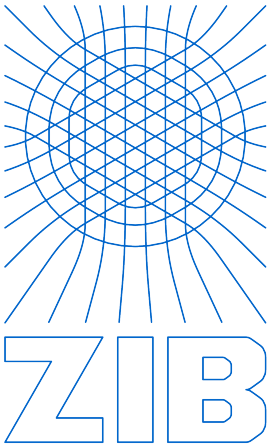
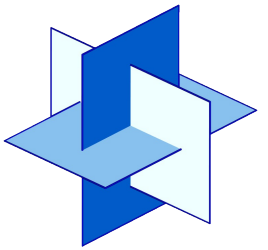
Workshop Kaskade 7 & Applications
Prerequisites: C++ Knowledge

M. Weiser,L. Lubkoll



Zuse Institute
Berlin



MATHEON

Ankara, 2013-10-03

Contents

C++ in a nutshell

- syntax
- classes
- overloading
- templates

Standard library

- containers
- algorithms
- IO

Template metaprogramming

- rationale
- metaprogramming concepts
- boost::fusion

C++ in a Nutshell

What is C++?

Taxonomy

- imperative
- procedural
- object-oriented
- generic
- functional

multi-paradigm language

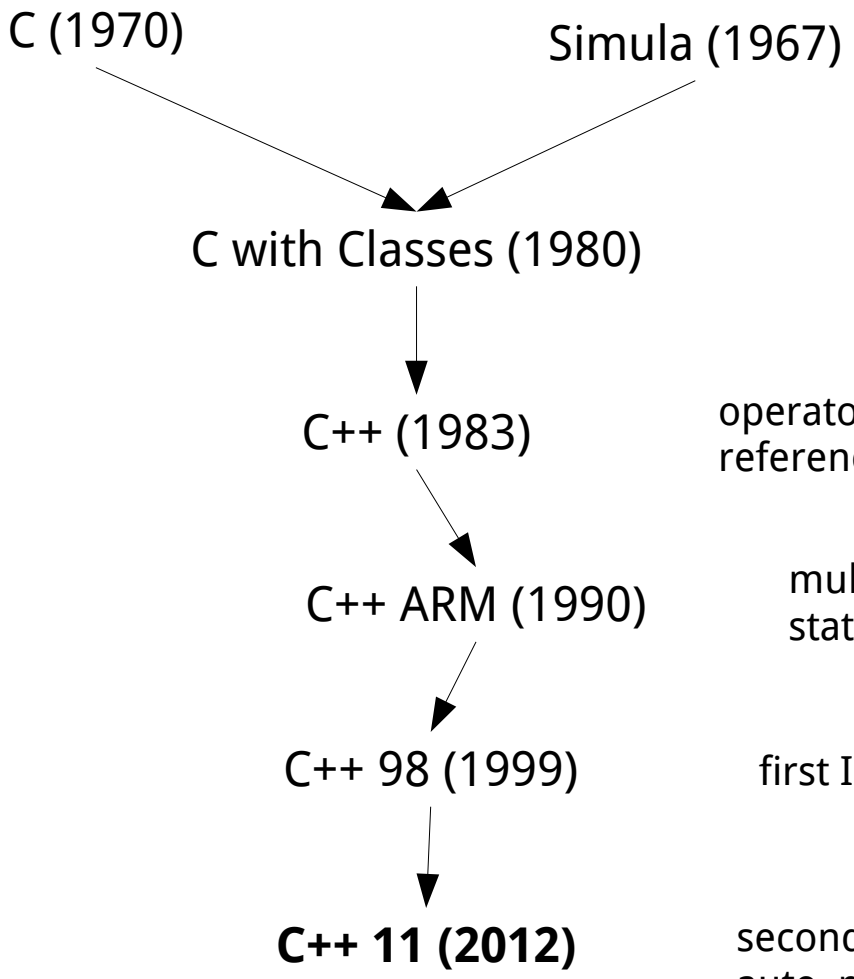
Design philosophy

- systems programming
- allow high efficiency
- provide abstractions

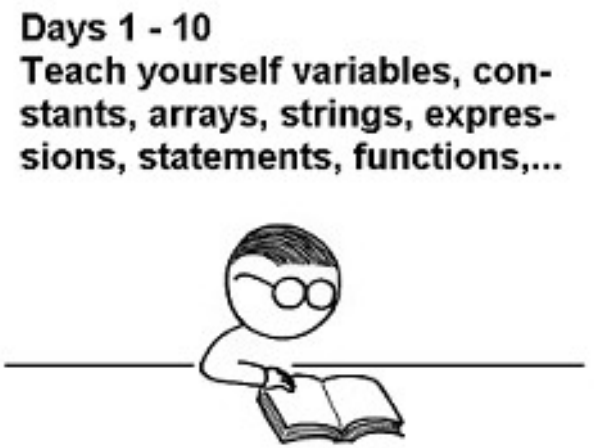
"In C++ it's harder to shoot yourself in the foot,
but when you do, you blow off your whole leg."

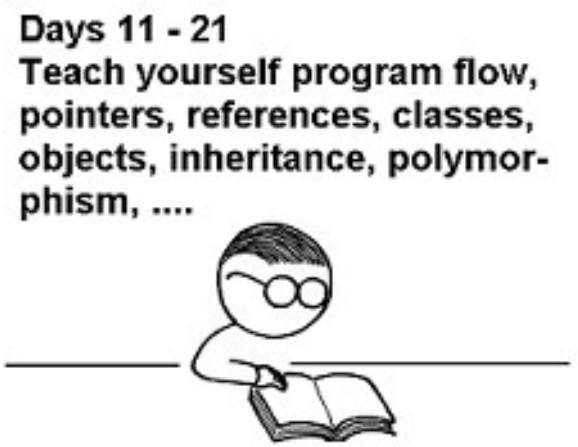
- Bjarne Stroustrup -

Short History



Teach Yourself C++ in 21 Days


Days 1 - 10
Teach yourself variables, con-
stants, arrays, strings, expres-
sions, statements, functions,...

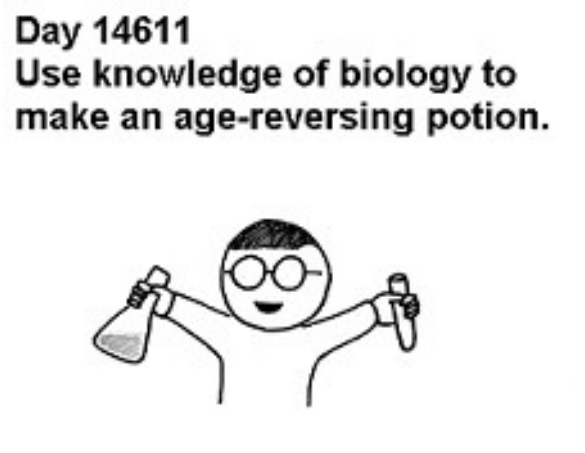

Days 11 - 21
Teach yourself program flow, pointers, references, classes, objects, inheritance, polymor-phism, ...

Days 22 - 697
Do a lot of recreational programming. Have fun hacking but remember to learn from your mistakes.

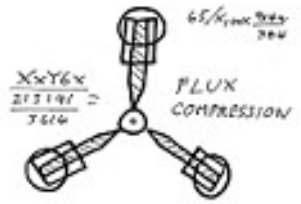
Days 698 - 3648
Interact with other programmers.
Work on programming projects
together. Learn from them.

