C++ Express

CEMRACS 2012

7 Août

Pascal Havé

<pascal.have@ifpen.fr>

Le C++ : programmation générique

- Problèmatique:
  - *Après avoir fait un vecteur d'entiers, de doubles… comment faire un tableau de «Truc» ?*

- D'après ce que nous avons déjà vu:
  1. Tableau de IObject, *Truc* héritant de IObject
     - Y mettre tout et n'importe quoi (héritant de IObject)
     - Surcoût des accès (mécanisme d'héritage + virtual)
  2. Mettre le type des données en *typedef*
     - Contrôle du contenu et accès optimal
     - Un seul type à la fois
  3. Utilisation de #define VECTOR(T)
     - Typiquement C, polyvalent et « optimal »
     - Difficilement extensible et débuggable

Le C++ : programmation générique

- Les Templates (Modèles)
  - L'utilisation de types comme des paramètres

```
Vector<int> vint;
Vector<double> vdouble;
Vector<Truc> vtruc;
```

    - Un seul source, multiple utilisations, instanciation « optimale »
    - Contraintes de compilation (organisation du code, temps de compilation)
    - Syntaxe parfois un peu *tricky*
      - Imbrication avec les notions déjà vues
      - Exploitation d'effets de bord

Le C++ : programmation générique

Exemples

```
// Classe Vector de T indexé par des SizeT (int par défaut)
template<typename T, typename SIzeT = int>
class Vector
{
// maintenant T défini un type dans toute cette classe
public:
  typedef T ContainerData;
public:
  T & operator()(const SizeT i);
  // On peut combiner Vector<T> avec Vector<U>
  template<typename U> void operator=(Vector<U> & v);
private:
  SizeT m_size;
  T * m_data;
};
```

```
// On peut réutiliser un type après sa
// déclaration template
template<class T, T value>
void fill(Vector<T> & v)
{
  /* … */
}
```

```
// Les paramètres peuvent aussi être
// des valeurs (ici int)
template<typename U, int N>
class TinyVector
{
  /* … */
};
```

```
Vector<int> vint;
Vector<double> vdouble;
Vector<Truc> vtruc;
```

- Le type template est muet, il n'a pas de sens en dehors du scope de la déclaration template courante.

Le C++ : programmation générique

Code commenté Vector 1/3

- En première approximation, écrire une classe générique peut s'obtenir en écrivant à partir d'un *typedef* puis en ajoutant les mots clefs *template*

Classe Vecteur de données de type T (typedef)

```
#include <cassert>
#include <iostream>
typedef int T;

/***************** début de vector.hpp ******************/


class Vector {
public:
  Vector(int n = 0) : m_size(n), m_data(new T[n]) { }
  Vector(const Vector & v);
  Vector & operator=(const Vector & v);
  virtual ~Vector() { delete[] m_data; }

public:
  inline int size() const { return m_size; }

  T & operator()(int i) { assert(i>=0&&i<m_size); return m_data[i]; }
  const T & operator()(int i) const;

  Vector operator*(T a) const;

  friend std::ostream & operator<<(std::ostream& o, const Vector& v)
    { v.print(o); return o; }

private:
  void print(std::ostream & o) const
    { for(int i=0;i<m_size;++i) o << m_data[i] << " "; }
  int m_size;
  T * m_data;
};
```

Classe Vecteur de données de type T (template)

```
#include <cassert>
#include <iostream>

/***************** début de vector.hpp ******************/

template<typename T>
class Vector {
public:
  Vector(int n = 0) : m_size(n), m_data(new T[n]) { }
  Vector(const Vector<T> & v);
  Vector<T> & operator=(const Vector<T> & v);
  virtual ~Vector() { delete[] m_data; }

public:
  inline int size() const { return m_size; }

  T & operator()(int i) { assert(i>=0&&i<m_size); return m_data[i]; }
  const T & operator()(int i) const;

  Vector<T> operator*(T a) const;

  friend std::ostream & operator<<(std::ostream& o, const Vector<T>& v)
    { v.print(o); return o; }

private:
  void print(std::ostream & o) const
    { for(int i=0;i<m_size;++i) o << m_data[i] << " "; }
  int m_size;
  T * m_data;
};
```

Le C++ : programmation générique

Code commenté Vector 2/3

Classe Vecteur de données de type T (typedef)

```
// Méthodes de la classe


Vector::
Vector(const Vector & v)
  : m_size(v.m_size) {
  m_data = new T[m_size];
  for(int i=0;i<m_size;++i) m_data[i] = v(i);
}

Vector &
Vector::
operator=(const Vector & v) {
  if (this != &v) {
    if (m_size != v.m_size)
      {
        delete[] m_data;
        m_size = v.m_size;
        m_data = new T[m_size];
      }
    for(int i=0;i<m_size;++i) m_data[i] = v(i);
  }
  return *this;
}

Vector
Vector::
operator*(T a) const {
  const Vector & v = *this;
  Vector newv(m_size);
  for(int i=0;i<m_size;++i) newv(i) = a*v(i);
  return newv;
}

const T &
Vector::
operator()(int i) const {
  assert(i>=0 && i<m_size);
  return m_data[i];
}
```

Classe Vecteur de données de type T (template)

```
// Méthodes de la classe

template<typename T>
Vector<T>::
Vector(const Vector<T> & v)
  : m_size(v.m_size) {
  m_data = new T[m_size];
  for(int i=0;i<m_size;++i) m_data[i] = v(i);
}
template<typename T>
Vector<T> &
Vector<T>::
operator=(const Vector<T> & v) {
  if (this != &v) {
    if (m_size != v.m_size)
      {
        delete[] m_data;
        m_size = v.m_size;
        m_data = new T[m_size];
      }
    for(int i=0;i<m_size;++i) m_data[i] = v(i);
  }
  return *this;
}
template<typename T>
Vector<T>
Vector<T>::
operator*(T a) const  {
  const Vector<T> & v = *this;
  Vector<T> newv(m_size);
  for(int i=0;i<m_size;++i) newv(i) = a*v(i);
  return newv;
}
template<typename U>
const U &
Vector<U>::
operator()(int i) const {
  assert(i>=0 && i<m_size);
  return m_data[i];
}
```

