Programmazione Avanzata per il Calcolo Scientifico
Advanced Programming for Scientific Computing
Lecture title: C++11 Overview

Luca Formaggia

MOX
Dipartimento di Matematica "F. Brioschi"
Politecnico di Milano

A.A. 2012/2013

C++11 feels like a new language

B. Stroustup.

- Things to make programming easier: **auto**, decltype, alias template, range based for loops; initializer lists and uniform initialization, delegating and inheriting constructors, right angle brackets, partial spec. template functions...
- Things to make programming safer: Explicit overrides and final; default/deleted constructor/assignment/dctor/, null pointer constant; fixed integer types; new string literals; static assertions, smart pointers; fixed size enums; class enums; explicit conversion operators.
- Things to improve efficiency: constexpr, Move semantic, perfect forwarding, std::array<>,External template explicit instance.
- Things to ease writing functions: lambda functions, binders, function wrappers; reference wrappers.

- Better and new containers. Emplace operations; Unordered containers; array<>, cbegin(), cend(), begin() and end() as free function.
- New functionalites: Tuples; Random numbers and distributions.
- Things to help parsing and string operations. New std::string operators; Regular expressions!.
- Support for generic and metaprogramming. Variadic templates, Type traits; New function syntax;
- Support for multithreading. Full in-build multithreading facilities.
- New exception handling.

automatic type deduction

```
std::vector<int> a;
auto it= a.begin();
double a;
int b;
std::decltype(a+b) c=a+b; // c is double
```

auto matches the type without qualifiers: use **auto** & if you want a reference. Also **const auto** is allowed.

decltype(expr) returns the type of an expression. There are other ways to deduce the common type of two types (in <type_traits>).

alias templates

```
template <typename T>
using Dict = std::map<string,T>
...
Dict<double> a;// defines a map<string,double>

// You do not need templates:
using Real= double; // Alternative to typedef.
```

Very useful if you have templates with many arguments but you normally fix most of them.

range based for loops

If you have to transverse a whole container you can use a simplified syntax:

```
vector<double> myVect;
...
for (auto & i : myVect) i*=2.0;
...
for (auto i : myVect) std::cout<<i<<" ";
```

Note that `i` is the value (or a reference) of a contained element, not an iterator.

It works on any std container as well as user defined containers that provide begin(), end(), either as method or as free functions, returning a well behaved iterator.

uniform initialization

```
vector<int> a{3,4,5};// a contains 3,4 and 5
vector<double> b={3.4,5.6};// alternative form
struct Foo{ int a; double b};
Foo foo={5,6.0};
Foo fooArray[]={{5,6.0},{7,9.0}};
Foo returnFoo(){return {0,3.0};}
array<array<int,3>,2> ia{{1,2,3},{4,5,6}};
```

It works on vector<>, array<>, strings, complex<T>,.. on all aggregate types, as well as on more complex types it you provide a constructor taking a std:: initializer_list in input.

Assignment form does not work if you have explicit constructors.

initializer lists

```
class ComplexClass{
public:
    Complex(std::initializer_list<double> l):myV_(l){}
    ...
private:
vector<double> myV_;
};
...
ComplexClass a={4.5,6.7,7.8};
```

delegating and inheriting ctors

We can use other declared constructor in the initialization list of a constructor (finally!!!):

```
class MyClass{
 public:
   MyClass(){...};
   MyClass(double a, int b):MyClass(){....};
...
}
class Base{
public:
 Base(int);
...};
class Derived:public Base{
using Base::Base;
};
...
Derived a(10);// calls Base(10);
```

interpretation of double right angle brackets

At last >> in template definitions is interpreted as one would expect:

```
vector<vector<double>> a; // now it works!
```

If you want >> to be interpreted as the bit shift operator you need to use brackets:

```
template <int T> myClass<T>;
const int a=0;
const int b=1;
myClass< a (>>) b> s;
```

default member initialization

Default initializers for non static members are now allowed:

```
std::string defaultStringValue();
class Foo{
private:
  string s= defaultStringValue();
  int b= 10;
// you may also use unif. init. syntax:
  const double c{3.0};
}
```

Note: constructor initializers override defaults

explicit overrides finale delete and default

```
class Foo final { // No one can derive
public:
  Foo()=default;// use synthetic constructor
  Foo &operator(const Foo &)=default;// the same for =
  Foo(Foo const &)=delete; // no copy ctor
}
struct Base{
 virtual double f(int);
 virtual double g(double);
 double g(int)=delete;// no double->int conv.
};
struct Derived:public Base{
// f overrides Base::f
virtual double f(int) override;
 // no more g available to children
virtual double g(double) final;
};
```

null pointer constant

In C++98 we have basically two ways of defining a null pointer, either using the macro NULL (C style) or the number 0 (preferred syntax in C++98).

