Institut Mines-Télécom
Didier GUERIOT (Dpt. ITI)
Johanne VINCENT (Dpt. INFO)
Reda BELLAFQIRA (Dpt. ITI)
Didier.Gueriot@imt-atlantique.fr
Johanne.Vincent@imt-atlantique.fr
Reda.Bellafqira@imt-atlantique.fr

Advanced C++ programming

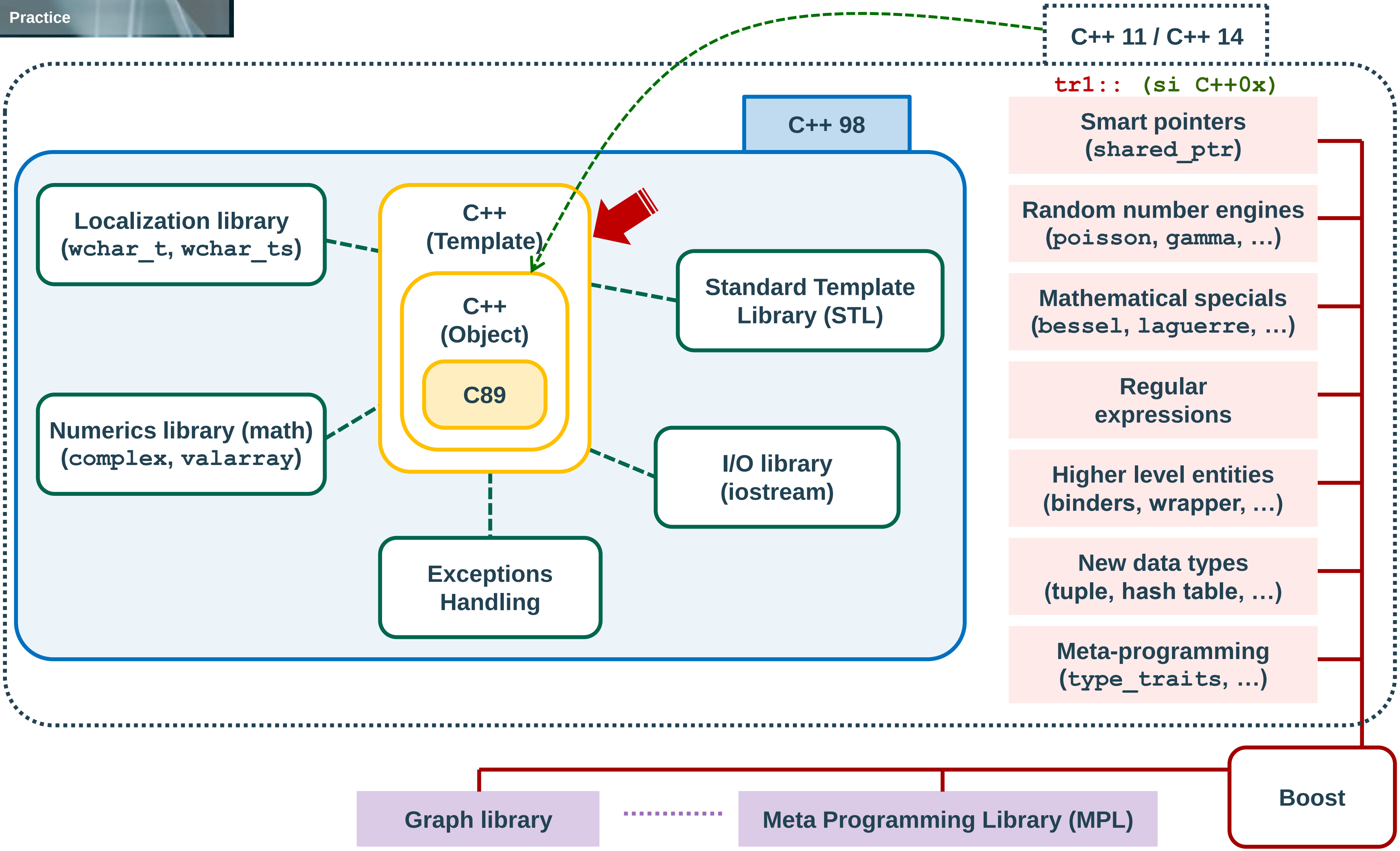
C++03
C++11
C++14
C++17
C++20

- **Introduction to C++**
  - → C++: from C and beyond
  - → Classes, objects and lifetime (vs. JAVA)
  - → Oriented-Object Programming (inheritance, polymorphism)
- **Memory management & object manipulation**
  - → References, operators, « copy » object construction
  - → « move » object construction, lambda functions
- **Template vs OO programming**
  - Slot 6 → Template functions and classes
- **The Standard Template Library**
  - → Containers, iterators and algorithms
  - → Using sequence & associative containers …
- **Smart pointers (STL & Boost)**

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

Introducing « templates »

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom



IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

« Templates » ?

- **Goal**
  - ▶ Do not write similar C++ code several times, when only the entity types change
  - Key idea: consider a « type » as another parameter…
- **When this happens, you create:**
  - Template functions
  - Template classes
- **On the road to meta-programming…**
  - You ask much more to the compiler…

Template functions (1/2)

- **Avoid redundancies**
  - Same algorithm, different types
  - Maintenance issues?

```
double min (double a , double b) {
  return a > b ? b : a ;
}

int min (int a , int b) {
  return a > b ? b : a ;
}

…
```

- **Introduction of template functions**
  - New keywords:
    - « **template** »: indicates the compiler how to create functions having the same code but operating on generic object types
    - « **typename** »: gives a « name » to the generic type

```
template <typename T>
T min (T a , T b) {
  return a > b ? b : a ;
}
```

TP 13

Template functions (2/2)

```
template <typename T>
T min (T a , T b) {
  return a > b ? b : a ;
}
```

- **How does it work ?**
  - Using a template function, the compiler “writes down” the specific code for all the required functions (with a specific type) before compiling them

```
double c = min (10.4 , 23.5) ;
int    d = min (10 , 23) ;
