Modern C++: prerequisites

For us

Basic C++98

Very basic startup material:

- http://www.cplusplus.com/doc/oldtutorial/

Something more:

- Thinking in C++, Bruce Eckel
  https://www.micc.unifi.it/bertini/download/programmazione/TICPP-2nd-ed-Vol-one-printed.pdf
- Effective C++: 55 Specific Ways to Improve Your Programs and Designs, Scott Meyers

Basic C++98

Bruce Eckel, Thinking in C++:

C++ is a language in which new and different features are built on top of an existing syntax. Because of this, it is referred to as a hybrid object-oriented programming language.

In his book he treats:

- the C in C++,
- data abstraction (the class)
- implementation hiding,
- inheritance,
- polymorphism,
- the standard template library

The C in C++

```
void test()
{
    int count = 0;
    for(int i=0; i<42; i++)
    {
        count ++;
    }
}
```

The class

It is used to represent a data structure, an object, a complex behaviour described by some variables

Example:

- I want an integer number which can assume values only inside the interval [0, max].
- I can assign a value to it and retrieve it.
- I don't care about it is implemented. I just want to use it.

The class

The visible interface of the class: file cpptest.h

```
// - include guard ------------------------------------------------------------

#ifndef _CPP_TEST_H_
#define _CPP_TEST_H_

#include <cstddef>

typedef unsigned int posv_t;

namespace test01 {

    void runit();

    class LN
    {   // Limited Number
    public:
        LN(const posv_t max, const posv_t val = 0); // ctor

        posv_t get() const;                         // getter
        void set(const posv_t val);                 // setter

    private:
        posv_t m_m;
        posv_t m_v;                                 // implementation
    };

}  // namespace test01
```

The class

The implementation of the class: file cpptest.cpp

```
// -------------------------------------------------------
// - public interface
// -------------------------------------------------------

#include "cpptest.h"
```

```
namespace test01 {

    LN::LN(const posv_t max, const posv_t val) : m_m(max), m_v(std::min(max, val)) {}

    posv_t LN::get() const
    {
        return m_v;
    }

    void LN::set(const posv_t val)
    {
        m_v = std::min(val, m_m);
    }

} // namespace test01

```

The class

Use of the class: file test.cpp

```

#include "cpptest.h"
```

```
namespace test01 {

    void run1();
    void run2();

    void runit()
    {
        run1();
        run2();
    }

    void run1()
    {
        LN ln1(99);
        ln1.set(1);
        posv_t value = ln1.get();
        value = value;
    }
```

ISTITUTO ITALIANO DI TECNOLOGIA

The class

The compiler implicitly generates some functions. They are called Special member functions.

From: https://en.cppreference.com/w/cpp/language/member_functions

Special member functions

constructors and destructors are non-static member functions that use a special syntax for their declarations (see their pages for details).

Some member functions are *special*: under certain circumstances they are defined by the compiler even if not defined by the user. They are:

- Default constructor
- Copy constructor
- Move constructor (since C++11)
- Copy assignment operator
- Move assignment operator (since C++11)
- Destructor

Special member functions along with the comparison operators (since C++20) are the only functions that can be *defaulted*, that is, defined using `= default` instead of the function body (see their pages for details)

The class

Use of the class: file test.cpp

```
    void run2()
    {
        LN ln1(99, 1);
        // the following works fine using what the compiler implicitly generates:
        // copy assignment (ca), copy constructor (cc), destructor (dtor)
        LN lnx = ln1;   // calls ca
        lnx = 7;        // calls LN(7), then ca
        lnx = LN(3, 4); // calls ctor and then ca
        posv_t vv0 = lnx.get();
        LN lny = LN(ln1);   // calls cc, then ca
        posv_t vv1 = lny.get();
        vv1 = vv1;
    }   // dtor for: ln1, lnx, lny

} // namespace test01
```

The class

Should I rely on the compiler to generate special member functions for me or should I not? Which special functions should I write myself?

The answer my friend is the **rule of three**

Rule of three

If a class requires a user-defined destructor, a user-defined copy constructor, or a user-defined copy assignment operator, it almost certainly requires all three.

