Programmation Système Répartie et Concurrente
Master 1 Informatique – MU4I400

Cours 3 : Lib standard du C++

Yann Thierry-Mieg
Yann.Thierry-Mieg@lip6.fr

Plan

On a vu au cours précédent

- La syntaxe de base, les variables, les références
- Les classes, attributs, fonctions membres (méthodes), opérateurs
- Gestion de la mémoire dynamique, new/delete

Aujourd'hui : La bibliothèque standard std::

- Templates, Généricité, **auto**
- Headers du C
- Entrées / sorties
- Utilitaires, Chrono
- Conteneurs, Itérateurs
- Algorithmes, Lambdas

Références : cppreference.com , cplusplus.com , docs.microsoft.com

Généricité, Auto

Fonctions Templates

- Deux fonctions surchargées (même nom) qui diffèrent par le typage
  - Même corps/code
- Version générique qui évite la duplication ?

```
int sum (int a, int b)
{   return a+b; }

double sum (double a, double b)
{   return a+b; }

int main ()
{
  cout << sum (10,20) << '\n';
  cout << sum (1.0,1.5) << '\n';
  return 0;
}
```

Fonctions Templates (2)

```
template<typename T>
T sum (T a, T b)
{   return a+b;  }
```

- On préfixe la déclaration par la déclaration des paramètres génériques
  - Introduit ici que T est un type
- On peut utiliser T là ou un nom de type ou de classe est attendu (signatures…)
- Le code n'est pas compilé tel quel, il doit être visible du client (e.g. dans un .h)
- Le code est instancié à l'invocation
  - sum (10,20) **-> instantiation T = int**
  - sum (1.0,2.0) **-> instantiation T = float**
  - sum("a","b") **-> erreur + pour const char ***
  - sum<string> ("a","b") **-> T = string (forcé)**
- On substitue à T un type effectif et on compile le code => nécessite une variante de : T operator+(T a,T b)

Classes Template

- Classe dont le type des attributs et/ou la signature des méthodes est générique
- Le corps des déclarations doit être présent dans le .h
- Un ou plusieurs argument génériques possibles

```
template <typename T, typename U>
class pair {
  public:
    T first; U second;
    mypair (T first, U second)
                    : first(first),second(second) {}
          // ... operateurs, accesseurs
};
```

```
template <typename T>
class mypair {
    T values [2];
  public:
    mypair (T first, T second)
    {       values[0]=first;
            values[1]=second;  }
};
```

Templates : concepts avancés

- Utilisation d'un type générique dans une signature de template

```
template < template <typename> class Cont, typename T >
const T & findFirst (const Cont<T> & c) { return c[0]; }
```

- Spécialisation partielle de template
  - vector<bool> => implémentation BitSet
- Paramètres génériques par défaut
  - E.g. Allocateur par défaut sur les conteneurs standard
- D'autres usages sont possibles en méta programmation
  - S'appuie sur l'instanciation de génériques pour générer du code
- Beaucoup de bibliothèques C++ sont très génériques
  - Cf Boost, presque entièrement header purs.
- Mon conseil personnel :
  - Beaucoup des arguments pour les codes génériques relèvent de l'optimisation. A EVITER DANS UN PREMIER TEMPS

auto

auto : inférence de type

- C++11 apporte une notion nouvelle d'inférence de type
  - Mot clé : auto
- Permet de typer :
  - Les variables dont le type est déduit de l'initialisation
  - Les paramètres de fonction anonymes (lambdas)
  - Le type de retour d'une fonction (déduit du typage de return)
- Très confortable, surtout pour les types dérivés ou à rallonge, et dans le code des fonctions génériques
  - Des exemples dans la suite du cours
- Les règles régissant « auto » sont en évolution (e.g. plus d'inférences possibles en C++17)

auto : exemples

- Attention différences entre
  - auto et auto &
- L'inférence fonctionne aussi avec les classes
- Remplace des expressions classiques

```
auto d = 5.0; // double
auto i = 1 + 2; // int

int add(int x, int y)
{ return x + y; }

int main()
{   auto sum = add(5, 6); // int
    return 0;
}
```

```
std::vector<int>::const_iterator it = vec.end(); //AVANT
auto it = vec.end();  // C++11
```

auto, auto &, const auto &

- La sémantique de auto = type simple
  - Sémantique par copie donc
- Bien souvent, une référence voire une const ref suffit
  - auto & var = rhs
  - const auto & var = rhs
  - Evitent la copie si la partie droite est un objet
- L'inférence de type
  - Facilite les mises à jour
  - Rend parfois plus difficile la lecture du code
  - À utiliser avec modération

