



# 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++


```
414  
415 void test()  
416 {  
417     int count = 0;  
418     for(int i=0; i<42; i++)  
419     {  
420         count ++;  
421     }  
422 }
```

# 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  $[0, \text{max}]$ . 
- I can assign a value to it and retrieve it.
- I don't care about how it is implemented. I just want to use it.


# The class




The visible interface of the class: file cpptest.h

```
19
20 // - include guard -----
21
22 #ifndef _CPP_TEST_H_
23 #define _CPP_TEST_H_
24
25 #include <cstdint>
26
27 typedef unsigned int posv_t;
28
29 namespace test01 {
30
31     void runit();
32
33     class LN
34     { // Limited Number
35     public:
36         LN(const posv_t max, const posv_t val = 0); // ctor
37
38         posv_t get() const; // getter
39         void set(const posv_t val); // setter
40
41     private:
42         posv_t m_m;
43         posv_t m_v; // implementation
44     };
45
46 } // namespace test01
```

# The class

The implementation of the class: file cpptest.cpp


```
20 // -----  
21 // - public interface  
22 // -----  
23  
24 #include "cpptest.h" 
```

```
78 namespace test01 {   
79   
80     LN::LN(const posv_t max, const posv_t val) : m_m(max), m_v(std::min(max, val)) {}  
81  
82     posv_t LN::get() const  
83     {  
84         return m_v;   
85     }  
86  
87     void LN::set(const posv_t val)  
88     {  
89         m_v = std::min(val, m_m);  
90     }  
91  
92 } // namespace test01  
93
```

# The class



Use of the class: file test.cpp

```
23
24  #include "cpptest.h" 
43  namespace test01 {
44
45      void run1();
46      void run2();
47
48      void runit()
49      {
50          run1();
51          run2();
52      }
53
54      void run1()
55      {
56          LN ln1(99);
57          ln1.set(1);
58          posv_t value = ln1.get();
59          value = value;
60      }
61
```



# The class

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

From: [https://en.cppreference.com/w/cpp/language/member\\_functions](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


```
62 void run2()
63 {
64     LN ln1(99, 1);
65     // the following works fine using what the compiler implicitly generates:
66     // copy assignment (ca), copy constructor (cc), destructor (dtor)
67     LN lnx = ln1;
68     lnx = 7;
69     lnx = LN(3, 4);
70     posv_t vv0 = lnx.get();
71     LN lny = LN(ln1);
72     posv_t vv1 = lny.get();
73     vv1 = vv0;
74 } // dtor for: ln1, lnx, lny
75
76 } // 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](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

INFO: the ctor of *BAD* will allocate on the heap an array of *capacity* elements.

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

```

49 namespace test02 {
50
51     void runit();
52
53     class BAD
54     { // BAD: Array of Limited Numbers
55     public:
56         BAD(const size_t capacity, const posv_t max, const posv_t val = 0); // ctor
57
58         posv_t get(const size_t pos) const; // getter
59         void set(const size_t pos, const posv_t val); // setter
60
61     private:
62         size_t m_c;
63         posv_t m_m;
64         posv_t *m_vv; // implementation
65     };
66
67
68 } // namespace test02
69


```

# The class

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

```
70 namespace test03 {
71
72     void runit();
73
74     class ALN
75     { // Array of Limited Numbers
76     public:
77         ALN(const size_t capacity, const posv_t max, const posv_t val = 0); // ctor
78
79         posv_t get(const size_t pos) const; // getter
80         void set(const size_t pos, const posv_t val); // setter
81
82         ALN(const ALN &other); // copy ctor
83         ALN& operator=(const ALN &other); // copy assignment
84         ~ALN(); // dtor
85
86     private:
87         size_t m_c;
88         posv_t m_m;
89         posv_t *m_vv; // implementation
90     };
91
92 } // namespace test03
```

# The class

```
136 namespace test06 {  
137  
138     void runit();  
139  
140     class LPAIR   
141     {  
142     public:  
143         LPAIR(const posv_t max, const posv_t v0 = 0, const posv_t v1 = 0);  
144  
145         void get(posv_t &v0, posv_t &v1) const;  
146         void set(const posv_t v0, const posv_t v1);  
147  
148     private:  
149         test01::LN ln0;  
150         test01::LN ln1;  
151     };  
152  
153 } // namespace test06  
154
```



# The class

```
300 namespace test06 {
301
302     void runit()
303     {
304         LPAIR lp(99, 2, 3);
305         lp.set(3, 4);
306         posv_t v0 = 0;
307         posv_t v1 = 1;
308         lp.get(v0, v1);
309     }
310
311
312     LPAIR::LPAIR(const posv_t max, const posv_t val) : ln0(max, val), ln1(max, val) {}
313
314     void LPAIR::get(posv_t &v0, posv_t &v1) const
315     {
316         v0 = ln0.get();
317         v1 = ln1.get();
318     }
319
320     void LPAIR::set(const posv_t v0, const posv_t v1)
321     {
322         ln0.set(v0);
323         ln1.set(v1);
324     }
325
326 } // namespace test06
327
```



# Inheritance