Because C++ copies and copy-assigns objects of user-defined types in various situations (passing/returning by value, manipulating a container, etc), these special member functions will be called, if accessible, and if they are not user-defined, they are implicitly-defined by the compiler.

The implicitly-defined special member functions are typically incorrect if the class is managing a resource whose handle is an object of non-class type (raw pointer, POSIX file descriptor, etc), whose destructor does nothing and copy constructor/assignment operator performs a "shallow copy" (copy the value of the handle, without duplicating the underlying resource).

Source: https://en.cppreference.com/w/cpp/language/rule_of_three

The class

```
class rule_of_three
{
    char* cstring; // raw pointer used as a handle to a dynamically-allocated memory block

    void init(const char* s)
    {
        std::size_t n = std::strlen(s) + 1;
        cstring = new char[n];
        std::memcpy(cstring, s, n); // populate
    }
 public:
    rule_of_three(const char* s = "") { init(s); }

    ~rule_of_three()
    {
        delete[] cstring;  // deallocate
    }

    rule_of_three(const rule_of_three& other) // copy constructor
    {
        init(other.cstring);
    }

    rule_of_three& operator=(const rule_of_three& other) // copy assignment
    {
        if(this != &other) {
            delete[] cstring;  // deallocate
            init(other.cstring);
        }
        return *this;
    }
};
```

The class

DEMO

- Is the following class OK?
- If not, how do I make it safe?

```
namespace test02 {

    void runit();

    class BAD
    {   // BAD: Array of Limited Numbers
    public:
        BAD(const size_t capacity, const posv_t max, const posv_t val = 0); // ctor

        posv_t get(const size_t pos) const;                // getter
        void set(const size_t pos, const posv_t val);      // setter

    private:
        size_t m_c;
        posv_t m_m;
        posv_t *m_vv;                                      // implementation
    };


}  // namespace test02

```

The class

ANSWERS: **NO! YOU MUST ADD DTOR, COPY CTOR, COPY ASSIGNMENT!**

```
namespace test03 {

    void runit();

    class ALN
    {   // Array of Limited Numbers
    public:
        ALN(const size_t capacity, const posv_t max, const posv_t val = 0); // ctor

        posv_t get(const size_t pos) const;                 // getter
        void set(const size_t pos, const posv_t val);       // setter

        ALN(const ALN &other);                              // copy ctor
        ALN& operator=(const ALN &other);                   // copy assignment
        ~ALN();                                             // dtor

    private:
        size_t m_c;
        posv_t m_m;
        posv_t *m_vv;                                       // implementation
    };

}  // namespace test03
```

The class

```
namespace test06 {

    void runit();

    class LPAIR
    {
    public:
        LPAIR(const posv_t max, const posv_t val = 0);

        void get(posv_t &v0, posv_t &v1) const;
        void set(const posv_t v0, const posv_t v1);

    private:
        test01::LN ln0;
        test01::LN ln1;
    };

}  // namespace test06
```

ISTITUTO ITALIANO DI TECNOLOGIA

The class

```
namespace test06 {

    void runit()
    {
        LPAIR lp(99, 2);
        lp.set(3, 4);
        posv_t v0 = 0;
        posv_t v1 = 1;
        lp.get(v0, v1);
    }


    LPAIR::LPAIR(const posv_t max, const posv_t val) : ln0(max, val), ln1(max, val) {}

    void LPAIR::get(posv_t &v0, posv_t &v1) const
    {
        v0 = ln0.get();
        v1 = ln1.get();
    }

    void LPAIR::set(const posv_t v0, const posv_t v1)
    {
        ln0.set(v0);
        ln1.set(v1);
    }

} // namespace test06
```

ISTITUTO ITALIANO DI TECNOLOGIA

Inheritance

```cpp
namespace test07 {

    void runit();

    class Shape
    {
    public:
        Shape(const test06::LPAIR &position, size_t area);
        size_t getarea() const;
        void moveto(const test06::LPAIR &position);
        virtual void draw();
    protected:
        test06::LPAIR _pos;
        size_t _area;
    };
```

