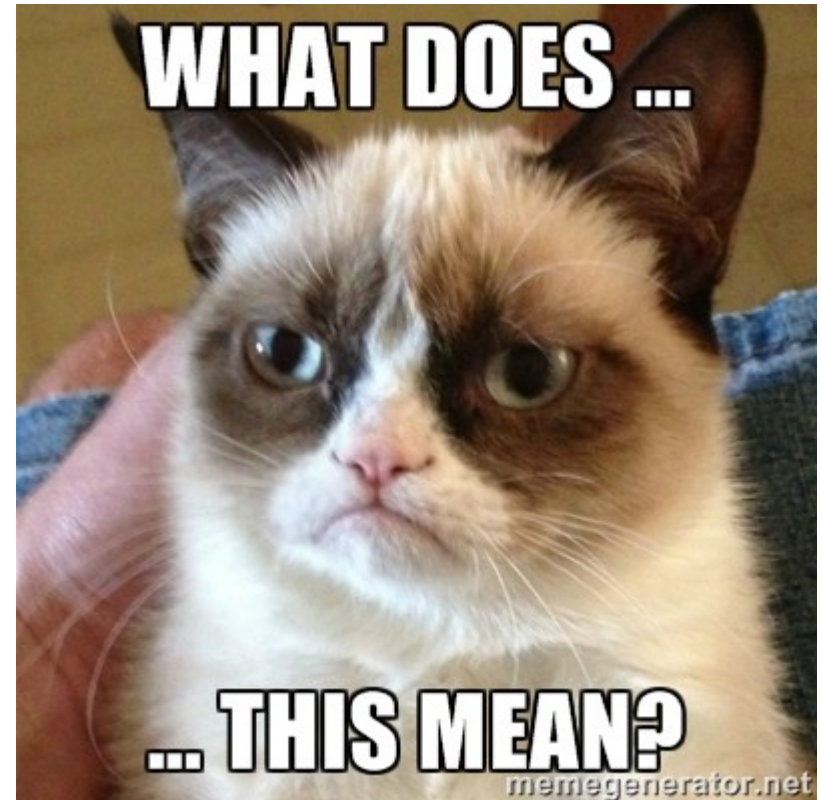
Unified modeling language

Refactoring

Unit testing

Performance, design and parallelization

Meta-template programming



Languages

Wikipedia on Programming Language:

“A programming language is a formal constructed language designed to communicate instructions to a machine, particularly a computer.”

Imperative: execution instructions for explicit algorithm
(for most people this is all)

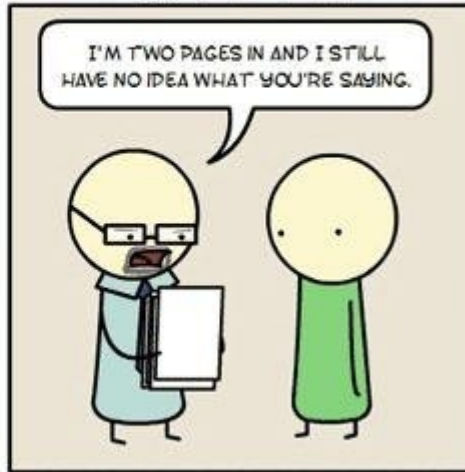
Declarative: express logic w/o fixing control flow
(Db query, regexp, configuration management,
Functional, ...)

Languages

PYTHON



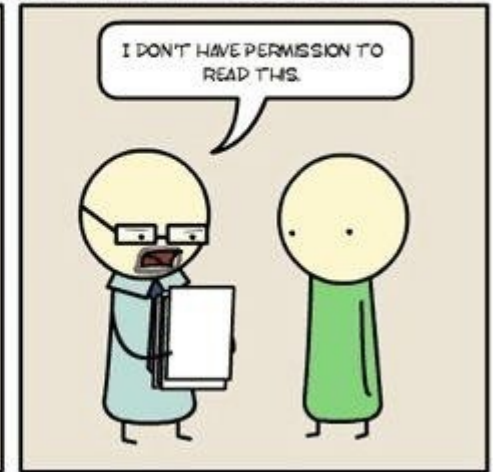
JAVA



C++



UNIX SHELL



ASSEMBLY



C



LATEX



HTML



The right tool for the job

- Scripting
 - Interpreted “line-by-line”, high-level only
 - Short “throw-away” programs (mostly)
 - Availability of libraries
- Compiled
 - Down to machine code, many optimisations possible
 - Structure your program → functions, subroutines, data structures, classes, packages
 - Long-term projects