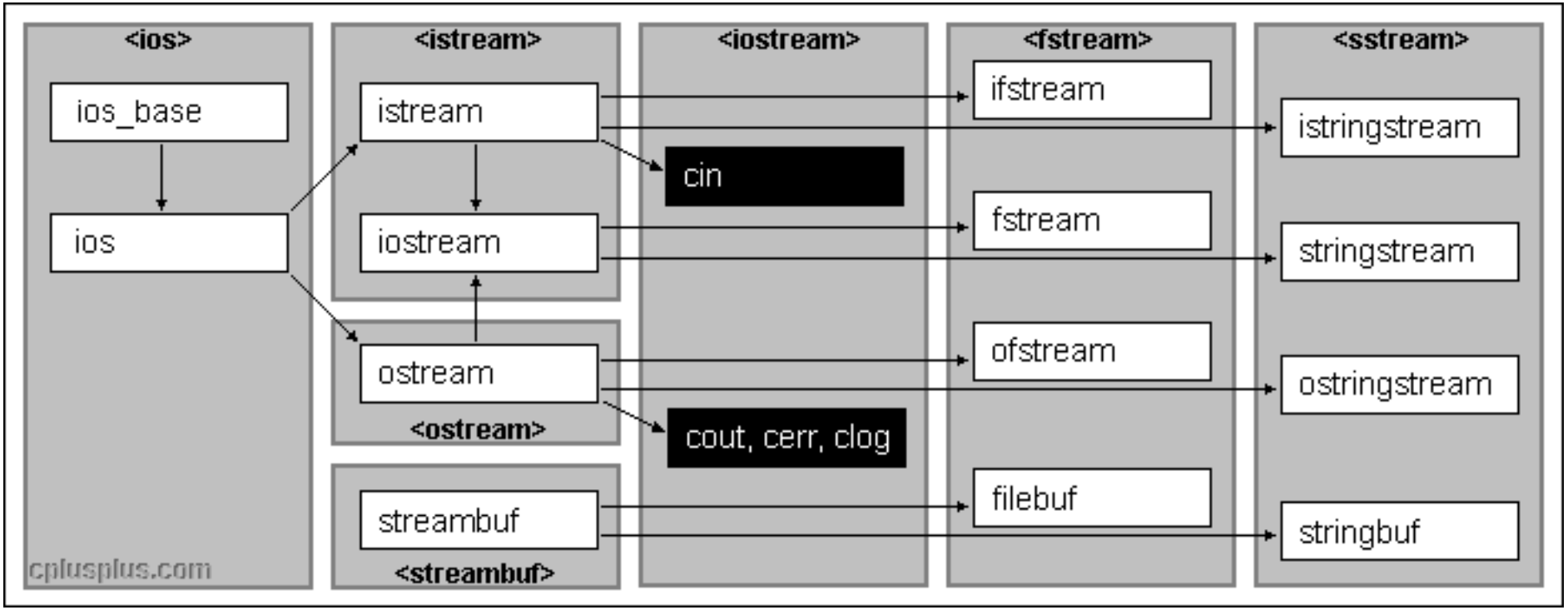
La lib standard du C++

Accès à la lib standard du C

- Inclut la lib standard du C (headers <cxxx>)
  - <cassert> : assertions
  - <cmath> : sqrt, pow, sin, cos…
  - <cstring> : strcmp, strcat, strcpy…
  - <cstdio> : printf, scanf, FILE, fopen, fclose…
  - <cstdlib> : atoi, rand (voir aussi <random>), size_t, malloc/free, exit
  - Ainsi que pas mal d'autres
- Les fonctions sont placées dans le namespace par défaut
  - Accès qualifié avec **::strcpy(dest,src)**
- On utilise : extern "C" { /* déclaration */ } pour importer des déclarations C en C++.