```

```
double min_double (double , double) ;
int    min_int    (int , int) ;
```

  - two functions « min » are written down by the compiler
  - each « min » function deals with one specific type: `double` then `int`
- Once written down, the compiler operates as usual:
  - all checking performed when compiling, **not at runtime**

```
Point  C = min (A , B) ;
```

compilation

The compiler will output an error when compiling the “min function dedicated to `Point` objects”, if operator “>” is not defined for `Point` class

Using template functions (1/5)

```
template <typename T>
T min (T a , T b) {
  return a > b ? b : a ;
}
```

- **Exact matching types rule**
  - Template functions arguments (the values/variables passed when calling the function) must exactly match the types given with the template declaration

compilation

```
double c = min (10.4 , 23) ;
```

- 10.4 is a floating value (type: `double`), 23 is an integer (type: `int`)
- function « `double min (double a , int b)` » does not exist!

➡ Solution: tell the compiler to produce and use a specific templated function

```
double c = min<double> (10.4 , 23) ;
```

- "`double min (double, double)`" will be written down, then compiled
- type casting may then occur as usual (here, 23 will be cast to `double`)

Using template functions (2/5)

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

```
template <typename T>
T min (T a , T b) {
  return a > b ? b : a ;
}
```

- **Several template types: OK**

```
template <typename T1 , typename T2>
T1 func (T1 one , T2 two , T1 three) {…}
```

- **Overloading template functions: OK**

```
template <typename T>
T min (T a , T b , T c) {
  T i = min (a , b) ;
  return min (i , c) ;
}
```

```
min (5 , 10 , 15) ;
min (4.5 , 15.75 , 8.25) ;
min (2 , 3.5 , 3.75) ;
```

- **Explicit template specialization (partial or full): OK**
  - You may give an extra specific version for specific types, bypassing the templated version for these types
  - ▶ The compiler selects the **most specific version**

```
template <>
string min (string a , string b) {
  return a.length() > b.length() ? b : a ; }
```

```
min (5 , 10) ;
min (4.5 , 8.75) ;
min ("tomate" , "chou")
```

Using template functions (3/5)

- **Mixing overloading and specialization**
  - How does the compiler behave?