C++98: base class

Inheritance

```cpp
Shape::Shape(const test06::LPAIR &position, size_t area) : _pos(position), _area(area) {}

size_t Shape::getarea() const
{
    return _area;
}

void Shape::moveto(const test06::LPAIR &position)
{
    _pos = position;
}

void Shape::draw()
{
    static size_t as_a_shape = 0;
    as_a_shape++;
}
```

```cpp
namespace test07 {

    void runit()
    {
        Shape sh(test06::LPAIR(100, 2, 2), 25);
        size_t a = sh.getarea();
        sh.draw();
```

Inheritance

```cpp
    class Square : public Shape
    {
    public:
        Square(const test06::LPAIR &position, const size_t len);
        void resize(size_t len);
        virtual void draw();
    private:
        size_t _len;
    };

} // namespace test07
```

C++98: derived class

Inheritance

```cpp
Square::Square(const test06::LPAIR &position, const size_t len) : Shape(position, len*len), _len(len) {}

void Square::resize(size_t len)
{
    _len = len;
    // must now operate on protected members of Shape
    _area = len*len;
}

void Square::draw()
{
    static size_t as_a_square = 0;
    as_a_square++;
}
```

```cpp
Square sq(test06::LPAIR(100, 2, 2), 3);
size_t area = sq.getarea();
sq.resize(4);
area = sq.getarea();
area =  area;
sq.draw();
```

C++98: derived class

Polymorphism

```
Shape *polym = new Square(test06::LPAIR(100, 2, 2), 3);
polym->moveto(test06::LPAIR(100, 9, 9));
polym->draw();
```

The pointer polym behaves like a Shape but also like a Square

Implementation hiding

- By means of private or protected members (variables or functions)
- By means of interfaces
- By means of PIMPL idiom (pointer to implementation)

Implementation hiding

Scott Meyers, Effective C++: … interface class *specifies an interface for derived classes (see Item 34). As a result, it typically has no data members, no constructors,* ***a virtual destructor*** *(see Item 7), and a set of pure virtual functions that specify the interface.*

```cpp
namespace test04 {

    void runit();

    class Person
    {   // no ctor, no data. only: a virtual dtor + pure virtual functions
    public:
        virtual std::string getname() const = 0;
        virtual int getsomething() const = 0;
        virtual void setsomething(const int &v) = 0;
        virtual ~Person() {};
    };

    Person * generate(const std::string& name);

}  // namespace test04
```

Implementation hiding

```
namespace test04 {

    void runit()
    {
        Person *p = generate("Scott Meyers");
        p->setsomething(3);
        int smt = p->getsomething();
        smt = smt;

        delete p;
    }
```

Implementation hiding

… at some point, of course, concrete classes supporting the Interface class's interface must be defined and real constructors must be called. That all happens behind the scenes inside the files containing the implementations of the virtual constructors. For example, the Interface class Person might have a concrete derived class RealPerson that provides implementations for the virtual functions it inherits.

```cpp
class RealPerson: public Person
{
public:
    RealPerson(const std::string& name) : _n(name), _e(0) {}
    virtual ~RealPerson() {}
    std::string getname() const { return _n; }
    int getsomething() const { return _e; }
    void setsomething(const int &v) { _e = v; }
private:
    std::string _n;
    int _e;
};

Person * generate(const std::string& name)
{
    return new RealPerson(name);
}
```

Implementation Hiding

Header file:

```
class Parser {
public:
    Parser(const char *params);
    ~Parser();

    void parse(const char *input);

    // Other methods related to responsibility of the class
    ...

private:
    Parser(const Parser&);                // noncopyable
    Parser& operator=(const Parser&);     //

    class Impl;     // Forward declaration of the implementation class
    Impl *impl_;    // PIMPL
};
```