In HEP

- Dominant languages
 - C++
 - Frameworks, DAQ, reconstruction, simulation, data reduction, analysis
 - Python
 - Analysis, data handling, workflows, system services
 - Unix shell (bash)
 - Workflows, system services
 - Fortran
 - Legacy, numerically intensive
- Many more for special purposes

In this workshop

- Concepts, methods, tools for “large” tasks
 - Experiment frameworks and their components
 - Analysis code
- Object oriented programming (OOP)
 - Established method in large systems
 - Code structuring and modularisation
 - Prepare for extensions and maintenance
 - Working in teams
- Enables unit tests and refactoring

The C++ language

- Since the 80ies as C with classes
 - B Stroustrup et al
 - ISO standards 1998, 2011 and 2014
 - OOP, generic programming (templates), system and applications, from low to high level
- Derived from C with OO constructs
 - Classes, public, private, const, inheritance, polymorphism
 - Low level efficiency with high level structures
- Best match for HEP when transition from Fortran was needed

C++ highlights: classes

A C++ class is a type description, default visibility private

```
class Complex {  
    double real;  
    double imaginary;  
public:  
    Complex( double, double );  
    ~Complex();  
    Complex& operator+( const Complex & );  
    ...  
};
```

```
Complex a(1,2); Complex b(3,4);  
Complex c= a+b;
```

Define your own types matching your problem

C++ struct: class with default visibility public members

C++ highlights: operators

Implement +, -, *, /, (), [], ... for your class

```
#include "Result.hh"
Class MyClass {
    public:
    Result operator();
};
```

```
MyClass mc; Result res= mc();
```

Not directly related to OOP

More flexibility for domain-specific-languages constructed using classes with well chosen methods and operators

Can lead to compact and powerful “C++ scripts”

Need to consider carefully expectations of users

C++ highlights: visibility

Visibility refers to the concept of private, public, protected data members or methods

public: free access via (pointer to) object (a la C struct)

private: no access except from within class code (data hiding)

protected: no access except private, and from subclasses

Key concept of OOP, hide concrete data structure, access only via methods (interface)

C++ highlights: class inheritance

Must implement “pure virtual” method signatures

Can re-implement (override) “virtual” methods

Can re-implement non-virtual methods, but w/o upcast!

```
class IFace {  
    public:  
    virtual Result doSomething( int ) = 0;  
    virtual Blob getBlob();  
};
```

```
class MyClass : public IFace {  
    public:  
    virtual Result doSomething( int );  
};
```

```
IFace* mc= new MyClass(); // type upcast  
Result res= mc->doSomething();  
Blob blob= mc->getBlob();
```

C++ highlights: const

Declare class methods to not modify data members

Ensure correctness and code readability (“The code is the comment”)

Enables multiple threads (if used consistently), since data members guaranteed to not change

```
class MyClass {  
    Data data;  
    public:  
    void doSomething() const;  
    void setData( const Data & );  
};
```

```
const MyClass mc;  
mc.doSomething();  
mc.setData( Data ); // error
```

C++ highlights: *&etcpp

C++ does not have built-in memory management

If you are used to java or scripting ...

```
#include "Variable.hh"
void func( const Variable& v );
...
{
Variable var; // constructed "on stack"
Variable* varp= new Variable(); // "on heap"
Variable& varr= var; // reference taken
func( var ); // reference to var passed
delete varp; // remove from heap
}
// var and varr out of scope, automatic deletion
```

Memory management via constructor and destructor

Memory leaks, heavy tools (valgrind), ...

C++ memory management

Old style: conventions about “object ownership”, avoid pointers(?), leaks, valgrind, bus error, segmentation violation, @#\$%

New C++11 features:

shared_ptr<Type>: manage “reference-count”, create objects, pass around multiple copies, deleted when not used anymore (ref-count=0)

unique_ptr<Type>: manage object “uniquely”, no ref-count, pass around unique instance, delete managed object when deleted

Solves most common mem.mgmt errors: no delete (mem-leak), double delete (crash), dangling pointer access (crash)