```
156 namespace test07 {  
157  
158     void runit();  
159  
160     class Shape  
161     {  
162     public:  
163         Shape(const test06::LPAIR &position, size_t area);  
164         size_t getarea() const;  
165         void moveto(const test06::LPAIR &position);  
166         virtual void draw();  
167     protected:  
168         test06::LPAIR _pos;  
169         size_t _area;  
170     };  
171
```

# Inheritance

```
347 Shape::Shape(const test06::LPAIR &position, size_t area) : _pos(position), _area(area) {}
348
349 size_t Shape::getarea() const
350 {
351     return _area;
352 }
353
354 void Shape::moveto(const test06::LPAIR &position)
355 {
356     _pos = position;
357 }
358
359 void Shape::draw()
360 {
361     static size_t as_a_shape = 0;
362     as_a_shape++; std::cout << "Shape::draw() is called" << std::endl;
363 }
```

```
329 namespace test07 {
330
331     void runit()
332     {
333         Shape sh(test06::LPAIR(100, 2, 2), 25);
334         size_t a = sh.getarea();
335         sh.draw();
336     }
```

# Inheritance

```
172  
173  class Square : public Shape  
174 {  
175     public:  
176         Square(const test06::LPAIR &position, const size_t len);  
177         void resize(size_t len);  
178         virtual void draw();  
179     private:  
180         size_t _len;  
181 };  
182  
183 } // namespace test07  
184
```

# Inheritance

```

365 Square::Square(const test06::LPAIR &position, const size_t len) : Shape(position, len*len), _len(len) {}
366
367 void Square::resize(size_t len)
368 {
369     _len = len;
370     // must now operate on protected members of Shape
371     _area = len*len;
372 }
373
374 void Square::draw()
375 {
376     static size_t as_a_square = 0;
377     as_a_square++; std::cout << "Square::draw() is called" << std::endl;
378 }

```

```

338 Square sq(test06::LPAIR(100, 2, 2), 3);
339 size_t area = sq.getarea();
340 sq.resize(4);
341 area = sq.getarea();
342 area = area;
343 sq.draw();

```

# Polymorphism



```
345 Shape *polym = new Square(test06::LPAIR(100, 2, 2), 3);  
346 polym->moveto(test06::LPAIR(100, 9, 9));  
347 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.

```
97 namespace test04 {
98
99     void runit();
100
101     class Person
102     { // no ctor, no data. only: a virtual dtor + pure virtual functions
103     public:
104         virtual std::string getname() const = 0;
105         virtual int getsomething() const = 0;
106         virtual void setsomething(const int &v) = 0;
107         virtual ~Person() {}
108     };
109
110     Person * generate(const std::string& name);
111
112 } // namespace test04
113
```

# Implementation hiding



```
228 namespace test04 {  
229  
230     void runit()  
231     {  
232         Person *p = generate("Scott Meyers");  
233         p->setsomething(3);  
234         int smt = p->getsomething();  
235         smt = smt;  
236  
237         delete p;  
238     }  
239 }
```



# 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.

```
241     class RealPerson: public Person
242     {
243     public:
244         RealPerson(const std::string& name) : _n(name), _e(0) {}
245         virtual ~RealPerson() {}
246         std::string getname() const { return _n; }
247         int getsomething() const { return _e; }
248         void setsomething(const int &v) { _e = v; }
249     private:
250         std::string _n;
251         int _e;
252     };
253
254     Person * generate(const std::string& name)
255     {
256         return new RealPerson(name);
257     }
```