Days 3649 - 7781
Teach yourself advanced theoretical physics and formulate a consistent theory of quantum gravity.

Days 7782 - 14611
Teach yourself biochemistry, molecular biology, genetics,...


Day 14611
Use knowledge of biology to
make an age-reversing potion.

Day 14611
Use knowledge of physics to build flux capacitor and go back in time to day 21.



„As far as I know, this is the easiest way to *Teach Yourself C++ in 21 days*"
(The Abstruse Goose, abstrusegoose.com/249)

C++: Variables

Available

Built-in types, Functions, Classes

Built-in types

int, unsigned int, double, char, ...

```
int a;                  // declaration (value of „a“ undefined!)
a = 42;                 // initialization
int b = 42;             // better both at same time
char c = 'x';
```

Declare read-only variable (const)

```
int const d1 = 42;      // must be initialized in declaration
const int d1 = a;       // same as above
```

Declare compile-time constant (constexpr)

can be understood by the compiler

```
constexpr int e = 42;
```

C++: Conditions & Loops

MATHEON

Syntax

- if: if(condition) statement

?: condition ? statement(condition=true) : statement(condition=false)

```
if(a==1) {
    a = 2;
}

if(a==1) std::cout << "Ankara" << std::endl;
else {
    a -= 1;
    std::cout << "Berlin" << std::endl;
}

std::cout << "computation " <<
(solver.failed() ? "failed" : "successful") << std::endl;
```

- while: while(condition) statement

```
while(a<1000) { b = b+1; }
```

C++: Conditions & Loops

Syntax

- for a) for(initialization; condition; increase) statement
  b) for(element : range) statement
  c) std::for_each (see STL: Algorithms)

```
std::vector<double> vec = { 1.0, 2.0, 3.0};

for(int i=0; i<vec.size(); ++i)
    std::cout << vec[i] << std::endl;

for(auto iter = vec.begin(); iter!=vec.end(); ++iter)
    std::cout << *iter << std::endl;

for(double const e : vec) std::cout << e << std::endl;
```

C++: Free Functions

MATHEON

```
double addDoubles(double a, double b);      // declaration (in .cpp-file)

double addDoubles(double a, double b) {     // definition (in .hh-file)
    return a+b;
}

void printHello() {                         // declaration + definition
    std::cout << "Hello Ankara" << std::endl;
}

constexpr int addInts(int a, int b) {       // function that can be
    return a+b;                             // evaluated during compilation
}                                           // if a and b are compile-time constants

typedef std::array<int,addInts(20,22)> MyIntArray;

int main() {
    double c = addDoubles(2.0,3.5);
    while(true) printHello();               // infinite loop :)
}
```

C++: References

(L-value) References '&'

- stores address of variable with longer lifetime than reference
- useful for large objects: avoids copying
- assignment & initialization in same place

```
int& b = a;
int const& c = b;
a = 1;
b = 2;
c = 3;
```

```
// b contains the address of a
// c contains the address of a and is fixed
    // c = b = a = 1
    // c = a = b = 2
    // error: can not assign to const&
```

- Danger: Referenced object may go out of scope! RESPONSIBILITY OF PROGRAMMER!
- Using large objects?
  i.e. a function that writes a grid to a file

```
void writeGrid(Grid grid, std::string filename);
```

unnecessary copy of grid (rel. big, >> 1 MB)

C++: References

MATHEON ZIB

(L-value) References '&'

- stores address of variable with longer lifetime than reference
- useful for large objects: avoids copying
- assignment & initialization in same place

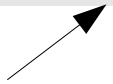
```
int& b = a;
int const& c = b;
a = 1;
b = 2;
c = 3;
```

```
// b contains the address of a
// c contains the address of a and is fixed
      // c = b = a = 1
      // c = a = b = 2
      // error: can not assign to const&
```

- Danger: Referenced object may go out of scope!
- Using large objects?
  i.e. a function that writes a grid to a file

better

```
void writeGrid(Grid const& grid, std::string filename);
```



no copy, can take temporaries
see http://herbsutter.com/2008/01/01/gotw-88-a-candidate-for-the-most-important-const/

C++: References

What to do with functions that return big objects?

```
BigObject bigObject0() { return BigObject(); }
std::unique_ptr<BigObject> bigObject1() {
    return std::unique_ptr<BigObject>(new BigObject());
}
BigObject& bigObject2() { return BigObject(); }

BigObject obj0 = bigObject0();
std::unique_ptr<BigObject> obj1( bigObject1() );
BigObject obj2 = bigObject2();
```

C++: References

What to do with functions that return big objects?

```
BigObject bigObject0() { return BigObject(); }
std::unique_ptr<BigObject> bigObject1() {
    return std::unique_ptr<BigObject>(new BigObject());
}
BigObject& bigObject2() { return BigObject(); }

BigObject obj0 = bigObject0();
std::unique_ptr<BigObject> obj1( bigObject1() );
BigObject obj2 = bigObject2();
```

create object, copy to stack, copy to obj0 => 2 copies

create pointer to object (object created on heap) => copy only pointer (cheap)

create reference to object, copy address to stack, delete object, obj2 tries to call the constructor with the deleted object => segmentation fault

C++: References

What to do with functions that return big objects?

```
BigObject bigObject0() { return BigObject(); }
std::unique_ptr<BigObject> bigObject1() {
    return std::unique_ptr<BigObject>(new BigObject());
}
BigObject& bigObject2() { return BigObject(); }

BigObject obj0 = bigObject0();
std::unique_ptr<BigObject> obj1( bigObject1() );
BigObject obj2 = bigObject2();
```

never return (l-value) references to temporary objects

most inefficient

ugly syntax

stupid, referenced object directly goes out of scope

create object, copy to stack, copy to obj0 => 2 copies

create pointer to object (object created on heap) => copy only pointer (cheap)

create reference to object, copy address to stack, delete object, obj2 tries to call the constructor with the deleted object => segmentation fault

C++: Lifetime

Scope & Lifetime

- variables "die" at end of scope (i.g. with '}')
- the scope of return values ends after the line containing the function call ends

```
int getOne() { return 1; }

int main(){
    int a = 1;

    for(int i=0; i<42; ++i)
    {
        a = b;                  // compile time error: b undefined
        int& b = a;             // b contains the address of a.
        int& c = getOne();      // c contains the address of the temporary result of getOne()
        int d = c;              // => undefined value of c => SEGMENTATION FAULT
        int const& e = getOne(); // const& extends the lifetime of temporaries to the lifetime of
                                // the reference
        int f = a;              // no problem with copies, but expensive for big objects.
    }                           // b - f are deleted.

}                               // a is deleted.
```

C++: References

R-value references ('&&')

- can only appear on right-hand side of an assignment ('=')
- can not have a name
- do not have to be in a meaningful state **at end of scope**
- admit move-semantics & perfect forwarding

Move semantics: instead of copy move data from one object to another

```
class BigObject { ...
    BigObject(BigObject&& bigObject);          // move constructor
...

};

BigObject bigObject3() { return BigObject(); }

BigObject obj4 = bigObject3(); // create object, copy address, move data
```

efficient & elegant syntax

Perfect forwarding: search via startpage.com

C++: Container

- use types as parameters, i.e. instead of

c-style container

implement specific container for each type and array size

```
class c_array_int_3
{
public:
    c_array_int_3(int a, int b, int c){
        data[0] = a; data[1] = b; data[2] = c;
    }

    int& operator[](size_t i) {
        return data[i];
    }

private:
    int[3] data;
};

c_array_int_3 myArray(1,2,3);
```