Implementation Hiding

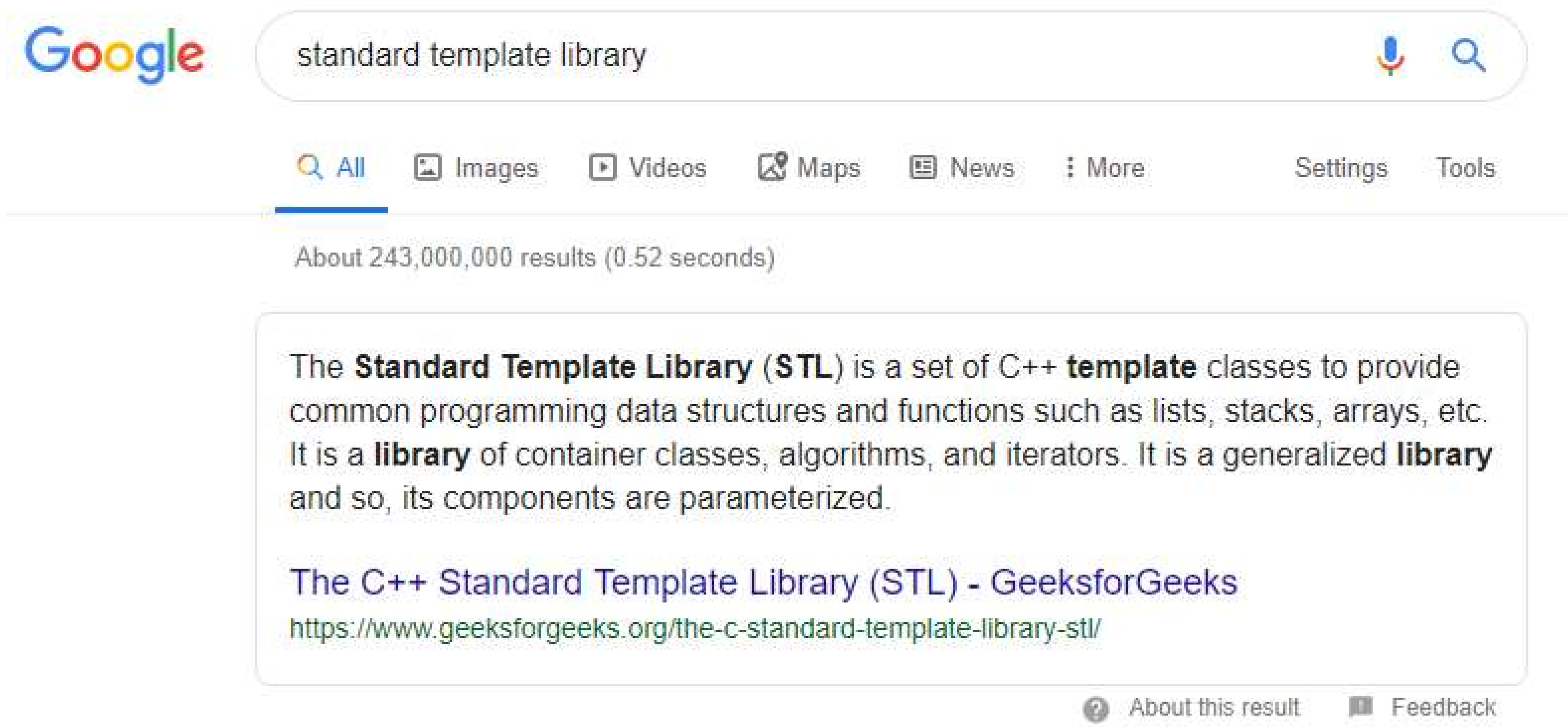
Source file:

```
// Include all headers the implementation requires

// The actual implementation definition:
class Parser::Impl {
public:
        Impl(const char *params) {
                // Actual initialization
                ...
        }

    void parse(const char *input) {
        // Actual work
        ...
    }
};
```