The compiler deals with overloading first,
then selects the most specific template version

- Example 1:

1
```
template <typename T>
void foo (T t)
```

Overloading

2
```
template <typename T>
void foo (T* t)
```

Specialization

3
```
template <>
void foo<int> (int t)
```

```
foo (12) ;
foo ((void*)NULL) ;
```

Using template functions (3/5)

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

• Mixing overloading and specialization

- How does the compiler behave?

The compiler deals with overloading first, then selects the most specific template version

• Example 2:

1

```
template <typename T>
void foo (T t)
```

Overloading

2

```
template <typename T>
void foo (T* t)
```

Specialization

3

```
template <>
void foo<int*> (int* t)
```

```
foo ((int*)NULL) ;
```

Using template functions (3/5)

- **Mixing overloading and specialization**
  - How does the compiler behave?

The compiler deals with overloading first, then selects the most specific template version

- Example 3:

1
```
template <typename T>
void foo (T t)
```

Overloading

2
```
template <typename T>
void foo (T* t)
```

Specializations

3
```
template <>
void foo<int> (int t)
```

4
```
template <>
void foo<int*> (int* t)
```

Specialization

5
```
template <>
void foo<int> (int* t)
```

```
foo ((int*)NULL) ;
```

Using template functions (4/5)

- `template<...>` must list all template types
  - for parameters, for returned value, inside the function
  - Example: « universal » conversion function between types

```
template <typename T , typename W>
T transform(W a) {
  return (T)a ;
}
```

```
transform<Point>(10) ;
```

Type casting W → T must exist
(if not, error when compiling)

- `template<...>` also accepts
  - numerical value types: `int`, `short`, `char`
  - pointers

```
template <typename T , int N=2>
T get (T* v) {
  return v[N] ;
}
```

Using template functions (5/5)

- A method can also be a template method
  - The compiler will only write down (insert into the class) the actually called specific methods

```
class A {
  int i ;     // Class attribute (data)

public:
  // This is a template method: the compiler will provide all the specific methods "add"
  //   according to actual calls to "add" (with specific types for "inc")
  template <class T>
  void add (T inc) ;
} ;

// Method implementation
template <class T>
void A::add (T inc) {
  // Add "inc" to "A::i"
  // error when compiling
  // if "inc" cannot be cast to int
  i = i + (int)inc ;
  return ;
}
```

```
A obj ;

// Compiler provides "void A::add(int)" to class A
obj.add(10) ;

// Compiler provides "void A::add(double)" to class A
obj.add(20.5) ;

// Compiler tries to provide "void A::add(char*)"
// to class A but fails as its code cannot compile
obj.add ("toto") ;
```

Template classes (1/4)

TP 14

- **Same requirements**
  - Avoid useless redundancies while authorizing specialization for **class** definition

```
class PointDouble {
  double x , y ;
}
```

```
class PointInt {
  int x , y ;
}
```

- **Example: generic `Point` class?**
  - same structure, different internal types
  - same behavior

➡ Solution : coordinates type ≡ template type? YES

```
template <typename T>
class Point {
  T x , y ;
}
```

```
Point<double> P ;
Point<int> I ;
```

type of `P`: `Point<double>`
type of `I`: `Point<int>`

Template classes (2/4)

- **Extending the example...**
  - Make the dimension of a point generic ? 2D, 3D, 4D, ...

```
template <typename T , int N>
class Point {
  T v[N] ;
}
```

```
Point<double,3> P ;
Point<int,2> I ;
```

type of P: `Point<double,3>`
type of I: `Point<int,2>`

  - Default template type value

```
template <typename T , int N=2>
class Point {
  T v[N] ;
}
```

```
Point<double> P ;
Point<int,4> I ;
```

type of P: `Point<double,2>`
type of I: `Point<int,4>`

Template classes (3/4)