C++: Container & Templates

MATHEON ZIB

we can write

templated c++-style container

one implementation for "all" types and sizes

```
template <class Data, int n>
class array {
public:
    array(){
        for(int i=0; i<n; ++i) data[i]=0;
    }

    Data& operator[](size_t i) {
        return data[i];
    }

private:
    Data[n] data;
};

array<int,3> myArray;
```

Note:
templates, as well as all template member functions, must be defined in header-files

C++: Namespaces

MATHEON ZIB

- are used to structure code
- allow the use of functions and classes with same signature in different namespaces
- here the most important namespaces are std, boost::fusion, Dune and Kaskade

same functions in same namespace

```
void printHello() {
    std::cout << "Hello Ankara" << std::endl;
}

void printHello() {                    // same name and arguments => on function call
    std::cout << "Hello Berlin" << std::endl;  // compiler does not know which
}                                      // printHello-function is meant
```

vector class in namespace 'std'

```
template <class Type, class Allocator=std::allocator<Type> >
class vector;
```

vector class in namespace 'boost::fusion'

```
template <typename T0=unspecified, typename T1=unspecified, ... ,
          typename TN=unspecified>
class vector;
```

C++: Pointers

Raw pointers ('*', see abstrusegoose.com/483 „bad boy“)

a contains address, ***a** contains value

Danger of memory leaks and segmentation faults

```
int* a = new int(5);
*a = *a + 1;
a = a + 1;
delete a;
```

```
// a contains the value 6
// a points to the next entry in memory
// remove allocated storage
```

Use raw pointers for pointing issues only

```
struct PersonalTrainer { …

    void setTrainedPerson(Person& person_) {
        person = &person_;
    }

private:
    Person* person;
};
```

```
// big person => copy is expensive
// use & to access address
```

Rule of thumb: - NEVER USE "NEW" WITHOUT MATCHING "DELETE"!
- NEVER USE "NEW", USE SMART POINTERS INSTEAD!

C++: Smart Pointers

Smart pointers

- similar to references, but may point to different objects
- initialization without assignment possible (nullptr)

use std::unique_ptr if only one pointer should point to an object

```
#include<memory>                                  // std::unique_ptr
#include <utilitiy>                               // std::move

std::unique_ptr<int> p(new int(5));               // resp. p(std::make_unique<int>(5))
std::unique_ptr<int> p0(p);                       // compile time error: no constructor
std::unique_ptr<int> p1(std::move(p));            // makes p an r-value reference
                                                  // => p contains null pointer (nullptr)
std::unique_ptr<int> p2(p1.get());                // shared unique_ptr => double free error
std::unique_ptr<int> p3(p1.release());            // ok, p1 releases pointer
int d = *p3;                                      // d = 5
```

C++: Smart Pointers

Smart pointers

- similar to references, but may point to different objects
- initialization without assignment possible (nullptr)

use std::shared_ptr if multiple pointers should point to an object

```
#include<memory>                                    // std::shared_ptr

std::shared_ptr<int> p(std::make_shared(5));
std::shared_ptr<int> p1(p);                         // ok for shared pointers
std::shared_ptr<int> p2(p1.get());                  // pointers don't know about each other
                                                    // => double free error
int d = *p1;                                        // d = 5
```

Double Free Error

MATHEON

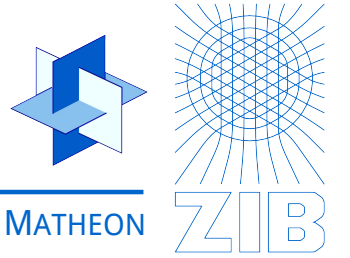
```
*** glibc detected *** test3: free(): invalid pointer: 0x0804c01c ***
======= Backtrace: =========
/lib/libc.so.6(+0x6ff0b)[0xb7526f0b]
/usr/lib/libstdc++.so.6(_ZdlPv+0x1f)[0xb771eb2f]
test3[0x8048cc8]
test3[0x804891d]
/lib/libc.so.6(__libc_start_main+0xf3)[0xb74d0003]
test3[0x80489d1]
======= Memory map: ========
08048000-0804a000 r-xp 00000000 08:03 4068942    /home/lars/ZIB/workspace/test3/Release/test3
0804a000-0804b000 r--p 00001000 08:03 4068942    /home/lars/ZIB/workspace/test3/Release/test3
0804b000-0804c000 rw-p 00002000 08:03 4068942    /home/lars/ZIB/workspace/test3/Release/test3
0804c000-0806d000 rw-p 00000000 00:00 0          [heap]
b74b4000-b74b7000 rw-p 00000000 00:00 0
b74b7000-b761e000 r-xp 00000000 08:01 917844     /lib/libc-2.14.1.so
b761e000-b7620000 r--p 00167000 08:01 917844     /lib/libc-2.14.1.so
b7620000-b7621000 rw-p 00169000 08:01 917844     /lib/libc-2.14.1.so
b7621000-b7624000 rw-p 00000000 00:00 0
b7624000-b7640000 r-xp 00000000 08:01 917846     /lib/libgcc_s.so.1
b7640000-b7641000 r--p 0001b000 08:01 917846     /lib/libgcc_s.so.1
b7641000-b7642000 rw-p 0001c000 08:01 917846     /lib/libgcc_s.so.1
b7642000-b766b000 r-xp 00000000 08:01 921827     /lib/libm-2.14.1.so
b766b000-b766c000 r--p 00028000 08:01 921827     /lib/libm-2.14.1.so
b766c000-b766d000 rw-p 00029000 08:01 921827     /lib/libm-2.14.1.so
```

possible reason:
smart pointer to an object that gets destroyed by some other source

Double Free Error


```
b766d000-b774f000 r-xp 00000000 08:01 1189091    /usr/lib/libstdc++.so.6.0.16
b774f000-b7753000 r--p 000e2000 08:01 1189091    /usr/lib/libstdc++.so.6.0.16
b7753000-b7754000 rw-p 000e6000 08:01 1189091    /usr/lib/libstdc++.so.6.0.16
b7754000-b775b000 rw-p 00000000 00:00 0
b777c000-b777f000 rw-p 00000000 00:00 0
b777f000-b779e000 r-xp 00000000 08:01 917515     /lib/ld-2.14.1.so
b779e000-b779f000 r--p 0001f000 08:01 917515     /lib/ld-2.14.1.so
b779f000-b77a0000 rw-p 00020000 08:01 917515     /lib/ld-2.14.1.so
bfa92000-bfab3000 rw-p 00000000 00:00 0          [stack]
ffffe000-fffff000 r-xp 00000000 00:00 0          [vdso]
```

Double Free Error in Eclipse (eclipse.org)




```
*** glibc detected *** /home/lars/ZIB/workspace/test3/Release/test3:
free(): invalid pointer: 0x0804c01c ***
======= Backtrace: =========
/lib/libc.so.6(+0x6ff0b)[0xb74d7f0b]
/usr/lib/libstdc++.so.6(_ZdlPv+0x1f)[0xb76cfb2f]
/home/lars/ZIB/workspace/test3/Release/test3[0x8048cc8]
/home/lars/ZIB/workspace/test3/Release/test3[0x804891d]
/lib/libc.so.6(__libc_start_main+0xf3)[0xb7481003]
/home/lars/ZIB/workspace/test3/Release/test3[0x80489d1]
======= Memory map: ========
08048000-0804a000 r-xp 00000000 08:03 4068942    /home
/lars/ZIB/workspace/test3/Release/test3
```