Both usage is error prone (NULL in fact is equal to 0!) since 0 is obviously convertible to **int**. C++11 has introduced a new keyword:nullptr for this purpose, which is of type std::nullptr_t and is not convertible.

```
double * p=nullprt;
```

std::nullptr_t is convertible to **bool** so we may test it in a conditional as before.

fixed width integer types

```
#include <cstdint>
int16_t i=10; // Exactly 16 bytes
uint_fast16_t b;// At least 16 bytes unsigned
intptr_t pi; // A integer that can store a pointer
intmax_t big;// The max with integer available
int64_t z(INT64_MAX);// initialized with max value;
```

Very useful when you need integers with specific width. The fast version may choose a larger width if it is more efficient in the given architecture.

New sting literals

We omit to describe the support of different unicodes, we mention just the raw string literal, very useful when parsing XML codes or when using regexp. It is treated verbatim

```
std::string s(R"(This␣string␣\␣is␣verbatim")";
std::string␣z(R"$Use a different delimiter ()$";
std::string␣k(R"(line
 feed)";
```

The first string is `This string \ is verbatim"`, the second is `Use a different delimiter ()` (so we can have parenthesis in the string). The third is `line \n feed`.

The general form is

```
R"delimiter string delimiter"
```

static assetion

A new kind of assertion has been added, that is evaluated at compile time (assert() does not guaranteed it!). Very useful together with type traits:

```
#include<type_trait>
template <class T>
void copy_swap( T& a, T& b)
{
 std::static_assert(std::is_copy_constructible<T>::value,
         "copy_swap␣requires␣copying");
 std::static_assert(std::is_copy_assignable<T>::value,
         "copy_swap␣requires␣copy-assignment");
  auto c = b;
  b = a;
  a = c;
 }
```

This is just an example. A good swap utility takes advantage of move semantic!!.

smart pointers

Boost pointers have been ported into the language and bettered!.
unique_ptr implements unique ownership:

```
#include<memory>
std::unique_ptr<Polygon> p{new Polygon};
// Move semantic! Simple assignment is forbidden
std::unique_ptr<Polygon> z=std::move(p);
// Now p is the null pointer!
// Can be used with containers
vector<std::unique_ptr<double>> vp;
// Can take arrays
std::unique_ptr<double []>(new double[10]);
```

We have comparison operators, methods to reset a pointer and release the resource, a null smart pointer converts to a nullptr_t of value nullptr.
Note: auto_ptr<> is dead! (and we are not going to miss it)

Smart Pointers

We also have std::shared_ptr<> and std:weak_ptr<>. Shared pointers share the resource they point to. Weak pointer provide a not owning view to a shared pointer. We omit the technical details, but they work similarly to the analogous Boost shared pointers.

A question: do we really need them so much? Remember that shared_ptrs have a computational and memory overhead. If you need unique ownership use unique_ptr<>, if you just need a not owning view, use an ordinary pointer (or a reference). B. Stroustrup affirms that shared_ptr<> should be used only in very particular cases. I think there is an abuse of them in LifeV.

Strongly typed enumerators

We can now specify the width of the integer forming an enum

```
enum Enum3 : unsigned long {Val1 = 1, Val2};
```

but, more interesting, we have class type enumerators:

```
enum class Bc  {Dirichet, NEumann};
Bc myBc = Bc::Dirichlet;
...
if (myBc==Bc::Dirichlet){
```

A class enumerator is not convertible to integral types and can be addressed only using the full qualified name. It makes the use of enumerators much safer.

partial specialization of function templates

Finally, you can partially specialize also function templates

```
template <class T>
foo(T const &); // generic version

template <class T>
foo(T const * &); // version for pointers

template<>
foo(double); // version for double
```

constexpr

A constant expression is an expression that may be computed at compile time. C++11 allows to specify that a function returns such type of expressions using the keyword `constexpr`:

```
constexpr double cube(const double x)
{ return x*x*x;}
```