Le C++ : programmation générique

Code commenté Vector 3/3

```cpp
// Fonctions

Vector
operator*(T a, const Vector & v)
{
  return v*a;
}

/****************** fin de vector.hpp ******************/

int main() {
  const int n = 10;

  Vector u(n);
  for(int i=0;i<n;++i) u(i) = 10+i; // initialisation
  std::cout << u << "¥n";

  Vector v; v = u;
  std::cout << v << "¥n";

  Vector w = v*2;
  std::cout << w << "¥n";

  Vector x; x = 2*w;
  std::cout << x << "¥n";
}
```

Classe Vecteur de données de type T (typedef)

```cpp
// Fonctions
template<typename T>
Vector<T>
operator*(T a, const Vector<T> & v)
{
  return v*a;
}

/****************** fin de vector.hpp ******************/

int main() {
  const int n = 10;

  Vector<int> u(n);
  for(int i=0;i<n;++i) u(i) = 10+i; // initialisation
  std::cout << u << "¥n";

  Vector<int> v; v = u;
  std::cout << v << "¥n";

  Vector<int> w = v*2;
  std::cout << w << "¥n";

  Vector<int> x; x = 2*w;
  std::cout << x << "¥n";
}
```

Classe Vecteur de données de type T (template)

Le C++ : programmation générique

- Petites particularités des templates
  - Erreurs de compilation plus ou moins obscures
    - Seul les parties instanciées sont compilées, des bugs de compilation peuvent toujours se cacher dans ce qui n'est pas utilisé

```
template<typename T>
struct LazyError {
  static T error(const T & t) { return t.error(); }
};

{
  LazyError<int> bug; // Pas d'erreur de compilation
  bug.error(); // bug de compilation
};
```

    - C'est aussi une technique utilisée dans des mécanismes avancés
    - Conseil : éviter d'écrire des bugs…
  - Syntaxe du double chevron : > « espace » > (C++03)
  - Template uniquement sur les valeurs de types intégraux : int, long…
  - Les besoins de visibilité pour l'instanciation spécifique à la compilation impose souvent de tout mettre dans des headers (.h)

STL : Standart Template Library

- La bibliothèque standard du C++
  - Nativement fournie avec tous les compilateurs C++
  - Orientée structure de données *simples*
  - Générique via le mécanisme de *template*
  - Namespace std
    - http://www.cplusplus.com/reference
    - http://www.cppreference.com

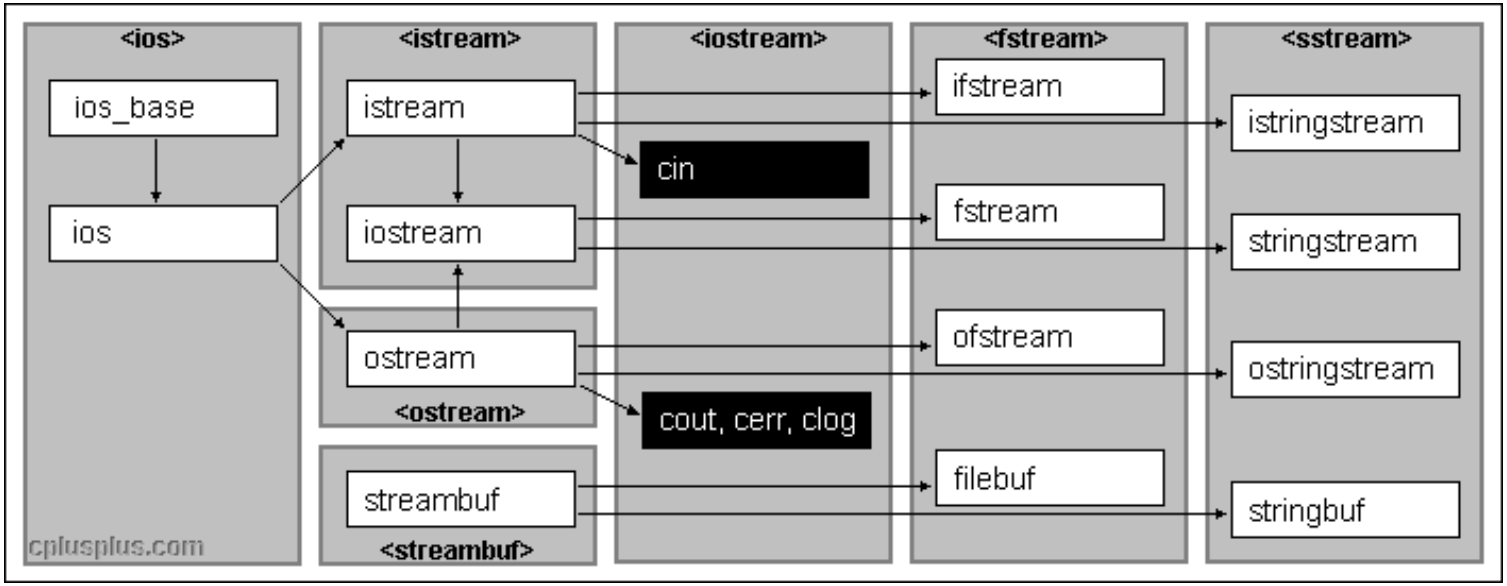
STL : Chaîne de caractères <string>

```
template< typename Char,
          typename Traits = std::char_traits<Char>,
          typename Allocator = std::allocator<Char>
        > class basic_string;
```

- *typedef* basic_string<*char*> **string**;
- Fonctions en vrac
  - algorithme: getline
  - container: size, length, resize
  - modificateurs: +=, erase, insert
  - méthodes propres : c_str, copy, find, rfind, substr, replace, compare
- NBs:
  - 
    Attention aux collisions avec les C String (#include <cstring>)

```
#include <iostream>
#include <string>
using namespace std;

int main ()
{
  string str ("There are two needles in this haystack with needles.");
  string str2;
  getline(cin,str2);

  size_t found;

  found=str.find(str2);
  if (found!=string::npos)
    cout << "first found at: " << int(found) << endl;

  found=str.find("needles are small",found+1,6);
  if (found!=string::npos)
    cout << "second found at: " << int(found) << endl;

  str.replace(str.find(str2),str2.length(),"preposition");
  cout << str << endl;

  return 0;
}
```