C++: Classes

MATHEON

Classes (user defined types)

can hold data

should reflect minimal independent sub-structures

```
class Point {
public:                                          // following code is visible to user
    Point(double x_, double y_, double z_)       // constructor
     : x(x_), y(y_), z(z_)                       // initialization
    {}

    void reset(){                                // member function
        x = y = z = 0;
    }

private:                                         // hidden from user
    double x,y,z;                                // declaration
};

Point p(1.0,1.,1.);
p.reset();
```

C++: Classes

Classes (user defined types)

new keywords: default, delete

```
class Point {
public:                                          // following code is visible to user
    Point() = default;                           // use default  default constructor

    Point(Point const&) = delete;                // no copy constructor
    Point& operator=(Point const& /*other*/) = delete;
                                                 // no copy by assignment
    ...
private:                                                    // hidden from user
    double x,y,z;                                           // declaration
};

Point p(1.0,1.,1.);
p.reset();
Point q;                                         // same as Point q(0.,0.,0.);
```

C++: Classes

static member variables

not part of object, exist independently
do not have an address until declared outside the class

```
class Point {
public:                                          // following code is visible to user
    static constexpr int dim = 3;
    ...
};

static constexpr int Point::dim;                 // now Point::dim has an address

std::cout << "dim: " << Point::dim << std::endl;
```

C++: Classes

Inheritance

build hierarchies of classes that make sense independently

```
// Base class
class Shape {
public:
    void setWidth(double w) {
        width = w;
    }

    void setHeight(double h) {
        height = h;
    }

    double getHeight() const {
        return height;
    }

    double getWidth() const {
        return width;
    }

protected:
    double width;
    double height;
};
```

```
// Derived class
class Rectangle: public Shape {
public:
    Rectangle() = default;

    double getArea() {
        return (width * height);
    }
};

double computeBoundingBox(Shape const& shape){
    return shape.getWidth()*shape.getHeight();
}

int main() {
    Rectangle rect = Rectangle();
    Shape shape0 = rect;        // shape is a shape
    Shape& shape = rect;        // shape is a rectangle
    shape.setWidth(1.0);
    shape.setHeight(1.0);
    double area =
    dynamic_cast<Rectangle>(shape).getArea();
}
```

C++: Classes

MATHEON ZIB

Virtual inheritance

use (possibly abstract) base classes as interface

```
// Base class
class Shape {
public:
    void setWidth(double w) {
        width = w;
    }

    void setHeight(double h) {
        height = h;
    }

    double getHeight() const {
        return height;
    }

    double getWidth() const {
        return width;
    }

virtual double getArea() const = 0;

protected:
    double width, height;
};
```

```
// Derived class
class Rectangle: public Shape {
public:
    Rectangle() = default;

    virtual double getArea() const {
        return (width * height);
    }
};

double computeBoundingBox(Shape const& shape){
    return shape.getWidth()*shape.getHeight();
}

int main() {
    Rectangle rect;
    Shape shape0 = rect;
    Shape& shape = rect;
    shape.setWidth(1.0);
    shape.setHeight(1.0);
    double area = shape.getArea();
}
```

no implementation
=> class `Shape` is abstract

C++: Operators

MATHEON ZIB

Operators

- arithmetic operators, i.e. =,+, -,*, /, ...
- relational operators, i.e. ==, !=, >, <, ...
- logical operators, !, &&, ||
- pointer operators, *, &, ->
- function operator (), i.e. for functors (classes whose objects can be used as functions)
- ...

see https://en.wikipedia.org/wiki/Operators_in_C_and_C++ for an overview

Syntax for member operators (here for '*=' and '=')

```
class Rectangle { ...
    Rectangle& operator*=(double scaling) {
        this->getWidth() *= scaling;
        this->getHeigth() *= scaling;
        return *this;
    }
};
```

C++: Operators

Operators

- arithmetic operators, i.e. =,+, -,*, /, ...
- relational operators, i.e. ==, !=, >, <, ...
- logical operators, !, &&, ||
- pointer operators, *, &, ->
- function operator (), i.e. for functors (classes whose objects can be used as functions)
- ...

see https://en.wikipedia.org/wiki/Operators_in_C_and_C++ for an overview

Syntax for free operators (here for '==')

```
void operator==(Rectangle const& rect1, Rectangle const& rect2)
{
    if( ( rect1.getHeight() != rect2.getHeight() ) ||
        ( rect1.getWidth()  != rect2.getWidth() ) return false;
    ... // more checks

    return true;
}
```

C++: Overloading

- same function name, different arguments
- typical use-case: overloading free operators such as operator<<, operator<, ...

overloading the stream operator <<

write std::vector<int> to console:

```
for(int e : vec) std::cout << e << " ";
std::endl;
```

with overloading of operator<<:

```
std::ostream& operator<<(std::ostream& os,
                         std::vector<int> const& vec)
{
    for(int e : vec) os << e << " ";
    os << std::endl;
    return os;
}
...
std::cout << vec << std::endl;
```

C++: Overloading

Summary:

- same name, different arguments
- legal in C++ (not in C)
- compiler "mangles" arguments into name to make linker happy
  i.e. _ZlsRSoRKSt6vectorIiSaIiEE, _ZlsRSoRKSt6vectorIdSaIdEE

```
std::ostream& operator<<(std::ostream& os,
                         std::vector<int> const& vec);

std::ostream& operator<<(std::ostream& os,
                         std::vector<double> const& vec);
```

- can not overload for only built-in types
- do not overload in counterintuitive ways

C++: File Structure

- Division into *.cpp and *.hh files.
- Point class: Put declarations into point.hh file

```
#ifndef POINT_HH_                                          // include guard: avoids
#define POINT_HH_                                          // multiple inclusion of
                                                           // header file
#include <array>

class Point
{
    public:                                                // visible to user
        Point(double x, double y, double z);               // constructor
        // reset point to origin
        void reset();                                      // member function
        // access coordinates
        double& operator[](size_t coordinateIndex);
        // read-only access
        double const& operator[](size_t coordinateIndex) const;

    private:                                               // hidden from user
        std::array<double,3> position;
};

#endif                                                     // end include guard
```

function does not change local member variables

C++: File Structure

- Division into *.cpp and *.hh files.
- Point class: Put declarations into point.cpp file

```
#include „point.hh“

Point::Point(double x, double y, double z) {
    position[0] = x;
    position[1] = y;
    position[2] = z;
}

Point::reset() { position.fill(0.0); }

double& Point::operator[](size_t coordinateIndex)
{
    return position[coordinateIndex];
}

double const& Point::operator[](size_t coordinateIndex) const
{
    return position[coordinateIndex];
}
```

Gerald Weinberg, computer scientist and author:

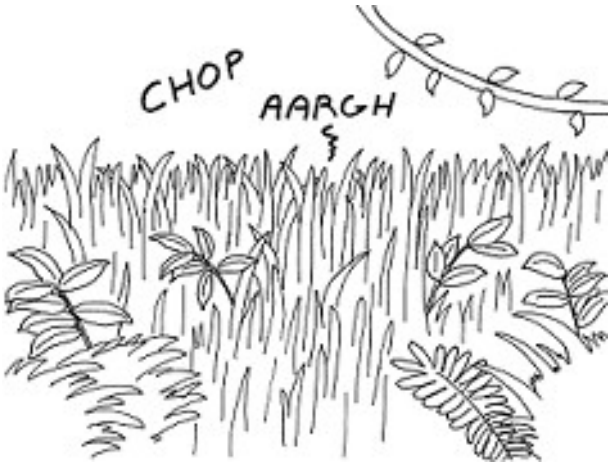
"If builders built buildings the way programmers write programs,
then the first woodpecker that came along would destroy civilization."