By doing that the compiler may evaluate at compile time the expression

```
a=cube(3.0); // replaced with 9.0
```

constexpr

The new keyword `constexpr` may help develop faster code and ease some *metaprogramming techniques*. But has STRONG limitations. In particular, a function returning a `constexpr` value has to comply with the following restrictions

- It must consist of single return statement (with a few exceptions)
- It can call only other constexpr functions
- It can reference only constexpr global variables

Reference semantic in std containers

Before C++11 std containers could hold or first class objects or (normal) pointers. The old smart pointer `auto_ptr` (now deprecated) could not be stored in a container.
The new smart pointer class can instead be stored in a container, but not references:

```
vector<unique_ptr<AbstractPolygon>> a; //OK
vector<AbstractPolygon &> n; //ERROR!
```

The reason is that references are not default constructable;

Reference semantic in std containers

C++11 has introduced a way to store references in a container. This can be useful, as an alternative to pointers. You need to use std::reference_wrapper defined in the header <functional>

```
Point p1(3,4);
Point p2(5,6);
std::vector<std::reference_wrapper<Point>> v;
v.push_back(p1);// it stores a reference
v.push_back(p2);//
p1.setCoord(7,8);// change coordinates of p1
```

Also v[0] has changed coordinates since it is a reference!.

Move semantic: an introduction

One of the problems of C++11 is that often objects can be of big size. Thus, we should avoid to make useless copies and temporaries.

Unfortunately, copies may happen in different places. Let's for instance look at this piece of software that swaps two matrices.

```
Matrix a,b;
... //some work with the matrices
Matrix temp(a);
b=a;
a=temp;
```

This is inefficient, we do not really need the temporary temp, we just want to swap the state of two Matrix objects!

Rvalues in C++

User defined types and operator overloading makes the definition of rvalues/lvalues rather complicated in C++. We avoid the formal definition contained in the standard (very technical) and we recall the one in thbecker.net:

An lvalue is an expression that refers to a memory location and allows us to take the address of that memory location via the & operator. An rvalue is an expression that is not an lvalue.

Rvalue reference type

C++11 defines a new type, called rvalue reference, indicated with the double ampersand &&. Suppose we have defined

```
// A function that returns a matrix
MyMat0 foo(){... return m;}
// Move assignment
MyMat0 & MyMat0::operator =(MyMat0 && rhs){
 delete[] this->data;
 this->data=rhs.data;// grab the resource
 // Make sure than when rhs exits form its scope
 // we are fine!
 rhs.data=nullprt;
 ...}
```

The new C++11 rules state that in the statement a=foo(); this new version of assignment, called move assignment is called. We avoid the temporary!

Rvalue references

An rvalue reference X&& behaves much like the ordinary reference X&, with several exceptions. The most important one is that when it comes to function overload resolution, lvalues prefer old-style references, whereas rvalues prefer the new rvalue references

```
void foo(X& x); // lvalue reference overload
void foo(X&& x); // rvalue reference overload
X foobar();
...
X x;
foo(x); //argument is lvalue: calls foo(X&)
foo(foobar()); //argument is rvalue: calls foo(X&&)
```

Note: T&& is different than T& &, the latter is a reference to a reference, which is allowed in C++11 and reverts to T&.

The general overloading rules

If you implement **void** foo(X&); but not **void** foo(X&&); the behavior is the usual C++98 one: `foo` can be called on lvalues, but not on rvalues.

If you implement **void** foo(X **const** &); but not **void** foo(X&&); then again, the behavior is unchanged: `foo` can be called on lvalues and rvalues, but it is not possible to distinguish between them. That is possible only by implementing **void** foo(X&&); as well.

Finally, if you implement **void** foo(X&&); but neither one of **void** foo(X&); and **void** foo(X **const** &); then `foo` can be called on rvalues, but trying to call it on an lvalue will trigger a compile error.

Forcing a move

The First Amendment to the C++ Standard states: The committee shall make no rule that prevents C++ programmers from shooting themselves in the foot. So there is the way to force move semantic (of course if it has been implemented) on lvalues by transforming them into rvalues using std::move(). This can indeed be very useful:

```
template<class T>
void swap(T& a, T& b) {
  T tmp(std::move(a));
  a = std::move(b);
  b = std::move(tmp);}
```

It type T implements move assignment and move constructors the swap can be made without creating temporaries!
Beware that a=std::move(b) leaves b "empty".