Entrées sorties

- <iosfwd> juste les prédéclarations (utilisation dans .h)
- <iostream> flux d'entrée sorties standards
- <fstream> flux sur fichiers
- <sstream> flux en mémoire



<utility>

- Classe pair<F,S>
  - Deux attributs : first et second publics
    - Notion de Plain Old Data, POD proche d'un struct
    - Mais définitions correctes de ==, <, hash, …
  - Attributs typés arbitrairement
  - Utilisé dans certaines API standard (map en particulier)
  - Construction générique : std::make_pair(a,b)
    - Évite de citer les types
  - Cf. aussi <tuple> de taille plus que deux
- rel_ops (attention il faut « using namespace std::rel_ops »).

```
namespace rel_ops {
  template <class T> bool operator!= (const T& x, const T& y) { return !(x==y); }
  template <class T> bool operator>  (const T& x, const T& y) { return y<x; }
  template <class T> bool operator<= (const T& x, const T& y) { return !(y<x); }
  template <class T> bool operator>= (const T& x, const T& y) { return !(x<y); }
}
```

<string>

- Classe string
  - Chaîne de caractères modifiable, accès comme un tableau de caractères avec []
  - + pour concaténer des string, += append
  - Compatible avec le C :
    - const char * c_str() const;
    - Construction implicite depuis const char *
- Conversions
  - stoi, stol, stof… convertit vers un type numérique (parse)
  - **string std::to_string (T val)** : T étant int, float, long… convertir les types numériques vers string
- Attention, + entre un entier et une string se décale dans la string (différent de Java, pas de toString implicite)

<regex>

- Expressions régulières
  - Une arme importante pour traiter des données « simples »
  - Ne remplace pas l'usage d'un « vrai » parser/grammaire pour les données plus complexes
  - Extraction de données depuis des chaînes de caractères
    - Engendre un petit automate, efficace et « juste »
  - Standard indépendant des langages, issu de sed, awk, perl…
- A priori, syntaxe : std::ECMAScript (cf web)

```
// syntax raw :  R"(contenu)" évite le double \\
std::regex greet(R"(\w+!)"); // un mot suivi d'un !
std::string s = "Hello World!";
std::string s2 = std::regex_replace(s,greet,"Master");
std::cout << s << std::endl; // Hello World!
std::cout << s2 << std::endl; // Hello Master
```

<chrono>

- Mesure du temps et des intervalles
  - **duration** définit la grandeur : hours, minutes, milliseconds, …
- On utilise une **steady_clock** (progression monotone stricte) ou une **system_clock** (horloge système)

```
// ou auto
std::chrono::steady_clock::time_point start =
                    std::chrono::steady_clock::now();
//          code dont on veut mesurer le temps d'exécution
auto end = std::chrono::steady_clock::now();
std::cout << "Duration "
          << std::chrono::duration_cast<std::chrono::milliseconds>
                (end - start).count()
          << "ms.\n";
```

Conteneurs, Itérateurs

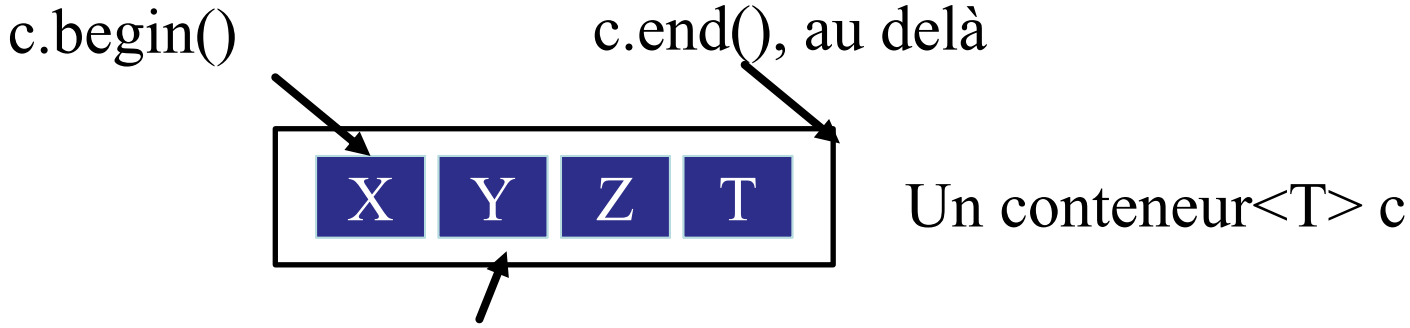
Les conteneurs

- Essentiels pour toute mise en place d'algorithmique
  - Nécessaire d'acquérir une familiarité avec l'API
  - Interaction avec <algorithm>
  - Offre les structures de données classiques sous forme générique
- Les conteneurs<T> :
  - vector : contigü, dynamique
  - list : double chaînée
  - deque : double edge queue, liste de blocs mémoire
  - set : arbre R/N équilibré
  - unordered_set : hash set
  - forward_list : simple chaînée, à la C
  - array : taille fixée à la construction
  - stack, queue, … des décorateurs