- **File organization**
  - Template class `Point` is fully defined in "`Point.h`" file

```
#include "Point.h"

// Constructor [1]
Point<int,3> P3 ;

// Constructor [2]
Point<int,3> M3 (P3) ;

// Constructor [1]
Point<int> I2 ;

// Constructor [3]
// Error when compiling if
// constructor [3] does not exist
Point<double,2> D2 (I2) ;
```

Point.h

```
template <typename T , int N=2>
class Point
{
public :
  T v[N] ;

  // [1] Default constructor (to write if wanted: it will not
  // exist automatically as other constructors are defined)
  Point () {
    for (int k = 0 ; k < N ; k++) v[k] = 0 ;
  }

  // [2] Copy constructor (same dimension / coordinates type)
  Point (const Point<T,N>& P) ;

  // [3] Copy constructor (from a same dimension Point
  // but with different coordinates type
  template <typename W> Point (const Point<W,N>& P) ;

} ;

template <typename T , int N>
Point<T,N>::Point (const Point<T,N>& P) {
  for (int k = 0 ; k < N ; k++) v[k] = P.v[k] ;
}

template <typename T , int N>
template <typename W>
Point<T,N>::Point (const Point<W,N>& P) {
  for (int k = 0 ; k < N ; k++) v[k] = transform<T>(P.v[k]) ;
}
```

Template classes (4/4)

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

- No standalone compilation for template class "Point"

The compiler must know the full source code of a template class (here `Point`) to write down the source code of its specific version (here `Point<int,3>`) and compile it!

```
#include "Point.h"

// Constructor [1]
Point<int,3> P3 ;

// Constructor [2]
Point<int,3> M3 (P3) ;

// Constructor [1]
Point<int> I2 ;

// Constructor [3]
// Error when compiling if
// constructor [3] does not exist
Point<double,2> D2 (I2) ;
```

Point.h

```
template <typename T , int N=2>
class Point
{
public :
  T v[N] ;

  // [1] Default constructor (to write if wanted: it will not
  // exist automatically as other constructors are defined)
  Point () {
    for (int k = 0 ; k < N ; k++) v[k] = 0 ;
  }

  // [2] Copy constructor (same dimension / coordinates type)
  Point (const Point<T,N>& P) ;

  // [3] Copy constructor (from a same dimension Point
  // but with different coordinates type
  template <typename W> Point (const Point<W,N>& P) ;

} ;

template <typename T , int N>
Point<T,N>::Point (const Point<T,N>& P) {
  for (int k = 0 ; k < N ; k++) v[k] = P.v[k] ;
}

template <typename T , int N>
template <typename W>
Point<T,N>::Point (const Point<W,N>& P) {
  for (int k = 0 ; k < N ; k++) v[k] = transform<T>(P.v[k]) ;
}
```

Building a binary executable

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

TP 14

main.cpp

```
#include "MaClasse.h"

int main (int argc, char* argv[])
{
  MaClasse<int> c ;

  c.run() ;

  return 0 ;
}
```

MaClasse.h

```
#ifndef __MACLASSE
#define __MACLASSE
#include <math>

template <typename T>
class MaClasse {
public :
  T hauteur ;

  double run () ;
}

template <typename T>
double MaClasse::run () {
  return std::cos(0) ;
}
#endif
```

math.h

```
namespace std {
…
  double cos (double) ;
…
}
```

libm.so

01010110111
10100010010
01010000101

1

Compilation :

```
g++ -c main.cpp
```

main.o

0101011
1010001
0101000

2

Edition de liens :

```
g++ -o mon_prog main.o -lm
```

mon_prog

01010110111
10100010010
01010000101

```
-std=c++11 -Wall
```

Introducing constraints (1/3)

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

- **Why giving several versions ?**

  ⇒ tell explicitly which models are authorized

- **Get along with everything generic**
  - always possible
  - BUT no help given

```
// Copy constructor (from everything>)
template <typename Obj>
Point (const Obj& P) {
  for (int k = 0 ; k < N ; k++)
    v[k] = P.v[k] ;
}
```