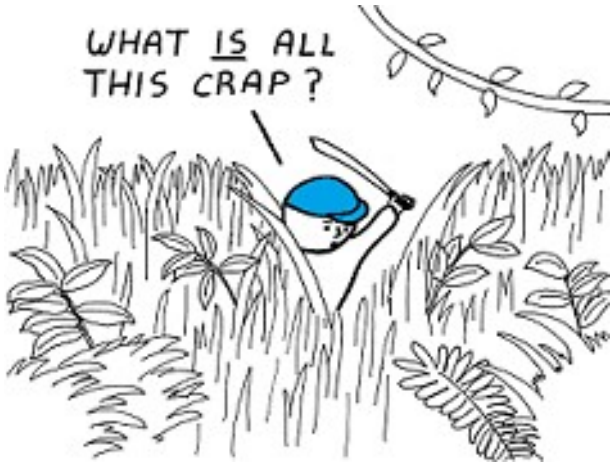
ağaçkakan

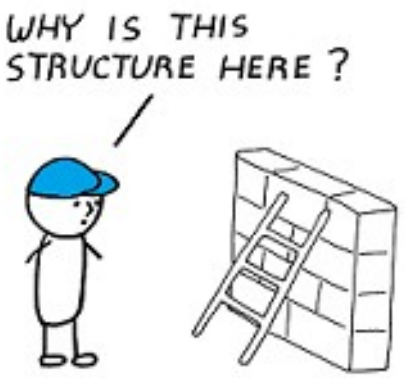
Robert C. Martin , „Clean Code: A Handbook of Agile Software Craftmanship“:

"Have you ever been significantly impeded by bad code? If you are a programmer of any experience then you've felt this impediment many times. [...]. We call it wading. We wade through bad code. We slog through a morass of tangled brambles and hidden pitfalls. We struggle our way, hoping for some hint, some clue, of what is goint on; but all we see is more and more senseless code."


CHOP
CHOP


CHOP
AARGH


WHAT IS ALL
THIS CRAP?


WHY IS THIS
STRUCTURE HERE ?


WHERE COULD THIS BRIDGE
POSSIBLY LEAD ?


THIS SIGN DOESN'T HELP ME MUCH.
//THIS IS BRIDGE


WHAT A HORRIBLY DESIGNED
STREET, MOST INEFFICIENT.

taken from abstrusegoose.com/432

Robert C. Martin , „Clean Code: A Handbook of Agile Software Craftmanship“:

"Have you ever been significantly impeded by bad code? If you are a programmer of any experience then you've felt this impediment many times. [...]. We call it wading. We wade through bad code. We slog through a morass of tangled brambles and hidden pitfalls. We struggle our way, hoping for some hint, some clue, of what is goint on; but all we see is more and more senseless code."

"Of course you have been impeded by bad code. So then

Robert C. Martin , „Clean Code: A Handbook of Agile Software Craftmanship“:

"Have you ever been significantly impeded by bad code? If you are a programmer of any experience then you've felt this impediment many times. [...]. We call it wading. We wade through bad code. We slog through a morass of tangled brambles and hidden pitfalls. We struggle our way, hoping for some hint, some clue, of what is goint on; but all we see is more and more senseless code."

"Of course you have been impeded by bad code. So then – why did you write it?"

Robert C. Martin , „Clean Code: A Handbook of Agile Software Craftmanship“:

"Were you trying to go fast? Were you in a rush? Probably so. Perhaps you felt that you didn't have time to do a good job; that your boss would be angry with you if you took the time to clean up your code. Perhaps you were just tired of working on this program and wanted it to be over. Or maybe you looked at the backlog of other stuff that you had promised to get done and realized that you needed to slam this module together so you could move on to the next. We've all done it."

Robert C. Martin , „Clean Code: A Handbook of Agile Software Craftmanship“:

"Were you trying to go fast? Were you in a rush? Probably so. Perhaps you felt that you didn't have time to do a good job; that your boss would be angry with you if you took the time to clean up your code. Perhaps you were just tired of working on this program and wanted it to be over. Or maybe you looked at the backlog of other stuff that you had promised to get done and realized that you needed to slam this module together so you could move on to the next. We've all done it."

"We've all looked at the mess we've just made and then have chosen to leave it for another day. We've all felt the relief of seeing our messy program work and deciding that a working mess is better than nothing. We've all said we'd go back and clean it up later.

Of course, in those days we didn't know LeBlanc's law: LATER EQUALS NEVER."

Difficulties in programming

1) Implicit assumptions by the programmer that are not well documented
   "One man's constant is another man's variable." (Alan Perlis)
2) Obscure variable, class and function names
3) Counterintuitive names, behaviour, ...
4) Unclear general library/program structure
5) (Premature) Optimization
6) ...

Necessary requirement for larger projects:
Coding Standards/Clean Code

Robert C. Martin , "Clean Code: A Handbook of Agile Software Craftmanship":

- "In software, 80% or more of what we do is quaintly called *maintenance*, the act of repair."

Expressive Naming

1) expressive function, variable and class names
2) document only if information can not be expressed by 1)

```
/// Returns the sum of a0 and a1.
double foo(double a0, double a1)
{
    return a0+a1;
}

// better

double add(double x, double y) {
    return x+y;
}
```

Clean Code

MATHEON

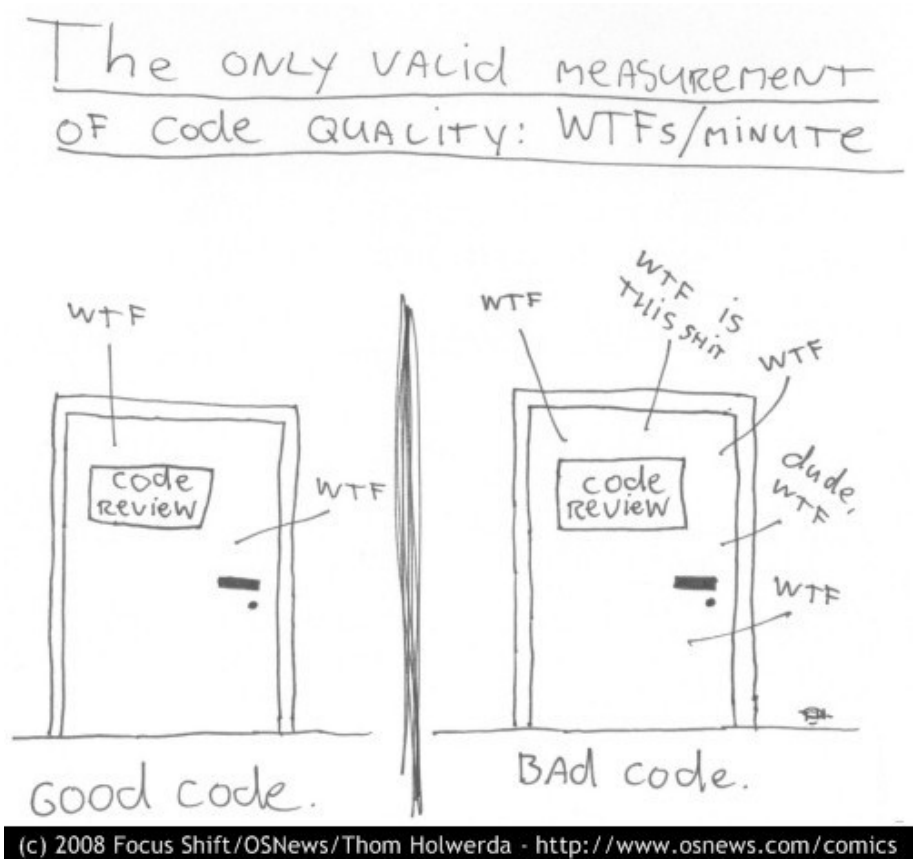
```
for(int i = 0; i<dim; ++i)
{
  if(container.vn[ vid[i] ].isRelevant)
  {
    m[i] = (tangs[evid[i]] * ne) / (tangs[evid[i]] * te);
    policy.apply(m[i]);
  }
  else m[i] = 0;
}
...
```

What is going on here?