Eléments communs des conteneurs

- Modèle générique à au moins deux paramètres
  - T : le type des éléments contenus
  - (défaut OK) Un allocateur gérant le new
  - (défaut OK) Une fonction de comparaison, de hash si nécessaire
- Accès via des itérateurs
  - begin et end rendent des itérateurs au début et au-delà de la fin
  - Versions const et non const
  - Construction par copie à partir d'itérateurs
- MAIS
  - Pas d'API complètement unifiée : pas de size sur forward_list, pas de push_front sur vector
  - Les opérations efficaces pour chaque structure sont là
  - https://en.cppreference.com/w/cpp/container

Design Pattern Itérateur



- Homogénéise l'accès à un aggrégat de données
- Le conteneur est responsable de fournir les itérateurs
  - begin : désigne le premier élément, ou égal à end si vide
  - end : au-delà du dernier élément, pas de contenu associé
- L'itérateur définit
  - operator*, operator-> : comme si c'était un pointeur
  - operator++ : incrément/décalage
  - operator!= : pour se comparer à end

Logique de l'API Itérateur C++

- L'itération doit ressembler le plus possible à une boucle classique utilisant un pointeur sur un tableau C

```
int tab [SIZE];
// init
for (int * pi=tab,_end=tab+SIZE; pi != _end ; ++pi) {
        int i = *pi; // ou : int & i
        // utiliser i, affectation ok si par référence
}
```

- Expansion foreach C++11

```
vector<int> vec;
// boucle "foreach"
for (int i : vec) {
        // body
}
```

```
{ for(auto it = vec.begin(),_end = vec.end();
                it != _end;
                ++it)
  {     int i = *it;
                // body
  }}
```

Catégories d'itérateurs

category				properties	valid expressions
all categories				*copy-constructible*, *copy-assignable* and *destructible*	`X b(a);` `b = a;`
				Can be incremented	`++a` `a++`
Random Access	Bidirectional	Forward	Input	Supports equality/inequality comparisons	`a == b` `a != b`
				Can be dereferenced as an *rvalue*	`*a` `a->m`
			Output	Can be dereferenced as an *lvalue* (only for *mutable iterator types*)	`*a = t` `*a++ = t`
				default-constructible	`X a;` `X()`
				Multi-pass: neither dereferencing nor incrementing affects dereferenceability	`{ b=a; *a++; *b; }`

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

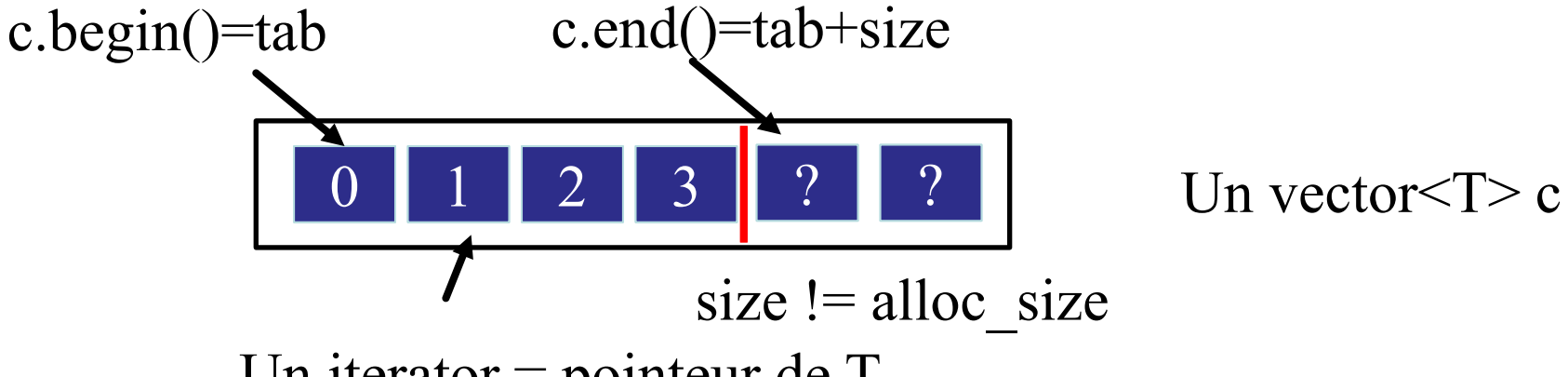
				Supports arithmetic operators + and -	`a + n`
					`n + a`
					`a - n`
					`a - b`