Synthetic move constructors and move assignment

A C++11 compiler provides synthetic move constructors and move assignment operators for a user defined class MyClass, with signature

```
MyClass(MyClass &&);
MyClass & operator =(MyClass&& rhs);
```

They perform operations analogous to those of their copying counterparts, but they move the members of the object passed as argument (using their move constructor/assignment ops).

Note: defining a destructor or a copy constructor explicitly stops the automatic generation of move constructors and assignment. You have to define your own (if you wish) or declare them using the **default** keyword.

Perfect forwarding

In C++11 we can use the std::forward<T>() function to solve the problem of forwarding

```
template<typename Arg>
unique_ptr<Base> factory(Arg&& arg, int switch){
  if(switch==1)
  return unique_ptr<Base>(
          new D1(std::forward<Arg>(arg)));
  ... etc}
```

Thanks to the conversion rule adopted for rvalues (we omit the details for simplicity) and the magic of std::forward<T>() now this version works both if Arg is an lvalue or an rvalue and it resolves both cases correctly. As a consequence, if move semantic has been implemented on Arg no useless temporaries will be created only when it is safe to do it, all of it automagically.

Fixed size array

C++11 provides a lightweight wrapper around fixed size array with a semantic very similar to vector<> (a part from the methods related to dynamic memory management that are, of course, missing)

```
#include <array>
std::array<double,3> anArray={1.0,2.0,3.0};
std::array<Matrix,5> fiveMatrices;
...
for (auto i: anArray) f(i);
```

External template explicit instantiation (C++11 only)

In C++11 we can tell the compiler that an explicit template istantiation is provided by another compilation unit, using the keyword `extern`.
In this way we can speedup the compilation of template classes and functions if we know beforehand that they will be mostly used for certain value of the template argument. Let's see the example in Templates/ExplicitInstantiation

The file mytemp.hpp

```
template <typename T>
class Myclass{
public:
  Myclass(T const &i ):my_data( i ){}
  double fun( );
...};

template
<typename T> double func(T const &a ){...}

// Extern explicit template instantiations
extern template class Myclass<double>;
extern template class Myclass<int >;
extern template double func<double>(double const &);
```

Now source files which include `mytemp.hpp` will not create the machine code for the templates declared `extern`, leaving the corresponding symbols undefined.

The file `myfile.cpp`

```
#include "mytemp.hpp"
template class Myclass<double>; // explicit instantiantion
template class Myclass<int>; // explicit instantiation
template double func(double const &); // explicit instantiation
```

When compiling this file the template functions and classes indicated are explicitly instantiated: the machine code is generated. Usually the corresponding object file is then put in a library.

The main file

```
int main(){
  Myclass<double> a(5.0);
  Myclass<int> b(5);
  Myclass<char> c('A');
  double d=func(5.0);// call on a double
  double e=func(5);// call on a int
}
```

Using `nm -demangling main.o` we can see that only the code for the constructor and destructor of Myclass<**char**> and that for func<**int**>() has been generated! The other template instances are left undefined and will be resolved by the linker (of course we need now to link the library that contains them!).

Emplacing

All containers (a part array) have now the emplace version of insert operations, which builds elements calling the constructor directly:

```
struct Foo{
 Foo(int, double);
}
vector<Foo> a;
// calls Foo(5,6.0) to create element
a.emplace_back(5,6.0);
```

New utilities on vectors

Containers API has been improved. We indicate some useful things about vectors:

```
vector<double> a;
...
// no need of using tricks to
// adjust vector capacity anymore.
a.resize(10);
a.shink_to_fit();
..
// No more &a[0]:
double * aData=a.data();
```

Note: the data() method is present also in array<>, so it is easy to pass array<>s to functions requiring a pointer in input.

Lambda calculus (C++11 only)

The new standard has introduced a very powerful syntax to create short (and inlined) function quickly: the lambda calculus. With lambda function it is normally indicated an unnamed function. C++ lambda do indeed create expressions that may be passed as arguments to other functions, like pointers of function objects. But first look at a simple usage

```
auto f= [](double x){return 3*x;};// f is a lambda function
...
auto y=f(9.0); // y is equal to 27.0
```

Note that I did not need to specify the return type in this case, the compiler deduces it as decltype(3*x), which returns the double type.

Lambda syntax

The definition of a lambda function is introduced by the `[]`, also called capture specification, the reason will be clear in a moment. We have two possible syntax

```
[ capture spec]( arguments){ code;  return something}
```

or

```
[ capture spec]( arguments)-> returntype
{ code
```

The second syntax is compulsory when the return type cannot be deduced automatically.