STL : I/O



- Membres en vrac
  - ios : eof, bad, fail, operator!
  - iostream : operator<<, operator>>, flush,
  - fstream : open, close
  - stringstream : str
- NBs:
  - ⚠ Performance vs C

STL : <iterator>

- Généralisation du concept de pointeur sur des structures complexes

category				characteristic	valid expressions
all categories				Can be copied and copy-constructed	`X b(a);` `b = a;`

				Can be incremented	`++a`
					`a++`
					`*a++`

Random Access	Bidirectional	Forward	Input	Accepts equality/inequality comparisons	`a == b` `a != b`
				Can be dereferenced as an *rvalue*	`*a` `a->m`
			Output	Can be dereferenced to be the left side of an assignment operation	`*a = t` `*a++ = t`
				Can be default-constructed	`X a;` `X()`

				Can be decremented	`--a`
					`a--`
					`*a--`

				Supports arithmetic operators + and -	`a + n`
					`n + a`
					`n - a`
					`a - b`

				Supports inequality comparisons (<, >, <= and >=) between iterators	`a < b`
					`a > b`
					`a <= b`
					`a >= b`

				Supports compound assignment operations += and -=	`a += n` `a -= n`
				Supports offset dereference operator ([])	`a[n]`

- advance, distance
- insert_iterator, back_inserter, front_inserter
- reverse_iterator
- istream_iterator, ostream_iterator
- iterator_traits

STL : Conteneurs

- Pleins de conteneurs différents pour pleins de besoins différents

			Sequence containers			Associative containers				
Headers			<vector>	<deque>	<list>	<set>				<bitset>
Members		complexity	vector	deque	list	set	multiset	map	multimap	bitset
	constructor	*	constructor	constructor	constructor	constructor	constructor	constructor	constructor	constructor
	destructor	O(n)	destructor	destructor	destructor	destructor	destructor	destructor	destructor	
	operator=	O(n)	operator=	operator=	operator=	operator=	operator=	operator=	operator=	operators
iterators	begin	O(1)	begin	begin	begin	begin	begin	begin	begin	
	end	O(1)	end	end	end	end	end	end	end	
	rbegin	O(1)	rbegin	rbegin	rbegin	rbegin	rbegin	rbegin	rbegin	
	rend	O(1)	rend	rend	rend	rend	rend	rend	rend	
capacity	size	*	size	size	size	size	size	size	size	size
	max_size	*	max_size	max_size	max_size	max_size	max_size	max_size	max_size	
	empty	O(1)	empty	empty	empty	empty	empty	empty	empty	
	resize	O(n)	resize	resize	resize					
element access	front	O(1)	front	front	front					
	back	O(1)	back	back	back					
	operator[]	*	operator[]	operator[]				operator[]		operator[]
	at	O(1)	at	at						
modifiers	assign	O(n)	assign	assign	assign					
	insert	*	insert	insert	insert	insert	insert	insert	insert	
	erase	*	erase	erase	erase	erase	erase	erase	erase	
	swap	O(1)	swap	swap	swap	swap	swap	swap	swap	
	clear	O(n)	clear	clear	clear	clear	clear	clear	clear	
	push_front	O(1)		push_front	push_front					
	pop_front	O(1)		pop_front	pop_front					
	push_back	O(1)	push_back	push_back	push_back					
	pop_back	O(1)	pop_back	pop_back	pop_back					
observers	key_comp	O(1)				key_comp	key_comp	key_comp	key_comp	
	value_comp	O(1)				value_comp	value_comp	value_comp	value_comp	
operations	find	O(log n)				find	find	find	find	
	count	O(log n)				count	count	count	count	count
	lower_bound	O(log n)				lower_bound	lower_bound	lower_bound	lower_bound	
	upper_bound	O(log n)				upper_bound	upper_bound	upper_bound	upper_bound	
	equal_range	O(log n)				equal_range	equal_range	equal_range	equal_range	
unique members			capacity reserve		splice remove remove_if unique merge sort reverse					set reset flip to_ulong to_string test any none

Invalidation des références sur objets modifiés (resize, insert, erase...)

STL : Conteneurs (adapteurs)

			Container Adaptors		
Headers			<stack>	<queue>	
Members			stack	queue	priority_queue
	constructor	*	constructor	constructor	constructor
capacity	size	O(1)	size	size	size
	empty	O(1)	empty	empty	empty
element access	front	O(1)		front	
	back	O(1)		back	
	top	O(1)	top		top
modifiers	push	O(1)	push	push	push
	pop	O(1)	pop	pop	pop

template < *class* T, *class* Container = deque<T> > *class* stack;
template < *class* T, *class* Container = deque<T> > *class* queue;
template < *class* T, *class* Container = vector<T>,
class Compare = less<*typename* Container::value_type> > *class* priority_queue;

```
#include <iostream>
#include <map>

int main () {
  std::multimap<char,int> mymm;
  mymm.insert(pair<char,int>('a',10));
  mymm.insert(pair<char,int>('b',20));  mymm.insert(pair<char,int>('b',30));
  mymm.insert(pair<char,int>('b',40));
  mymm.insert(pair<char,int>('c',50));   mymm.insert(pair<char,int>('c',60));
  mymm.insert(pair<char,int>('d',60));

  cout << "mymm contains:\n";
  for (char ch='a'; ch<='d'; ch++)
  {
    std::cout << ch << " =>";
    std::pair<std::multimap<char,int>::iterator, std::multimap<char,int>::iterator>
      ret = mymm.equal_range(ch);
    for (std::multimap<char,int>::iterator it=ret.first; it!=ret.second; ++it)
      cout << " " << (*it).second;
    std::cout << std::endl;
  }
return 0; }
```

```
// priority_queue::push/pop
#include <iostream>
#include <queue>
using namespace std;

int main ()
{
  priority_queue<int> mypq;

  mypq.push(30);
  mypq.push(100);
  mypq.push(25);
  mypq.push(40);

  cout << "Popping out elements";
  while (!mypq.empty())
  {
    cout << " " << mypq.top();
    mypq.pop();
  }
  cout << endl;

  return 0;
}
```