# 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

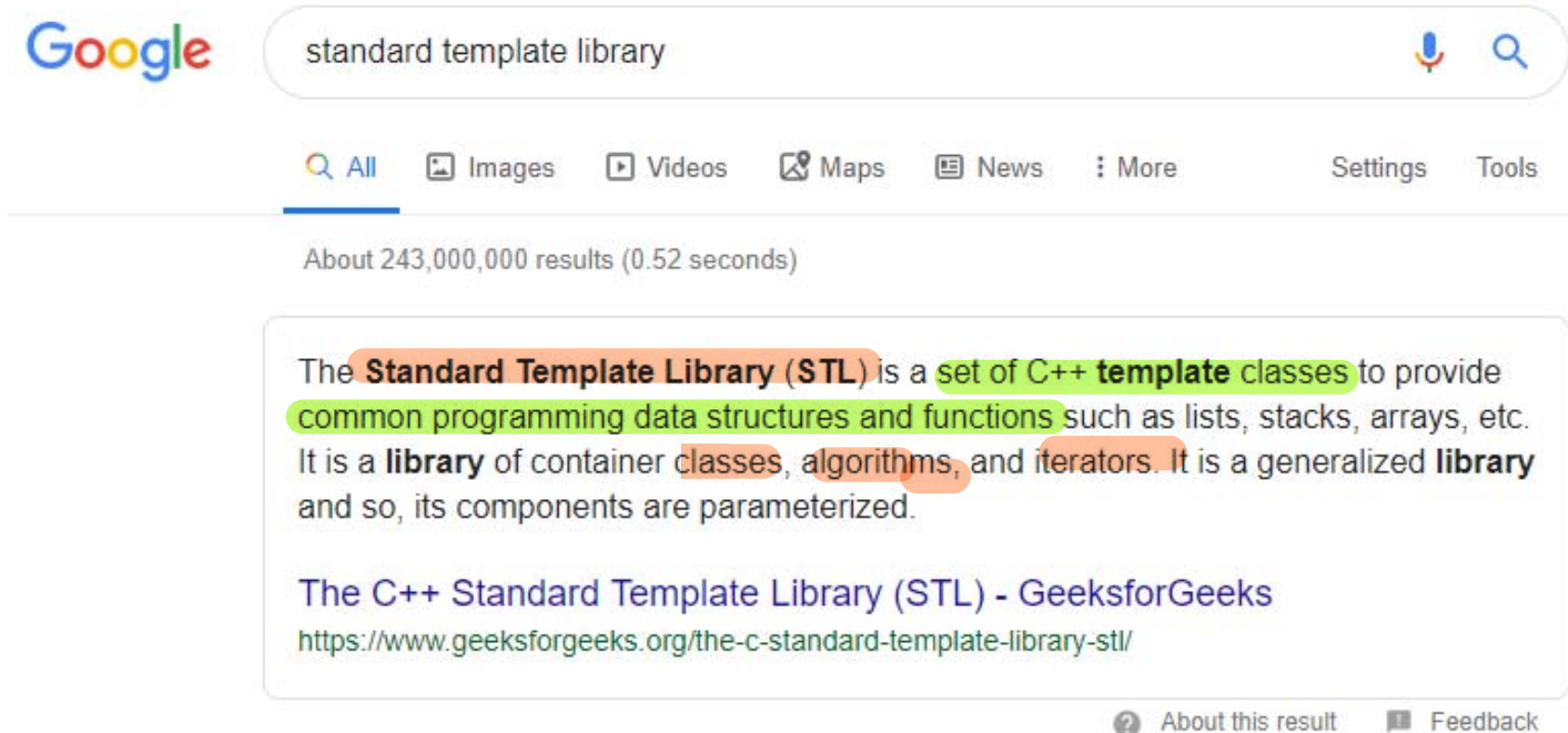
```
// Create an implementation object in ctor
Parser::Parser(const char *params)
: impl_(new Impl(params))
{}

// 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
...
```

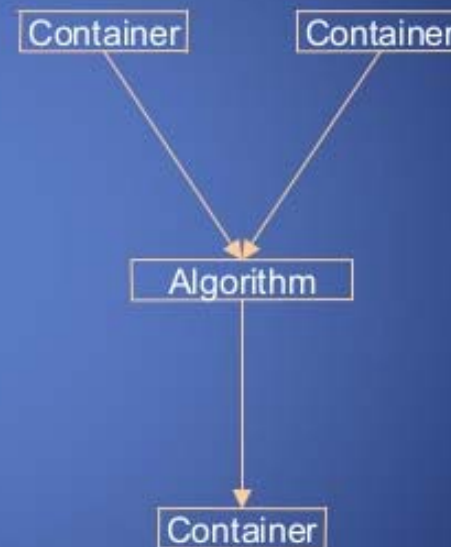
# The Standard Template Library



# The Standard Template Library

## STL components overview

- **Data storage, data access and algorithms** are separated
  - *Containers* hold data
  - *Iterators* access data
  - *Algorithms, function objects* manipulate data
  - *Allocators*... allocate data (mostly, we ignore them) 



638 × 479 - Images may be subject to copyright. Find out more

# The Standard Template Library

```
388  #include <vector>
389  #include <algorithm>
390
391  namespace test08 {
392
393      void runit()
394      {
395          std::vector<int> vv;
396          vv.push_back(3);
397          vv.push_back(1);
398          vv.push_back(5);
399          // now the vector contains three values: {3, 1, 5}
400          // ok, now i want to sort the vector.
401          std::sort(vv.begin(), vv.end());
402          // ok, now the order is {1, 3, 5}
403
404          // is there the number 3 inside?
405          std::vector<int>::iterator r = std::find(vv.begin(), vv.end(), 3);
406          bool found = false;
407          if(r != vv.end())
408          {
409              found = true;
410          }
411      }
412
413  } // namespace test08
414
```