Clean Code

```
for(int i = 0; i<dim; ++i)
{
  if(container.vn[ vid[i] ].isRelevant)
  {
    m[i] = (tangs[evid[i]] * ne) / (tangs[evid[i]] * te);
    policy.apply(m[i]);
  }
  else m[i] = 0;
}
...
```

```
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/* fill right hand side */
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
// first determine the desired gradients and apply threshold
for(int vertexIdInEdge = 0; vertexIdInEdge<dim; ++vertexIdInEdge)
{
  if(container.vertexNormals[ globalEdgeVertexIds[vertexIdInEdge] ].isRelevant)
  {
    desiredGradients[vertexIdInEdge] =
                  (tangents[localEdgeVertexIds[vertexIdInEdge]] * faceNormal) /
                  (tangents[localEdgeVertexIds[vertexIdInEdge]] * edgeDirection);
    policy.applyGradientThreshold(desiredGradients[vertexIdInEdge]);
  }
  else desiredGradients[vertexIdInEdge] = 0;
}
...
```


The ONLY VALid MEASUREMENT OF Code QUALITY: WTFs/MINUTE
WTF
code review
WTF
Good code.
WTF
WTF IS THIS SHIT
WTF
code review
dude, WTF
WTF
BAd code.
(c) 2008 Focus Shift/OSNews/Thom Holwerda - http://www.osnews.com/comics

Clean Code

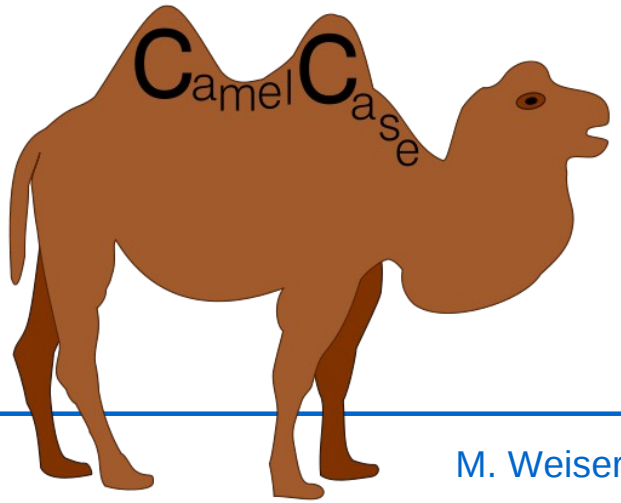
MATHEON

Standardize syntax, i.e. for Kaskade7 this means

1) For type names we always use "UpperCamelCase",
   i.e. "class VariableSetDescription".
2) For functions, objects, ... we always use "lowerCamelCase"
   i.e. "VariableSetDescription variableSetDescription".
3) We never use tabs, instead we use whitespaces
   (you can tell your editor to replace tabs with whitespaces).
4) Access on private member variables via get..., set...
   i.e. access private member **`int dim`** of **`class Point`** via

```
Point& setDim(int dim_);
int getDim() const;
```

5) curly braces start on a new line (not rigorously done on the slides because of lacking space)
6) ...



Clean Code

Donald Knuth, 1974:
"Programmers waste enormous amounts of time thinking about or worrying about, the speed of noncirtical parts of their programs, and these attempts at efficiency actually have a strong impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil.
Yet we should not pass up our opportunities in that critical 3%."

Clean Code

- Think about overall structure and modularity. Professionally designed software does not require much optimization.
- If you have a clever idea, forget it. („Premature optimization is the root of all evil!“)
- Premature pessimization is not better,
  i.e. if you pass a reference and are not sure about life time,
  do not change this reference to a copy of your object,
  instead think about how to guarantee adequate life time.
- Programming is about being scrupulous and honest, not about begin creative!
  i. e. think about your code and test it in different situations.

Clean Code

What is more important, readability or performance?

1) performance matters, but
2) sub-optimal performance, good readability
   => code works and can be used by others
3) optimize only if it pays off
   - almost everything can be optimized for performance
   - in most cases this will save some milliseconds in a program that runs for some seconds/minutes (=> do not optimize)

"The most impressive performance increase is the not-working to working transition."

- John Ousterhout -

Clean Code

From „C++ Coding Standards: 101 Rules Guidelines, and Best Practices“ (Herb Sutter, Andrei Alexandrescu):

1. Compile cleanly at high warning levels.
3. Use a version control system (svn, git, ...)
4. Invest in code reviews.
5. Give one entity one cohesive responsibility.
6. **Correctness, simplicity, and clarity come first.**
11. Use const proactively.
17. Avoid magic numbers
32. Be clear what kind of class you're writing.

    "The most important single aspect of software development is to be clear about what you are trying to build."

    (Bjarne Stroustrup, developer of C++)
68. Assert liberally to document internal assumptions and invariants

...

Literature

C++

- www.cppreference.com
- Herb Sutter: www.gotw.ca
  "Exceptional C++"
  "More Exceptional C++"
- Alexei Alexandrescu : "Modern C++ Design"
- http://www.drdobbs.com/cpp
- C++ In-Depth Series

Coding Standards/Clean Code

- Herb Sutter, Andrei Alexandrescu, "C++ Coding Standards"
- Robert C. Martin, "Clean Code: A Handbook of Agile Software Craftmanship"

Alan Perlis, computer scientist

„Get into a rut early: Do the same process the same way. Accumulate idioms. **Standardize.** The only difference between Shakespeare and you was the size of his idiom list – not the size of his vocabulary.“

„Fools ignore complexity. Pragmatists suffer it. Some can avoid it. Geniuses remove it.“

Standard Library

containers, algorithms and IO

"Just because the standard provides a cliff in front of you,
you are not necessarily required to jump off it."
Norman Diamond

STL: Containers

MATHEON ZIB

std::array

fixed size array (no insertion or removal of elements)
constant time random access

```
std::array<int,3> = { 1, 2, 3 };
```

std::list

insertion & removal of elements in constant time
linear complexity for accessing elements (slow!)

std::map

sorted, associative container
search, insertion, removal in logarithmic complexity

```
std::map<std::string,char> gradeList;
gradeList["Martin"] = 'A';
gradeList["Lars"] = 'A';
```

STL: Containers

MATHEON

std::vector

„dynamic“ array
constant time random access
insertion and removal of elements in linear complexity wrt. distance to the end

```
std::vector<double> v0 = { 1.1, 2.2, 3.3 };

std::vector<int> v1(20);
std::vector<int> v2,v3;
v2.resize(20);

for(int i=0; i<20; ++i)
{
    v1[i] = v2[i] = i;
    v3.push_back(i);            // most inefficient, allocate storage + store value
}
```