STL : <algorithm>

« Et que puis-je faire sur ces nouvelles structures … »

- Non-modifying sequence operations:
  - for_each Apply function to range
  - find Find value in range
  - find_if Find element in range
  - find_end Find last subsequence in range
  - find_first_of Find element from set in range
  - adjacent_find Find equal adjacent elements in range
  - count Count appearances of value in range
  - count_if Return number of elements in range satisfying condition
  - mismatch Return first position where two ranges differ
  - equal Test whether the elements in two ranges are equal
  - search Find subsequence in range
  - search_n Find succession of equal values in range
- Heap
  - push_heap Push element into heap range
  - pop_heap Pop element from heap range
  - make_heap Make heap from range
  - sort_heap Sort elements of heap

STL : <algorithm>

Mais ce n'est pas tout…

- Modifying sequence operations:
  - copy Copy range of elements
  - copy_backward Copy range of elements backwards
  - swap Exchange values of two objects
  - swap_ranges Exchange values of two ranges
  - iter_swap Exchange values of objects pointed by two iterators
  - transform Apply function to range
  - replace Replace value in range
  - replace_if Replace values in range
  - replace_copy Copy range replacing value
  - replace_copy_if Copy range replacing value
  - fill Fill range with value
  - fill_n Fill sequence with value
  - generate Generate values for range with function
  - generate_n Generate values for sequence with function
  - remove Remove value from range
  - remove_if Remove elements from range
  - remove_copy Copy range removing value
  - remove_copy_if Copy range removing values
  - unique Remove consecutive duplicates in range
  - unique_copy Copy range removing duplicates
  - reverse Reverse range
  - reverse_copy Copy range reversed
  - rotate Rotate elements in range
  - rotate_copy Copy rotated range
  - random_shuffle Rearrange elements in range randomly
  - partition Partition range in two
  - stable_partition Partition range in two - stable ordering

STL : <algorithm>

Et pour conclure les algorithmes …

- Sorting
  - sort Sort elements in range
  - stable_sort Sort elements preserving order of equivalents
  - partial_sort Partially Sort elements in range
  - partial_sort_copy Copy and partially sort range
  - nth_element Sort element in range
- Recherche binaire (sur structures triées)
  - lower_bound Return iterator to lower bound
  - upper_bound Return iterator to upper bound
  - equal_range Get subrange of equal elements
  - binary_search Test if value exists in sorted array
- Fusion (sur structures triées)
  - merge Merge sorted ranges
  - inplace_merge Merge consecutive sorted ranges
  - includes Test whether sorted range includes another sorted range
  - set_union Union of two sorted ranges
  - set_intersection Intersection of two sorted ranges
  - set_difference Difference of two sorted ranges
  - set_symmetric_difference Symmetric difference of two sorted ranges
- Min / Max
  - min / max Return the lesser / greater of two arguments
  - min_element / max_element Return smallest / largest element in range

STL : <valarray>

- Valarray n'est pas un conteneur *standard*
- Valarray est un conteneur optimisé pour les types numériques
- Toutes les opérations sur valeurs : +, -, %=, <<=, ~ ...
- Constructeurs:

```
valarray();
explicit valarray (size_t n);
valarray (const T& val, size_t n);
valarray (const T* p, size_t n);
valarray (const valarray& x);
valarray (const slice_array<T>& sub);
valarray (const gslice_array<T>& sub);
valarray (const mask_array<T>& sub);
valarray (const indirect_array<T>& sub);
```

STL : <functional>

- Base classes:
  - unary_function Unary function object base class
  - binary_functionBinary function object base class
- Arithmetic operations:
  - plus Addition function object class
  - minus Subtraction function object class
  - multiplies Multiplication function object class
  - divides Division function object class
  - modulus Modulus function object class
  - negate Negative function object class
- Comparison operations:
  - equal_to Function object class for equality comparison
  - not_equal_to Function object class for non-equality comparison
  - greater Function object class for greater-than inequality comparison
  - less Function object class for less-than inequality comparison
  - greater_equal Function object class for greater-than-or-equal-to comparison
  - less_equal Function object class for less-than-or-equal-to comparison

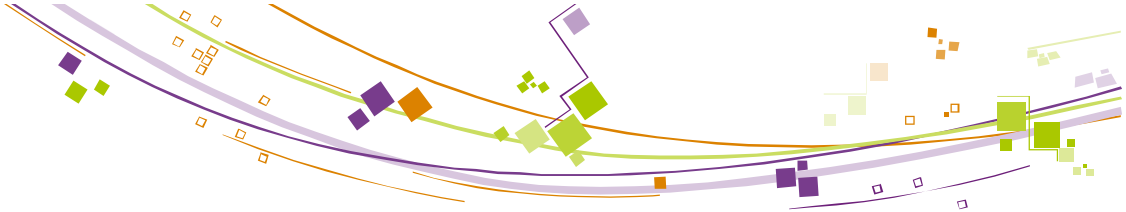
STL : <functional>

- bind1st Return function object with first parameter binded
- bind2nd Return function object with second parameter binded
- ptr_fun Convert function pointer to function object
- mem_fun Convert member function to function object (pointer version)

```
#include <iostream>
#include <functional>
#include <vector>
#include <algorithm>
#include <string>
using namespace std;

int main () {
  vector <string*> numbers;

  // populate vector of pointers:
  numbers.push_back ( new string ("one") );
  numbers.push_back ( new string ("two") );
  numbers.push_back ( new string ("three") );
  numbers.push_back ( new string ("four") );
  numbers.push_back ( new string ("five") );

  vector <int> lengths ( numbers.size() );

  transform (numbers.begin(), numbers.end(), lengths.begin(),
             mem_fun(&string::length));

  for (int i=0; i<5; i++)
    cout << *numbers[i] << " has " << lengths[i] << " letters.\n";
  return 0;
}
```

```
include <iostream>
#include <utility>
#include <functional>
#include <algorithm>
using namespace std;

int main () {
  pair<int*,int*> ptiter;
  int foo[]={10,20,30,40,50};
  int bar[]={10,20,40,80,160};
  ptiter=mismatch ( foo, foo+5, bar, equal_to<int>() );
  cout << "First mistmatching pair is: " << *ptiter.first;
  cout << " and " << *ptiter.second << endl;
  return 0;
}
```

```
#include <iostream>
#include <functional>
#include <algorithm>
using namespace std;

int main () {
  int numbers[] = {10,-20,-30,40,-50};
  int cx;
  cx = count_if ( numbers, numbers+5, bind2nd(less<int>(),0) );
  cout << "There are " << cx << " negative elements.\n";
  return 0;
}
```