The capture specification allows you to use inside the lambda variables in the enclosing scope, either by value (a local copy is made) or by reference.

[]	Capture nothing
[&]	Capture any referenced variable by reference
[=]	Capture any referenced variable by making a copy
[=, &foo]	Capture any referenced variable by making a copy, but capture variable `foo` by reference
[bar]	Capture only `bar` by making a copy
[**this**]	Capture the `this` pointer of the enclosing class

The capture specification gives a great flexibility to the lambdas
We make some examples: return the first element i such that $i > x$ and $i < y$

```
#include<algorithm>
int f(vector<int> const &v,int x, int y)
auto pos = find_if (coll.cbegin(), coll.cend(),   // range
     [=](int i) {return i > x && i < y;});// criterion
return *pos;
```

An example of use of [this]

```
class foo{
public:..
  void prova();
private:
double _x;
vector<double> _v};
... // definition
void foo::prova(){
  auto prod=[this](double a){_x*=a;};
  std::for_each(_v.begin(),_v.end(),prod);
}
```

Now the method `prova()` compute a cumulative product of the contents of `v` and stores it in the member variable `_x`.

Binders

C++11 binders have been taken from the Boost. `bind()` binds parameter for a callable object: if a function, member function, function object, lambda requires some parameters you can bind them so specific or passed arguments. For the passed arguments one uses predefined placeholders `_1`, `_2`,.. defined in the namespace `std::placeholders`.

A simple (useless) example

```
#include<iostream>
#include<functional>
double fun(double a, double b){return a*b;}

int main(){
  using namespace std;
  using namespace std::placeholders;
  auto f=bind(fun,3.0,_1);
  cout<<f(4);// calls fun(3,4)
}
```

The placeholder `_1` indicates the first (and only) passed argument to the bound function `f`. So `f(a)` is equivalent to `fun(3.0,a)`. Note the use of `using`, otherwise we should have written `std::placeholders::_1`.

Binding member functions

```
class foo{
public:
 double fun(double,int);
};
...
auto f=bind(&foo::fun,_1,3.0,6);
foo pippo;
double d= f(pippo)// calls pippo.fun(3.0,6)
...
auto g=bind(&foo::fun,pippo,_2,_1);
d=g(9,4.5);// calls pippo.fun(4.5,9)
```

When you bind to a member function, the second argument of `bind` defines the object for which the member function is called.

Passing reference to bound function

If an argument should be passed by reference to the bound function you need to use `ref()` or `cref()` (for const references). If not the arguments are passed by value!

```
void f(Matrix & n1, double const & n2);
int main(){
 using namespace std;
 using namespace std::placeholders;
 Matrix A;
 auto bound_f = bind(f, ref(A), _1);
 bound_f(35.0)// calls f(A,35.0) A passed by ref.
```

`bind()` versus lambda

Everything that can be done by using `bind()` can be done with `lambdas`. It is a matter of taste.

Personally I find lambdas great!

Function type wrappers

An now the catchall wrapper. The class `std::function<>` declared in `<functional>` provides polymorphic wrappers that generalize the notion of function pointer. It allows you to use callable objects (functions, member functions, function objects and lambdas) as first class objects.

```
int func(int, int);
...
// a vector of functions
vector<function<int(int,int)>> tasks;
tasks.push_back(fun);
tasks.push_back([](int x,int y){return x*y;});
for (auto i : tasks) cout<<i(3,4)<<endl;
```

It prints the result of `func(3,4)` and 12.

Function wrappers are very useful when you want to have a common interface to callable objects.

It may wrap also member functions.

New function declaration syntax

C++11 has introduced a new function declaration syntax

```
// new syntax for double fun(double, int)
auto fun(double, int) -> double;
```

The reason is that sometimes the return type of a function depends on an expression processed with the arguments

```
template<class T1, class T2>
auto add (T1 x, T2 y) -> decltype(x+y);
```

The return type is defined as the type of the result of $x + y$. Using the more common syntax

```
template<class T1, class T2>
 decltype(x+y) add (T1 x, T2 y);
```

is an error since `x` and `y` are not in scope when `decltype` is used.

unordered containers (hash tables)

New containers have been added, called unordered_set<> and unordered_map (and the corresponding `multi` version) with $O(1)$ access time. They are based on hash tables. For all PODs, string, complex<>, tuple<> of POds an hash function is already provided, otherwise the user has to provide the equality operator equal_to<> and the hash function by specializing std::hash<T>.