"By default, use **vector** when you need a container"
(Bjarne Stroustrup, developer of C++)

more about stl-containers: www.cppreference.com or any C++-book

STL: Algorithms

- operate on stl-containers and stl-compliant containers, i.e. containers that provide iterators to access data, provide first-class or move-semantics
- provide widely used basic functionalities

std::max, std::min

```
int a=3, b=4;
int c = std::max(a,b)                                   // c = 4
```

std::max_element, std::min_element;

```
std::vector<int> vec = { 1, 2, 3 };
int a = *std::min_element(vec.begin(), vec.end());      // a = 1
int b = *std::max_element(vec.begin(), vec.begin()+2);  // b = 2
```

STL: Algorithms

std::for_each

apply function to range

```
void modifyIf22(double& d)
{
    if(d = 2.2) d = -1;
}

std::vector<double> vec = { 1.1, 2.2, 3.3 };
std::for_each(vec.begin(), vec.end(), &modifyIf22);
```

same with lambda-expression

```
std::for_each(vec.begin(), vec.end(),
    [](double& d)
    {
        if(d=2.2) d=-1;
    }
);
```

STL: Algorithms

MATHEON ZIB

more algorithms

- find/find_if/find_if_not
- find_first_of
- all_of/none_of/any_of
- copy/copy_if
- transform
- swap
- unique
- sort
- lower_bound/upper_bound
- accumulate
- ...

STL-algorithms are

- **FAST**
- **VERIFIED**
- **WIDELY KNOWN**

more about stl-algorithms: www.cppreference.com

STL: Input/Output

general idea: first write to stream-objects, then print to display/file

- std::cout // write to terminal
- std::ofstream, std::iostream // write to file
- std::istream, std::iostream // read from file

std::cout

global object

```
std::cout << "Nice to be in Ankara." << std::endl;
```

std::ofstream

```
std::ofstream os("filename");
os << "Some data: " <<  " and more data" << std::endl;
…
os.close();                          // write file
```

more about stl-input/output: www.cppreference.com

STL: Input/Output

std::istream

```
std::istream file(„filename“);
if(file.good())
{
    char line[1000];
    while(!file.eof())
    {
        file.getline(line,1000);
        ...
    }
}
```

// if file has been opened successfully

// while not at end of file do ...

// read up to 1000 characters of line

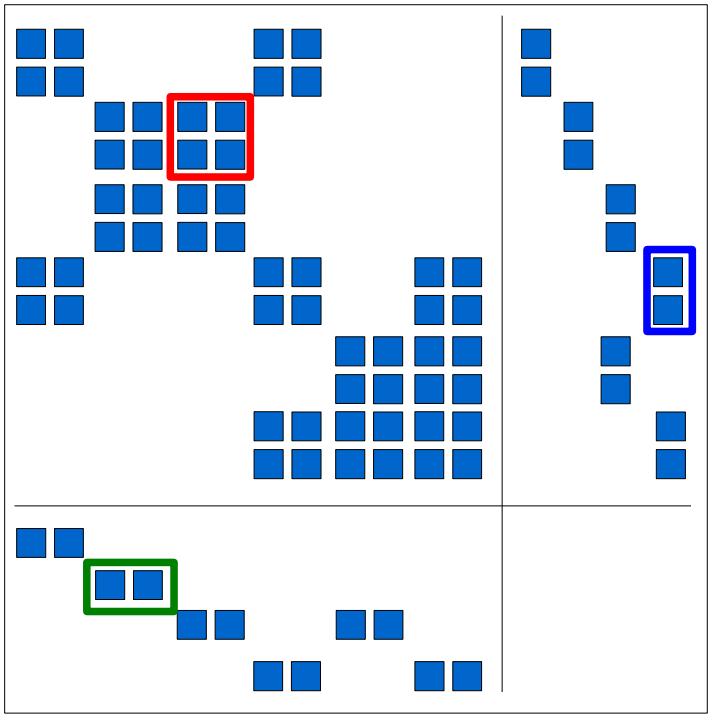
Template Metaprogramming

Why Template Metaprogramming

MATHEON

Stokes problem

$$-\Delta u + \nabla p = f$$
$$\operatorname{div} u = 0$$



sparsity structure of stiffness matrix

- heterogeneous block structure (dxd, 1xd, dx1)
- different possibilities of organizing storage
- either explicitly stored and checked metainformation or implicit assumptions throughout the code

Why Template Metaprogramming

Storing FE coefficient vectors

- u and p live on different grid entities ➡ no interleaving

  [[u],[p]]

- components of u have physical meaning in every point ➡ grouping

  [[[u11,u12],[u21,u22],[u31,u32],...],[p1,p2,p3,...]]

- structured representation in std::vector's is highly inefficient (and looses structure)

```
vector<vector<vector<double>>,vector<vector<double>>>
```

- 'flat' representation contains no structure

```
vector<double>
```

Why Template Metaprogramming

Type safety and structured data

Compilers are much better at tedious error checking than programmers.
- array access checking
- array blocking

Aggressive compiler optimizations

Compilers are much better at low level optimizations than programmers.
- loop unrolling
- inlining
- constant folding
- dead code elimination

Code transformations

Computers (e.g. compilers) are much better at repetitive tasks than programmers.
- expression templates

A Template Metaprogram


```
template <int p, int i> struct isPrime {
  static int const value = (p==2) || (p%i) && isPrime<p,i-1>::value;
};
template <int p> struct isPrime<p,1> {
  static int const value = true;
};

template <int p> struct previous {
  static int const value = isPrime<p-1,p-2>::value? p-1:
                                                    previous<p-1>::value;
};
template <> struct previous<2> {
  static int const value = 1;
};

template <int p> struct is {
  static int const value = is<previous<p>::value>::value;
};
template <> struct is<1> {};

int main() { return is<23>::value; } // what does main() return?
```

[Unruh '95]

A Template Metaprogram


```
$ g++ prime.cpp
prime.cpp: In function 'int main()':
prime.cpp:17:   instantiated from 'const int is<2>::value'
prime.cpp:17:   instantiated from 'const int is<3>::value'
prime.cpp:17:   instantiated from 'const int is<5>::value'
prime.cpp:17:   instantiated from 'const int is<7>::value'
prime.cpp:17:   instantiated from 'const int is<11>::value'
prime.cpp:17:   instantiated from 'const int is<13>::value'
prime.cpp:17:   instantiated from 'const int is<17>::value'
prime.cpp:17:   instantiated from 'const int is<19>::value'
prime.cpp:17:   instantiated from 'const int is<23>::value'
prime.cpp:21:   instantiated from here
prime.cpp:17: error: 'value' is no element of 'is<1>'
```

[GCC 4.0.2]

C++ templates form a complete Turing machine, executing (weird) code at compile time.

Template Metaprogramming for FE

MATHEON

FE coefficient vectors

Stokes: [[[u11,u12],[u21,u22],[u31,u32],...], [p1,p2,p3,...]]