- Non standard, mais portable et de grande qualité; apologie du **Template**
- « Antichambre » des TRx du C++ (10 libs dans TR1)
- Nombreuses bibliothèques (divers degrés de maturité)

String and text processing Conversion, Format, Iostreams, Lexical Cast, Locale, Regex, Spirit, String Algo, Tokenizer, Wave, Xpressive Containers Array, Bimap, Circular Buffer, Container, Dynamic Bitset, GIL, Graph, ICL, Intrusive, Multi-Array, Multi-Index, Pointer Container, Property Map, Property Tree, Unordered, Variant Iterators GIL, Graph, Iterators, Operators, Tokenizer Algorithms Algorithm, Foreach, Geometry, GIL, Graph, Min-Max, Range, String Algo, Utility Function objects and higher-order programming Bind, Function, Functional, Functional/Factory, Functional/Forward, Functional/Hash, Functional/Overloaded Function, Lambda, Local Function, Member Function, Phoenix, Ref, Result Of, Signals, Signals2, Utility Generic Programming Call Traits, Concept Check, Enable If, Function Types, GIL, In Place Factory, Typed In Place Factory, Operators, Property Map, Static Assert, Type Traits Template Metaprogramming Function Types, Fusion, MPL, Proto, Static Assert, Type Traits Preprocessor Metaprogramming Identity Type, Preprocessor Concurrent Programming Asio, Interprocess, MPI, Thread Math and numerics Accumulators, Geometry, Integer, Interval, Math, Math Common Factor, Math Octonion, Math Quaternion, Math/Special Functions, Math/Statistical Distributions, Multi-Array, Numeric Conversion, Operators, Random, Ratio, Rational, uBLAS Correctness and testing Concept Check, Static Assert, Test Data structures Any, Bimap, Compressed Pair, Container, Fusion, Heap, ICL, Multi-Index, Pointer Container, Property Tree, Tuple, Uuid, Variant Domain Specific Chrono, CRC, Date Time, Units, Uuid Image processing GIL Input/Output Asio, Assign, Format, IO State Savers, Iostreams, Program Options, Serialization Inter-language support Python Language Features Emulation Exception, Foreach, Move, Parameter, Scope Exit, Typeof Memory Pool, Smart Ptr, Utility Parsing Spirit Patterns and Idioms Compressed Pair, Flyweight, Signals, Signals2, Utility Programming Interfaces Function, Parameter State Machines Meta State Machine, Statechart System Chrono, Date Time, Filesystem, System, Thread Miscellaneous Conversion, Lexical Cast, Numeric Conversion, Optional, Polygon, Program Options, Swap, Timer, TR1, Tribool, Utility, Value Initialized Broken compiler workarounds Compatibility, Config

- <boost/foreach.hpp> : ou comment *appliquer une action sur les éléments d'un conteneur*
  (variante de std::for_each).

```
BOOST_FOREACH( char ch, hello_str ) { std::cout << ch; }
BOOST_REVERSE_FOREACH( char ch, hello_str ) { std::cout << ch; }
```

- <boost/smart_ptr.hpp> : « *Smart Pointers* »

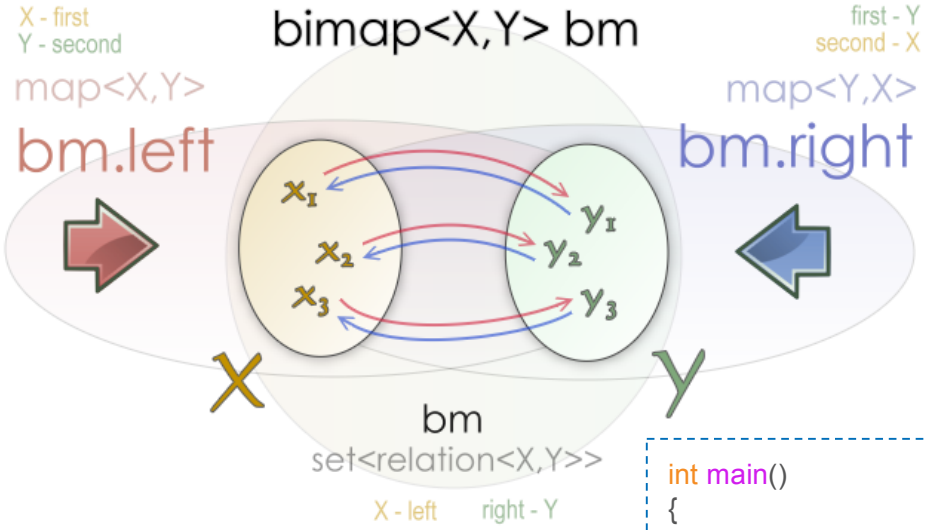
  Délégation de la gestion de la copie et de la destruction des pointeurs
  - <boost/scoped_ptr.hpp> Simple sole ownership of single objects.
  - <boost/scoped_array.hpp> Simple sole ownership of arrays.
  - <boost/shared_ptr.hpp> Object ownership shared among multiple pointers.
  - <boost/make_shared.hpp> Efficient creation of shared_ptr objects.
  - <boost/shared_array.hpp> Array ownership shared among multiple pointers.
  - <boost/weak_ptr.hpp> Non-owning observers of an object owned by shared_ptr.
  - <boost/intrusive_ptr.hpp> Shared ownership of objects with an embedded reference count.

```
{
  boost::shared_ptr<Type> p1 ( new Type(/* args */ )); /* 1 référence */
  boost::weak_ptr<Type> wp1 = p1;
  boost::shared_ptr<Type> p2 = p1;                     /* 2 références */
  p1.reset( new Type(/* … */) );                       /* 1 référence, 1 référence */
  boost::shared_ptr<Type> p3 = wp1.lock();             /* 2 références */
  if (p3) { /* still alive*/ }
  p2.reset();                                          /* 1 référence */
  p3.reset();                                          /* 0 référence : destruction */
}                                                      /* 0 référence : destruction */
```