The semantic of unordered containers is otherwise practically identical to their ordered counterparts.

New begin/end

We now have cbegin() and cend() that return constant iterators and begin() and end() implemented as free functions (makes easier to make existing container-type classes std-compliant).

Boost tuple library has been ported in the language.

```
#include <tuple>
...
// Create a 4 elements tuple.
tuple<string,int,int,complex<double>> t;
// create and initialize a tuple explicitly
tuple<int,float,string> t1(41,6.3,"nico");
///extract an element
cout << get<1>(t1) << "␣";
get<2>(t1)="mario";// change an element
// use the utility make_tuple
auto t2 = make_tuple(22,44,"nico");
```

They have been made compatible with pair<>!.

Random number and distributions

We finally have a decent random number and statistical univariate distribution tool in the language!

```
#include <random>
#include <iostream>
int main(){
  std::random_device gen;
  std::uniform_int_distribution<> dis(1, 6);
  for(int n=0; n<10; ++n)std::cout << dis(gen) << ' ';
  std::cout << std::endl;
}
```

New strings operator

Operators to convert string to numbers have been enhanced:

```
#include <string>
double b=std::stod(std::string("9.0"));
```

Regexp

We now have full support for regepx. Very useful to parse ascii files!

```
const char *reg_esp = "[ ,.\\t\\n;:]";
// this can be done using raw string literals:
// const char *reg_esp = R"([ ,.\t\n;:])";
std::regex rgx(reg_esp);
std::cmatch match;
const char *target = "Unseen University - Ankh-Morpork";
// Identifies all words of 'target' separated by characters of 'reg_
if (std::regex_search(target, match, rgx)) {
  // If words separated by specified characters are present.
   auto n = match.size();
  for (decltype(n) a = 0; a < n; a++) {
    std::string str (match[a].first, match[a].second);
     std::cout << str << "\n";
  }}
```

Variadic templates

A special type of template that can take an arbitrary number of arguments. The use is rather technical but it is a very powerful feature(see the nice lecture by Alexandrescu on GoingNavive 2012). We show here only the simplest use of it

```
template<typename... Arguments>
void SampleFunction(Arguments... params);// decl only.
// Specializations
template<>
void SampleFunction(){...}
template<typename T>
void SampleFunction(T a){...}
template<typename T1, typename T2>
void SampleFunction(T1 a, T2 b){...}
```

Variadic templates

And what about a static composer class:

```
template<typename... BaseClasses>
class Composite : public BaseClasses...
{....};
```

Type traits

A full set of type traits (many taken from Boost) have been added to help implement concepts in templates, or conditional compilation. Too many to explain them here, we give just a few examples:

```
#include<type_traits>
template<class B, class D>
class{
static_assert(is_base_of<B,D>::value,"B must be base class of D");
...
};
```

Organization of standard type_traits

- Primary type categories: is_int<T>, is_pointer<T>, is_function<T>, is_rvalue_reference<T>, is_enum<T>, etc. Used to interrogate some fundamental characteristic of types.
- Composite type categories: is_scalar<T>, is_reference<T>, is_member_pointer<T> etc.
- Type properties: is_const<T>, is_trivial<T>, is_abstract<T>, is_polymorphic<T>
- Supported operations: is_copy_constructible<T>, is_assignable<T>, has_vistual_destructor<T>.
- Type relationships: is_base_of<B,D>, is_convertible<From,To>
- Type modifications: remove_const<T>, add_const<T>, make_unsigned<T>.

Support for multithreding

C++11 introduces high level support for concurrent programming. Mutexes, critical sections, spawning processes.. etc. The underlying protocol is left to the implementation.
As a consequence new versions of the Standard Library may well be multi-threaded.

Better exception handling

throw() declaration is now deprecated and replaced by the new keyword noexcept.
Standard exception handling have been expanded and all handling of standard exception objects has been revised.

Miscellanea

- Full support of ISO encoding, encoding conversion. Connection to the machine locale.
- ratio<> classes for compile time rational numbers, with predefined objects.
- Clocks and timers.
- iota

```
std::list<int> l(10);
std::iota(l.begin(), l.end(), -4);
```

Now l contains $[-4, -3, \cdots, 5]$

Out of date

Deprecated since C++11:

- **auto**+ptr, replaced by unique_ptr.
- The old use of **export**.
- **throw**() specification.