Point.h

```
template <typename T , int N=2>
class Point
{
public :
  T v[N] ;

  // [1] Default constructor (to write if wanted: it will not
  // exist automatically as other constructors are defined)
  Point () {
    for (int k = 0 ; k < N ; k++) v[k] = 0 ;
  }

  // [2] Copy constructor (same dimension / coordinates type)
  Point (const Point<T,N>& P) ;

  // [3] Copy constructor (from a same dimension Point
  // but with different coordinates type
  template <typename W> Point (const Point<W,N>& P) ;

} ;


template <typename T , int N>
Point<T,N>::Point (const Point<T,N>& P) {
  for (int k = 0 ; k < N ; k++) v[k] = P.v[k] ;
}

template <typename T , int N>
template <typename W>
Point<T,N>::Point (const Point<W,N>& P) {
  for (int k = 0 ; k < N ; k++) v[k] = transform<T>(P.v[k]) ;
}
```

Introducing constraints (2/3)

- **Going further to specify constraints on template type**
  - SFINAE mechanism when compiling
    - "Substitution failure is not an error"

C++11

Verbose + not easy to read

```
class Base {} ;
class Derived : public Base {} ;

// Substitution OK when T is a type derived from Base
template<typename T,
  std::enable_if_t<std::is_baseof_v<Base, T>, bool> Dummy = true
>
void foo (T t) {}

// Substitution OK when T is not a type derived from Base
template<typename T,
  std::enable_if_t<not std::is_baseof_v<Base, T>, bool> Dummy = true
>
void foo (T t) {}
```

```
int main() {

  Derived der ;

  foo(der) ;
  foo(123) ;
}
```

- **Improvements as C++ evolves but...**
  - still exposing compiler internal mechanisms making the code and compiler error messages difficult to read!

C++14

C++17

Introducing constraints (3/3)

- **How to easily tell the kind of template that may be provided** (the compiler will check)?
  - New keyword "**concept**"

C++20

```
#include <string>
#include <cstddef>
#include <concepts>
using namespace std::literals;

// Declaration of the concept "Hashable", which is satisfied by
// any type T such that for values a of type T:
//   - the expression std::hash<T>{}(a) compiles
//   - its result is convertible to std::size_t
template<typename T>
concept Hashable = requires(T a) {
  { std::hash<T>{}(a) } -> std::convertible_to<std::size_t>;
};

struct meow {} ;        // "meow" does not satisfy Hashable concept

template<Hashable T>    // Constrained C++20 function template
void f(T);              //   T must be Hashable

int main() {
  f("abc"s);    // OK, std::string satisfies Hashable
  f(meow{});    // Error: meow does not satisfy Hashable
}
```

```
// Alternative way to apply
//  the same constraint:

template<typename T>
  requires Hashable<T>
void f(T);
```

Interesting features (1/3)

C++11

- **Variadic templates**
  - Take an arbitrary number of template arguments of any type

```
void print () { }

template <typename T, typename... Types>
void print (const T& firstArg , const Types&... args) {
  cout << firstArg << endl ;
  print (args...) ;
}
```

```
print (7.5 , "hello" , std::bitset<16>(377) , 42) ;
```

  - Allow collections of different types objects: `std::tuple<>`

std::tuple <>

```
template<typename... V>
class tuple ;
```

```
typedef std::tuple <int , string , double> product_t ;

product_t mag (100 , "Science" , 6.5) ;

// Full access to every element of the tuple object
int id = std::get<0>(mag) ;
std::get<1>(mag) = "USA Today" ;

// Another way to build a tuple
auto mag2 = std::make_tuple (101 , "TV mag", 8.5) ;
```

Interesting features (2/3)