- <boost/unordered.hpp> : les conteneurs associatifs de la STL en table de hachage
  - Les conteneurs associatifs de la STL utilise des arbres binaires,mais ce n'est pas toujours le plus optimal…
  - Résurrection de quelques anciennes extensions non standards

```
namespace boost {
  template < class Key, class Hash = boost::hash<Key>, class Pred = std::equal_to<Key>,
             class Alloc = std::allocator<Key> > class unordered_set;
  template< class Key, class Hash = boost::hash<Key>, class Pred = std::equal_to<Key>,
            class Alloc = std::allocator<Key> > class unordered_multiset;
  template < class Key, class Mapped, class Hash = boost::hash<Key>, class Pred = std::equal_to<Key>,
             class Alloc = std::allocator<Key> > class unordered_map;
  template< class Key, class Mapped, class Hash = boost::hash<Key>, class Pred = std::equal_to<Key>,
            class Alloc = std::allocator<Key> > class unordered_multimap;
}
```

- <boost/bimap.hpp> : un conteneur associatif bi-directionnel
  - Les deux ensembles de clés sont customisables avec les différente autres conteneurs




```
#include <string>
#include <iostream>
#include <boost/bimap.hpp>

template< class MapType >
void print_map(const MapType & map, const std::string & separator,
                    std::ostream & os )
{
  typedef typename MapType::const_reverse_iterator iterator;
  for( iterator i = map.rbegin(), iend = map.rend(); i != iend; ++i )
    os << i->first << separator << i->second << std::endl;
}
```

```
int main()
{
  typedef boost::bimap< int, std::string> results_bimap;
  typedef results_bimap::value_type score;

  results_bimap results; // 2012-08-04 at 11:31
  results.insert( score(22, "UK"    ) );
  results.insert( score(43, "USA"   ) );
  results.insert( score(42, "China" ) );
  results.insert( score(19, "France") );

  results_bimap::right_map right_results = results.right;
  std::cout << "France has " << right_results.at("France") << " medals¥n";
  std::cout << "Countries names ordered by their number of medals:¥n";
  // works like a std::map< std::string, int >
  print_map( results.left, " medals for ", std::cout );
  return 0;
}
```

<boost/MPI.h> et <boost/serialization> : duo gagnant pour MPI & C++

```cpp
#include <iostream>
#include <string>
#include <set>
#include <boost/mpi.hpp>
#include <boost/shared_ptr.hpp>
#include <boost/foreach.hpp>
#include <boost/lexical_cast.hpp>
#include <boost/serialization/string.hpp>
#include <boost/serialization/set.hpp>
namespace mpi = boost::mpi;

struct PrimData {
  PrimData() { }
  PrimData(float val) { this->val = val; }
  float val;
};

namespace boost { namespace serialization {
template<class Archive>
void serialize(Archive & ar, PrimData & g, const unsigned int version)
{ ar & g.val; }
} }

class Data {
private:
  friend class boost::serialization::access;
  template<class Archive>
  void serialize(Archive & ar, const unsigned int version)
    { ar & m_data1; ar & m_data2; ar & m_data3; }

  int m_data1;
  std::string m_data2;
  std::set<boost::shared_ptr<PrimData> > m_data3;
public:
  Data() : m_data1(0) { }
  Data(int val) {
    m_data1 = val;
    m_data2 = boost::lexical_cast<std::string>(val);
    for(int i=0;i<val/10;++i) m_data3.insert(new PrimData(val-i));
  }
  friend std::ostream & operator<<(std::ostream & o, const Data & d) {
    o << "d1=" << d.m_data1 << " d2=¥"" << d.m_data2 << "¥" d3=( ";
    BOOST_FOREACH(const PrimData * i, d.m_data3) { o << i->val << " "; }
    return o << ")";
  }
};

int main(int argc, char* argv[])
{
  mpi::environment env(argc, argv);
  mpi::communicator world;
  int tag1 = 0, tag2 = 1;

  if (world.rank() == 0) {
    mpi::request reqs[2];
    Data data_in, data_out(42);
    reqs[0] = world.isend(1, tag1, data_out);
    reqs[1] = world.irecv(1, tag2, data_in);
    mpi::wait_all(reqs, reqs + 2);
    std::cout << "Proc" << world.rank() << " receives : " << data_in << "¥n";
  } else {
    mpi::request reqs[2];
    Data data_in, data_out(69);
    reqs[0] = world.isend(0, tag2, data_out);
    reqs[1] = world.irecv(0, tag1, data_in);
    mpi::wait_all(reqs, reqs + 2);
    std::cout << "Proc" << world.rank() << " receives : " << data_in << "¥n";
  }
}
```