C++ highlights: templates

C++ templates (see Friday) allow “compile time polymorphism”

```
template<typename T> class Vector {
    T* rep;
    int size;
public:
    Vector() : rep(0) {}
    Vector( int _size ) : rep(new T[_size]),
                          size(_size) {}
    ~Vector() { delete[] rep; }
    // operator+-*/[] etc pp
};
```

```
Vector<double> vd(5);
Vd[0]= 0.0;
Vector< unique_ptr<MyClass> > vmc(10);
unique_ptr<MyClass> upmc( new MyClass() );
vmc[5]= upmc;
```

C++ highlights: templates

- Template compile time polymorphism $\neq$ OOP
- Tension between OOP and “meta-template-programming”
 - OOP: abstraction and modularity via inheritance
 - MTP: generic programming, common code around generic type with certain properties
 - After “instantiation” all types are concrete
- In an OOP project use templates with care
 - e.g. templated class methods can't be virtual

C++ highlights: final, override

More detail for class method declarations with inheritance

final: class or virtual method can't be overridden in subclasses

override: expect to override virtual method in parent class

```
class MyClass : public Iface {  
    public:  
    virtual Result doSomething( int ) final;  
    Blob getBlob() override;  
};
```

Enforce design choices beyond conventions

final: helps “devirtualisation” compiler optimisation

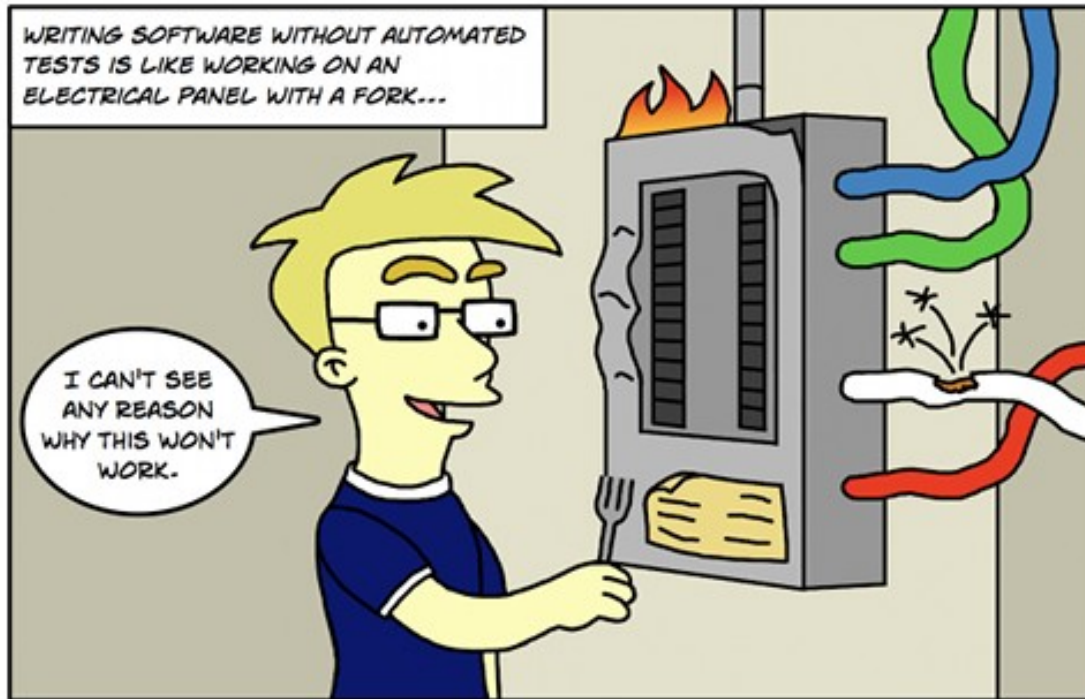
Refactoring

Change the code without changing its behaviour



The traditional approach: edit (behavior changing!) changes in, meanwhile code is broken, debug, debug, debug, validate
Now: control behavior with unit tests, change code in many small steps until ready for new feature, new tests, new feature

xUnit tests



Test runner

Test case

Test fixture

Test suite

Test execution

Test result formatting

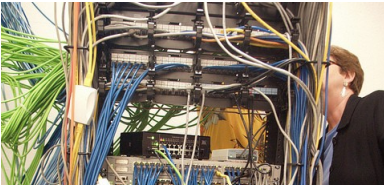
JUnit, PyUnit, googletest, ...

OOP, unit tests, refactoring



Object-oriented
programming

Refactoring,
reengineering



Unit tests,
Test driven
development