				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]`

<vector>

- Stockage contigu en mémoire, réallocation dynamique, notion de capacité >= size le nombre d'éléments contenus
  - reserve demande une allocation
- Accès indicé rapide : random acess iterator
  - operator[], at (attention à size)
  - Ajout en fin : push_back, pop_back, emplace_back
  - insert => reallocation
  - Copie des objets dans certains cas : T doit être copiable
  - Voir aussi <deque> si insertions en tête



vector<T>

Docs : https://docs.microsoft.com/en-us/cpp/standard-library/vector-class et/ou https://en.cppreference.com/w/cpp/container/vector

at, operator[] Returns a reference to the element at a specified location in the vector.

begin Returns a random-access iterator to the first element in the vector.

end Returns a random-access iterator that points to the end of the vector.

rbegin, rend, cbegin, cend, crbegin, crend variantes **const** et **reversed**

back, front premier et dernier élément

capacity, size, empty : taille allouée et taille actuelle, test à vide

reserve, resize, shrink_to_fit : Gestion de l'espace disponible.

insert, push_back, emplace, emplace_back : ajout d'éléments

erase, pop_back : suppressions d'élément

clear Erases the elements of the vector.

vector: exemple

```
vector<int> v;
v.reserve(10); // size = 0
for (int i=0; i < 10 ; i++) {      v.push_back(i);    }
cout << v[3] << v.at(3) << endl;
auto it = find(v.begin(),v.end(),5);
if ( it != v.end() ) {
   cout << it - v.begin() <<":" << *it << endl;
}
for (int i : v) {
   cout << i << " ";
}
for (int & i : v) {
   i++;
}
cout << endl << *v.begin() << endl;
v.clear(); // size = 0
```

```
33
5:5
0 1 2 3 4 5 6 7 8 9
1
```

<list> <forward_list>

- Structure de liste doublement chaînée <list>
  - Maintient des pointeurs sur les chainons de tête et de queue
  - Chaque chaînon stocke un pointeur vers son prédécesseur et son successeur
    - Bidirectional iterator
  - Itérable dans les deux directions, pas d'accès indicé
  - Itérateur = pointeur sur chaînon cur
    - ++ : cur = cur->next;
    - * : return cur->data;
  - Itérateurs stables, mais coût mémoire relativement élevé
- Structure de liste simplement chainée <forward_list>
  - Forward iterator seulement, pas de size
  - La liste chaînée du C, version « propre » (sans macros)

Itérateurs et const

- On est forcés par le typage des fonctions de fournir deux types d'itérateurs : const_iterator et iterator (modifiant)
  - operator*() : quel type rendu ? T ou T &
- Le corps des deux implémentations sont souvent très similaires
  - return val ; // le sens change selon le typage de la fonction
  - Un peu de duplication est nécessaire
- Les types iterator et const_iterator sont définis *à l'intérieur* de la classe conteneur
  - Pas d'accès privilégié à une instance comme Java
  - On utilise souvent un typedef, pour déléguer la définition du type à un itérateur imbriqué