- <boost/thread.hpp> : des threads utilisateur portable
- <boost/bind.hpp> et <boost/function.hpp> : pour jouer avec le concept « fonction »

```
#include <iostream>
#include <cmath>
#include <boost/thread.hpp>
#include <boost/date_time.hpp>
#include <boost/bind.hpp>
#include <boost/function.hpp>

double f(double x, double y) { return sin(x) / y; } // une fonction

class Test1 {
public:  Test1(boost::function<double(double)> f) : m_f(f) { }
public: void compute();
private: boost::function<double(double)> m_f;
};

class Test2 {
public:  Test2(int n) : m_n(n) { }
public:  void compute();
private: int m_n;
};

int main(int argc, char* argv[])
{
  Test1 myTest1(boost::bind(f,_1,_1));
  boost::thread myThread1(boost::bind(&Test1::compute, myTest1));
  Test2 myTest2(3);
  boost::thread myThread2(boost::bind(&Test2::compute, myTest2));
  myThread2.join();
  myThread1.interrupt();
  myThread1.join();
  return 0;
}
```

```
boost::mutex myMutex; // global mutex

void Test1::compute() {
  boost::this_thread::disable_interruption di; // no interruption
  boost::posix_time::millisec sleep_time(10);
  for(unsigned int i = 1; true ; ++i)
    {
      boost::mutex::scoped_lock lock_now( myMutex);
      boost::this_thread::sleep(sleep_time); // cannot be interrupted
      double temp = m_f(i);
      std::cout << "¥r" << std::setw(20) << i << std::setw(20) << temp;
      std::cout.flush();
      if (boost::this_thread::interruption_requested()) {
                    std::cout << "¥nBye bye¥n";
                    return;
      }
    }
}

void Test2::compute() {
  boost::posix_time::millisec sleep_time(1000);
  for(int i=0;i<m_n;++i) {
    myMutex.lock();
    std::cout << "¥n" << "Remaining : " << m_n - i << std::endl;
    myMutex.unlock();
    boost::this_thread::sleep(sleep_time);
  }
  myMutex.lock();
  std::cout << "¥n" << "TIMEOUT" << std::endl;
  myMutex.unlock();
}
```

<boost/multi-index>

quand un type devient une base de données

```
#include <iostream>
#include <string>
#include <boost/multi_index_container.hpp>
#include <boost/multi_index/ordered_index.hpp>
#include <boost/multi_index/identity.hpp>
#include <boost/multi_index/member.hpp>
#include <boost/multi_index/mem_fun.hpp>

using boost::multi_index_container;
using namespace boost::multi_index;
using std::string;

struct Employee
{
  Employee(int id_,string name_,int age_):id(id_),m_name(name_),age(age_){}
  string name() const { return m_name; }
  friend std::ostream& operator<<(std::ostream& os,const Employee& e)
  { return os<<e.id<<" "<<e.m_name<<" "<<e.age<<std::endl; }
public: int id, age;
private: string m_name;
};

template<typename Tag,typename MultiIndexContainer>
void print_out_by(const MultiIndexContainer& s)
{
  /* obtain a reference to the index tagged by Tag */
  const typename boost::multi_index::index<MultiIndexContainer,Tag>::type&
    i= get<Tag>(s);
  typedef typename MultiIndexContainer::value_type value_type;

  /* dump the elements of the index to cout */
  std::cout << "by " << Tag::tag_name << std::endl;
  std::copy(i.begin(),i.end(),std::ostream_iterator<value_type>(std::cout));
  std::cout << std::endl;
}
```

```
/* tags for accessing the corresponding indices of Employee_set */
struct Id   { static const char * tag_name; }; const char *
   Id::tag_name = "Id";
struct Name { static const char * tag_name; }; const char *
   Name::tag_name = "Name";
struct Age  { static const char * tag_name; }; const char *
   Age::tag_name = "Age";

// define a multiply indexed set with indices by id and name
typedef multi_index_container<
  Employee,
  indexed_by<
    ordered_unique< tag<Id>,
BOOST_MULTI_INDEX_MEMBER(Employee,int,id)>,
    ordered_non_unique< tag<Name>,
BOOST_MULTI_INDEX_CONST_MEM_FUN(Employee,string,name)>,
    ordered_non_unique< tag<Age>,
BOOST_MULTI_INDEX_MEMBER(Employee,int,age)> >
> EmployeeSet;

int main()
{
  EmployeeSet es;

  es.insert(Employee(0,"Joe",31));
  es.insert(Employee(1,"Robert",27));
  es.insert(Employee(2,"John",40));

  /* next insertion will fail, as there is an Employee with the same ID */
  es.insert(Employee(2,"Aristotle",2387));
  es.insert(Employee(3,"Albert",20));
  es.insert(Employee(4,"John",57));

  /* list the employees sorted by ID, name and age */
  print_out_by<Id>(es);
  print_out_by<Name>(es);
  print_out_by<Age>(es);
  return 0;
}
```

Le C++ : programmation générique

- Utilisation avancée des templates
  - Metaprogramming
    - Ecriture de code à la compilation
  - Traits
    - Classification de types
  - Static polymorphism
    - Le polymorphisme dès la compilation
    - Un contrôle des templates

Le C++ : programmation générique

- La spécialisation, pièce maîtresse de la généricité ?

Code commenté

```
/* Déclarations et spécialisations simples */
template<typename T> struct A          { enum { value = sizeof(T) }; };
template<>           struct A<void*>{ enum { value = 0        }; };

template<int i> int f   (int x)   { return x+i; }
template<>      int f<0>(int x) { return x; }

// Spécialisation
template<typename U, typename V> struct B                { enum { value = 0 }; };
template<typename W>             struct B<W,double> { enum { value = 2 }; };
template<typename W>             struct B<W,W>       { enum { value = 4 }; };
```

```
/* Quelques instanciations */
int main() {
  std::cout << "A<int>                    : " << A<int>::value << std::endl;
  std::cout << "A<void*>                  : " << A<void*>::value << std::endl;
  std::cout << "typeid(A<int>).name()  : " << typeid(A<int>).name() << std::endl;

  std::cout << "f<1>(1)                     : " << f<1>(1) << std::endl;
  std::cout << "f<0>(1)                     : " << f<0>(1) << std::endl;

  std::cout << "B<int,double>          : " << B<int,double>::value << std::endl;
  std::cout << "B<int,int>             : " << B<int,int>::value << std::endl;
  /* ... */
```

Le C++ : programmation générique

- Le *template* metaprogramming : utilisation du langage C++ comme un langage dans le langage.