Implementation Hiding

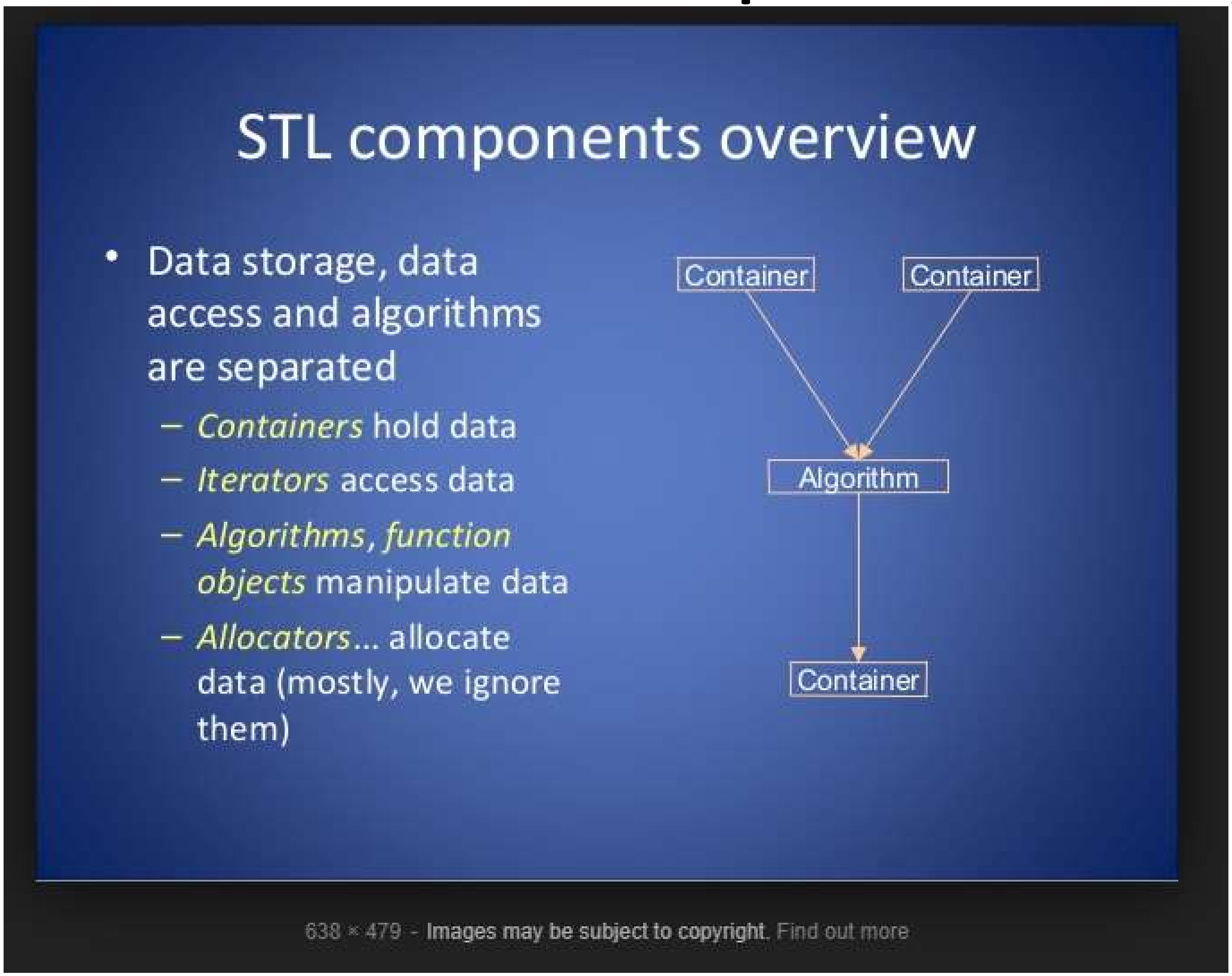
```
// Create an implementation object in ctor
Parser::Parser(const char *params)
: impl_(new Impl(params))
{}

// Delete the implementation in dtor
Parser::~Parser() { delete impl_; }

// Forward an operation to the implementation
void Parser::parse(const char *input) {
    impl_->parse(input);
}

// Forward other operations to the implementation
...
```

The Standard Template Library



ISTITUTO ITALIANO DI TECNOLOGIA

The Standard Template Library



The Standard Template Library

```cpp
#include <vector>
#include <algorithm>

namespace test08 {

    void runit()
    {
        std::vector<int> vv;
        vv.push_back(3);
        vv.push_back(1);
        vv.push_back(5);
        // now the vector contains three values: {3, 1, 5}
        // ok, now i want to sort the vector.
        std::sort(vv.begin(), vv.end());
        // ok, now the order is {1, 3, 5}

        // is there the number 3 inside?
        std::vector<int>::iterator r = std::find(vv.begin(), vv.end(), 3);
        bool found = false;
        if(r != vv.end())
        {
            found = true;
        }
    }

} // namespace test08
```