Fixed-size vectors

```
template <class Scalar, int n>
class FieldVector {
  Scalar data[n];
public:
  Scalar& operator[](int i) { return data[i]; }
  void operator+=(FieldVector& v) {
    for (int i=0; i<n; ++i) data[i] += v[i]; // loop unrolling
  }
};
```

```
[ std::vector<FieldVector<double,n>>, std::vector<double> ]
```

- compiler knows and enforces that all u's have the same number of components
- coefficients of u and p are contiguous in memory
- different type of coefficient array for each variable

Heterogeneous Containers

MATHEON ZIB

```
template <class T0, class T1 /* maybe more */>
struct HeterogeneousVector {
  T0 data0;  // for u: T0 = std::vector<FieldVector<double,2>>
  T1 data1;  // for p: T1 = std::vector<FieldVector<double,1>>
  // maybe more
};

// access by index
template <class T0, class T1, int n>
struct VectorAccess {};

template <class T0, class T1>
struct VectorAccess<T0,T1,1> { // partial specialization n=1
  typedef T1 type;
  type& at(HeterogeneousVector<T1,T2>& v) { return v.data1; }
};

template <class T1, class T2, int n>
VectorAccess<T1,T2,n>::type& at(HeterogeneousVector<T1,T2>& v) {
  return VectorAccess<T1,T2,n>::at(v);
}
```

Heterogeneous Containers

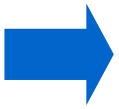

```
boost::fusion::vector<T1,T2>
```

... and a lot more!

Element access

```
using namespace boost::fusion;

vector<std::string,double> v;
at_c<0>(v) = "pi";
at_c<1>(v) = 3.1415;
```



data structures become
- general
- structure preserving
- type-safe
- efficient
- ugly

Loop Unrolling


```
double dynamic_sum(double* x,
                   int n) {
  double sum = 0;
  for (int i=0; i<n; ++i)
    sum += x[i];
  return sum;
}

…

  double data[3];
  sum = dynamic_sum(data,3);
```

```
template <int n>
double static_sum(double* x) {
  double sum = 0;
  for (int i=0; i<n; ++i)
    sum += x[i];
  return sum;
}

…

  double data[3];
  sum = static_sum<3>(data);
```

Loop Unrolling

```
.LFB508:
        pushl   %ebp
.LCFI0:
        movl    %esp, %ebp
.LCFI1:
        movl    12(%ebp), %edx
        movl    8(%ebp), %ecx
        testl   %edx, %edx
        jle     .L9
        fldz
        xorl    %eax, %eax
        .p2align 4,,7
.L5:
        faddl   (%ecx,%eax,8)
        addl    $1, %eax
        cmpl    %edx, %eax
        jne     .L5
        popl    %ebp
        ret
.L9:
        popl    %ebp
        fldz
        ret
```

```
.LFB513:
        pushl   %ebp
.LCFI2:
        movl    %esp, %ebp
.LCFI3:
        movl    8(%ebp), %eax
        fldz
        popl    %ebp
        faddl   (%eax)
        faddl   8(%eax)
        faddl   16(%eax)
        ret
```

[g++ -O3 -S]

code becomes
- more compact
- faster

Inlining

```
struct Integrand {
  virtual double f(double x)=0;
};

struct Constant: public Integrand {
  virtual double f(double x) {
    return 1; }
};

double integral(Integrand& f,
                int n) {
  double sum = 0;

  for (int i=0; i<n; ++i)
    sum += f.f((i+0.5)/n);

  return sum/n;
}

...
Constant f;
integral(f,1000000000);
```

```
struct Constant {
  double f(double x) {
    return 1; }
};

template <class Integrand>
double Integral(Integrand& f,
                int n) {
  double sum = 0;

  for (int i=0; i<n; ++i)
    sum += f.f((i+0.5)/n);

  return sum/n;
}

...
Constant f;
integral(f,1000000000);
```

Inlining


```
_ZN8ConstantclEd:
.LFB2:
        movsd   .LC0(%rip), %xmm0
        ret

main:
.LFB4:
        pushq   %rbp
.LCFI4:
        pushq   %rbx
.LCFI5:
        movl    $1, %ebx
        subq    $40, %rsp
.LCFI6:
        leaq    16(%rsp), %rbp
        movq    $_ZTV8Constant+16, 16(%rsp)
        movsd   .LC3(%rip), %xmm0
        movq    %rbp, %rdi
        call    *_ZTV8Constant+16(%rip)
        movapd  %xmm0, %xmm1
        addsd   .LC1(%rip), %xmm1
        .p2align 4,,7
.L13:
        cvtsi2sd    %ebx, %xmm0
        movq    16(%rsp), %rax
        movsd   %xmm1, (%rsp)
        movq    %rbp, %rdi
        addl    $1, %ebx
        addsd   .LC2(%rip), %xmm0
        divsd   .LC4(%rip), %xmm0
        call    *(%rax)
        movsd   (%rsp), %xmm1
        cmpl    $1000000000, %ebx
        addsd   %xmm0, %xmm1
        jne .L13
        divsd   .LC4(%rip), %xmm1
        addq    $40, %rsp
        popq    %rbx
        popq    %rbp
        cvttsd2si %xmm1, %eax
        ret
```

```
main:
.LFB4:
    movsd   .LC0(%rip), %xmm0
    movl    $1, %eax
    movapd %xmm0, %xmm1
    .p2align 4,,7
.L2:
    addl    $1, %eax
    addsd   %xmm1, %xmm0
    cmpl    $1000000000, %eax
    jne .L2
    divsd   .LC1(%rip), %xmm0
    cvttsd2si %xmm0, %eax
    ret
```

[g++ -O3 -S]

Timings

dynamic	11.85s
static	1.48s

Template Metaprogramming Impact

✔ compact code
✔ general code
✔ type safety
✔ performance

✗ long compile times
✗ strange syntax

Template Metaprogramming Impact

✔ compact code
✔ general code
✔ type safety
✔ performance

✗ long compile times
✗ strange syntax

Example: tiny matrices

```
template <class T, int rows, int cols>
class Matrix {
public:
    T const& operator(int r, int c) const;
};

template <class T, int n, int k, int m>
Matrix<T,n,m> operator*(Matrix<T,n,k> const& A,
                        Matrix<T,k,m> const& B) {
  Matrix<T,n,m> AB(0);
  for (int i=0; i<n; ++i)
    for (int j=0; j<m; ++j)
      for (int l=0; l<k; ++l
        AB(i,j) += A(i,l)*B(l,j);
  return AB;
}
```

- one piece of code works for many situations
- "debug once, run everywhere"

Template Metaprogramming Impact

✔ compact code
✔ general code
✔ type safety
✔ performance

✗ long compile times
✗ strange syntax

```
Example: tiny matrices

Matrix<double,2,3> A, B;
Matrix<double,3,3> AB = A*B;  // compile time error
Matrix <complex<double>,3,2> C;
Matrix <double,3,3> CA = C*A; // compile time error
```

- less need to debug programs during runtime
  (in fact, we rarely do)
- more need to digest cryptic compiler error messages

Rule of thumb: When the compiler accepts the code, it is semantically correct.

Template Metaprogramming Impact

MATHEON

- ✔ compact code
- ✔ general code
- ✔ type safety
- ✔ performance

- ✗ long compile times
- ✗ strange syntax

Example: tiny matrices

```
template <class T, int rows, int cols>

Matrix<double,2,2> A, B;
Matrix<double,2,2> AB = A*B;

// equivalent to

AB(0,0) = A(0,0)*B(0,0) + A(0,1)*B(1,0);
AB(0,1) = A(0,0)*B(0,1) + A(0,1)*B(1,1);
AB(1,0) = A(1,0)*B(0,0) + A(1,1)*B(1,0);
AB(1,1) = A(1,0)*B(0,1) + A(1,1)*B(1,1);
```

- inlining
- loop unrolling
- dead code elimination