Code commenté

```
/* Version « standard » de la factorielle */
int factorial(int n)
{
  if (n == 0) return 1;
  else return n * factorial(n-1);
}

/* Metaprogramming */
template<int N> struct Factorial { enum { value = N * Factorial<N-1>::value }; };
template<> struct Factorial<0> { enum { value = 1 }; };
```

```
std::cout << "factorial(6)          : " << factorial(6) << std::endl;
std::cout << "Factorial<6>          : " << Factorial<6>::value << std::endl;
```

- Les mécanismes *Template* sont résolus à la compilation
- A l'exécution le code est déjà développé et optimisé

Le C++ : programmation générique

Code commenté

```
#include <iostream>
#include <cstring> // pour memcpy

struct TrueType { };
struct FalseType { };

// POD Signifie Plain Old Data, ie type de base qui peuvent être correctement recopiés sans l'utilisation d'un constructeur par copie
template<typename T> struct DataTraits        { typedef FalseType IsPODType; };
template<>           struct DataTraits<bool>  { typedef TrueType  IsPODType; };
template<>           struct DataTraits<int>   { typedef TrueType  IsPODType; };
template<>           struct DataTraits<long>  { typedef TrueType  IsPODType; };
template<>           struct DataTraits<double> { typedef TrueType  IsPODType; }; /* liste non complète ... */

template<typename T, int N>
class TinyVector {
  typedef typename DataTraits<T>::IsPODType IsPODType;
public:
  TinyVector() { }
  TinyVector(const TinyVector<T,N> & v) { internal_copy(v,IsPODType()); }
  const TinyVector & operator=(const TinyVector<T,N> & v) { internal_copy(v,IsPODType()); }
private:
  void internal_copy(const TinyVector<T,N> & v, TrueType is_pod_type) {
    std::cout << "Use fast memory copy" << std::endl;
    ::memcpy(m_data,v.m_data,N*sizeof(T));
  }
  void internal_copy(const TinyVector<T,N> & v, FalseType is_pod_type) {
    std::cout << "Use copy constructor" << std::endl;
    for(int i=0;i<N;++i) m_data[i] = v.m_data[i];
  }
private:
  T m_data[N];
};
```

```
struct Truc { };

int main()
{
  std::cout << "Building TinyVector<int,4> :¥n";
  TinyVector<int,4>  v1; v1 = TinyVector<int,4>();
  std::cout << "Building TinyVector<Truc,4>:¥n";
  TinyVector<Truc,4> v2; v2 = TinyVector<Truc,4>();
  return 0;
}
```

Le C++ : programmation générique

Code commenté

```
/* Static polymorphism */
template <class DerivedT>
struct Base
{
  void interface()
  {
    // ...
    static_cast<DerivedT*>(this)->implementation();
    // ...
  }
};

// Curiously Recurring Template Pattern (CRTP)
struct Derived : public Base<Derived>
{
  void implementation()
  {
    std:: cout << "This is " << typeid(*this).name() << std::endl;
  }
};
```

```
template<typename T>
void foo(Base<T> & b) { b.interface(); } // static dispatch
```

```
Derived d;
foo(d);
```

Le C++ : programmation générique

- Substitution failure is not an error (SFINAE)
  - Un exemple simple

```
struct Test
{
    typedef int type;
};

template <typename T> void f(typename T::type) {} // definition #1

template <typename T> void f(T) {}                // definition #2

int main()
{
    f<Test>(10); // appelle #1
    f<int>(10);   // appelle #2 sans erreur grâce à SFINAE
}
```

Le C++ : programmation générique

- Substitution failure is not an error (SFINAE)

```
#include <iostream>

template <typename T>
struct has_typedef_type
{
   // yes et no sont de tailles différentes
   typedef char yes[1]; // sizeof(yes) == 1
   typedef char no[2];   // sizeof(no) == 2

   template <typename C> static yes& test(typename C::type*);
   template <typename> static no& test(...);

   // if the sizeof the result of calling test<T>(0) is equal to the sizeof(yes),
   // the first overload worked and T has a nested type named type.
   static const bool value = sizeof(test<T>(0)) == sizeof(yes);
};

struct foo { typedef float type; };

int main()
{
   std::cout << std::boolalpha;
   std::cout << has_typedef_type<int>::value << std::endl;
   std::cout << has_typedef_type<foo>::value << std::endl;
}
```

Le C++ : programmation générique

- Une algèbre de fonctions avec différentiation automatique

  Exemple de f = $(2x+1)^4 - \dfrac{(x-1)^3}{1+x+\left(\frac{x}{2}\right)^3}$

	Interfaces virtuelles 	Template metaprogramming
Taille du code	239 lignes (8568 chars)	438 lignes (17619 chars) dont 150 d'optimisation statique
Dev from scratch	30mn	4h
Compilation GCC -O3	1.5s	1s
Ecriture C++	((2*x+1)^4)-((x-1)^3)/(1+x+((x/2.)^3))	pow<4>(2*x+1)-pow<3>(x-1)/(1+x+pow<3>(x/2.))
Instantiation de d(f)	(((4) * ((((2) * (x)) + (1)) ^3)) * ((((0) * (x)) + ((2) * (1))) + (0))) + (((0) * ((((x) + (-1)) ^3) * (inv(((x) + (1)) + (((0.5) * (x)) ^3))))) + ((-1) * (((((3) * (((x) + (-1)) ^2)) * ((1) + (0))) * (inv(((x) + (1)) + (((0.5) * (x)) ^3)))) + ((((x) + (-1)) ^3) * (((-1) * (((1) + (0)) + (((3) * (((0.5) * (x)) ^2)) * (((0) * (x)) + ((0.5) * (1)))))) * (inv((((x) + (1)) + (((0.5) * (x)) ^3)) ^2)))))))	(((((x) * (2)) + (1)) ^3) * (8)) + ((((x) + (-1)) ^3) * (((1) + ((((x) * (0.5)) ^2) * (1.5))) * ((inv((((x) + (1)) + (((x) * (0.5)) ^3)) * (((x) + (1)) + (((x) * (0.5)) ^3)))) * (1)))) + ((((x) + (-1)) ^2) * ((inv(((x) + (1)) + (((x) * (0.5)) ^3))) * (-3)))) Version avec optimisation statique et on peut faire encore mieux
Empreinte mémoire d(f)	2592 bytes	No static opt: 336 bytes Static opt: 208 bytes
2^22 évaluations de f et d(f) (BO3)	8.28s NB: L'évaluation par fonctions *style C* écrites à la main de l'expression de f et d(f) prend 2.48s	No static opt: 2.83s Static opt: 2.11s