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

C++11

- Template aliases
  - **using** : allow "typedef" for templates

```
template <typename T>
using vec = std::vector<T> ;

// vec is an alias for std::vector<>
vec<int> collection ;
```

```
typedef void (*Type)(double) ;
using OtherType = void (*)(double) ;
```

```
template<size_t N , size_t M>
class Matrix { // ....
} ;

typedef Matrix<N,1> Vector<N> ;

template <int N>
using Vector = Matrix<N,1> ;
```

compilation

- Extern templates
  - Tell the compiler not to instantiate a template as you know it will be instantiated somewhere else ⇒ compile time optimization

```
template <typename X> class A {
public :
   void test(X) ;
} ;
```

```
// AA sera connue mais pas instanciée ici : extern dit
// que l'instanciation sera faite plus loin ou dans un
// autre module ≠ template class AA<int> { ... } ;
extern template class AA<int> ;
```

Interesting features (3/3)

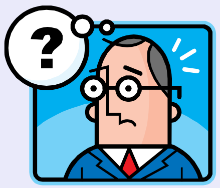
- Type inference & new syntax for functions

```
// How to write the type of
// the returned object ???
template<class T, class U>
??? add (T x, U y) {
  return x+y ;
}
```

```
// When reading decltype(x+y),
// the compiler has never seen
// x or y: error
template<class T, class U>
decltype(x+y) add (T x, U y) {
  return x+y ;
}
```

compilation

```
// Correct solution but difficult to read
template<class T, class U>
decltype(*(T*)(0)+*(U*)(0)) add (T x, U y) {
  return x+y ;
}
```



C++11

```
// Using the function new syntax
template<class T, class U>
auto add (T x, U y) -> decltype(x+y) {
  return x+y ;
}
```

```
// Other solution using type inference on
//   function return value
template<class T, class U>
decltype(auto) add (T x, U y) {
  return x+y ;
}
```

C++14

Generic programming (1/3)

- **Introducing the "concept" notion**
  - 1 concept ≡ a set of "services" or properties, a type must provide or satisfy (≈ OOP interface)
  - A concept allows to specify how an entity may be used.
- **C++ implementation (≠ JAVA generics)**
  - Through **templates** mechanisms
    - no inheritance: two classes satisfying a concept must only implement a common set of method signatures
    - naming the used concept / entities is very **important** as it helps understanding the purpose / actual use of the entity
  - Everything's checked and done when compiling

Generic programming (2/3)

• OOP vs. Generic Programming

Runtime polymorphism

```
// Oriented Object Programming
struct Base
{
  virtual void Foo() = 0 ;
} ;

struct A : public Base
{ void Foo() ; } ;

struct B : public Base
{ void Foo() ; } ;

// The "Foo" method of actual "obj"
// type is chosen using the virtual
// function table
// at RUNTIME
void Bar (Base& obj)
{ obj.Foo() ; }
```

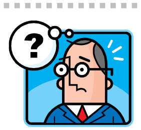
```
// Generic programming
struct A {
  inline void Foo() ;
} ;

struct B {
  inline void Foo() ;
} ;

template<typename T>
void Bar (T& obj) {
  obj.Foo() ;
}

// The compiler produces two specific "Bar"
// functions: "void Bar(A&)" and "void Bar(B&)"
// if these are actually called. Everything's done
// when COMPILING
```

Compilation polymorphism




```
A a ;
B b ;
```

```
Bar (a) ;
Bar (b) ;
```

Generic programming (3/3)

- **OOP vs. Generic Programming**
  - Oriented Object Programming (inheritance)
    - When calling a function, the provided value types must match or derive from the types given in the declaration
    - Use of virtual methods: runtime actual method selection ▸ extra cost
  - Generic Programming (templates)
    - When calling a function, the provided value types must only implement the methods that are called on them, in the function
    - Use of template mechanism:
      - Everything's done when compiling ▸ no extra cost at runtime
      - All types must be known when compiling
- **Two different approaches**
  - ▸ **The STL uses the "Generic Programming" approach**

Practice

- **Template functions**
  - Writing a template function
  - Building a program that uses template functions
  - Using template functions with objects

- **Going further with template classes...**
  - A “Tree” template class ?