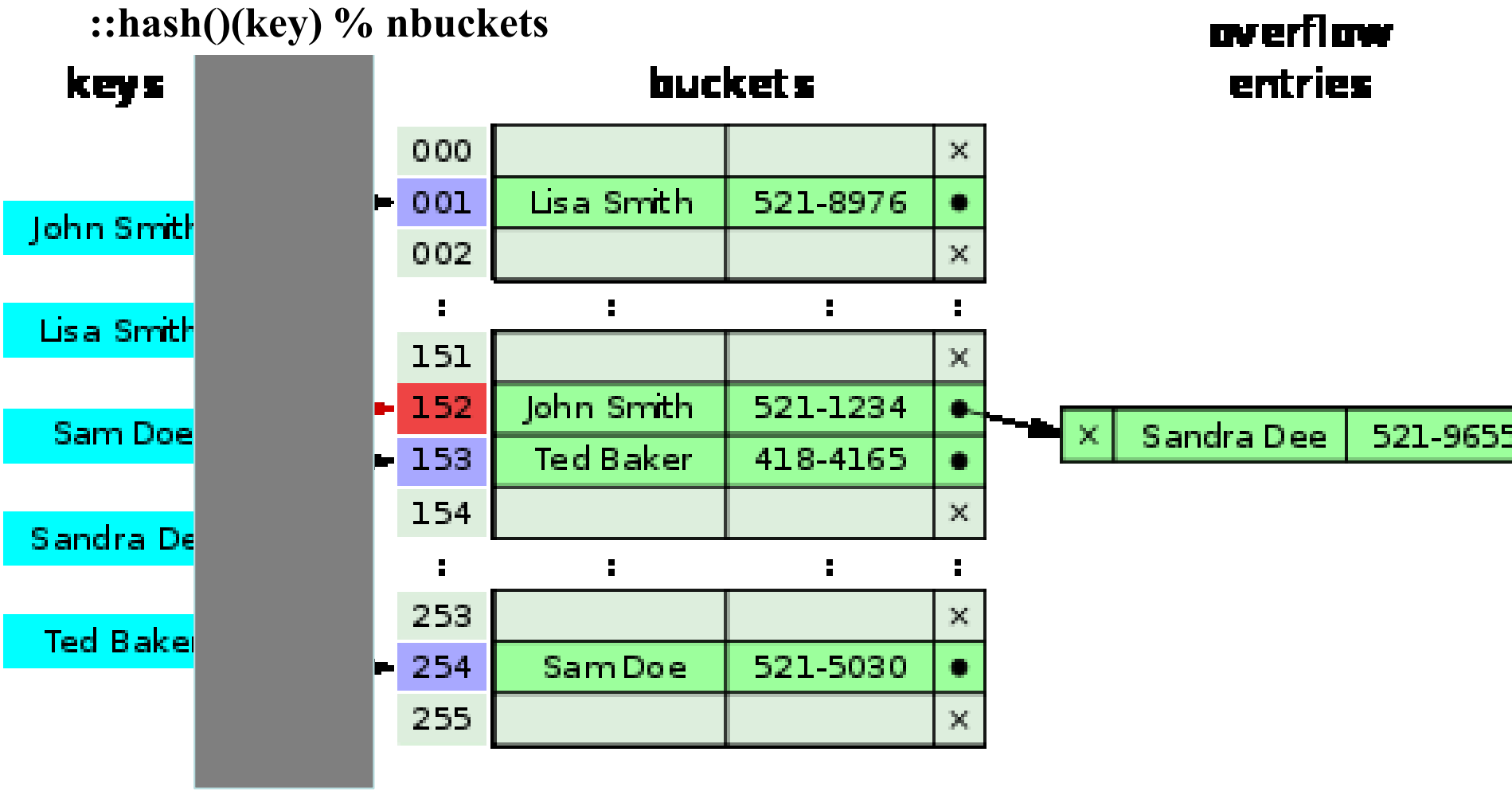
```
typedef vector<int>::const_iterator iterator;
```

Les conteneurs associatifs

- Matérialisent une table associative <map> <unordered_map>
- Soit map<string, int> agenda. On peut voir agenda comme :
  - ✓ Un ensemble de paires clé/valeur :
    - { <toto;0601>, <tata;0602>}
  - ✓ Une fonction; i.e. une application de string dans int telle que :
    - agenda(toto)=0601 et agenda(tata)=0602
  - ✓ Un tableau indicé sur des string :
    - agenda[« toto »] = 0601, agenda[« tata »] = 0602
- Propriété de clé : à une clé donnée ne correspond qu'une valeur
- Bonnes complexités du test d'existence d'une clé, recherche et ajouts.

Table de Hash : principes

- On *hash* la clé => size_t, on cherche modulo le nombre de *buckets* dans la liste à cet indice
- Si le hash est rapide et « bon », O(1) pour trouver une entrée (paire)



Itérateur sur map

- L'itérateur référence une entrée dans la table
  - * rend un pair<const K,V> : first =clé, second=valeur
- Opérations fréquentes de map :
  - iterator find(const K & key)
    - Rend un itérateur correct ou end si pas trouvé O(1)
  - pair<iterator,bool> insert (pair<K,V> kv)
    - Vrai si l'insertion a eu lieu (sinon pas modifié) O(1)
    - L'itérateur pointe l'élément de clé égale à kv.first
  - operator[] :
    - map[« clé »] = valeur
    - Attention, crée l'entrée si elle n'existe pas O(1)

Map : propriétés

<unordered_map>,<unordered_set>

- ✓ Recherche et insertion O(1) => Excellente complexité
- ✓ Itérations dans le désordre en O(capacité)
- ✓ La réindexation coûte cher, réservez la bonne capacité
- ✓ Cf. aussi : google::sparse_hash_set

<map>,<set>

- ✓ Basé sur un arbre binaire de recherche rouge/noir équilibré
- ✓ Recherche et insertion en O(log2(n))
- ✓ Itérations dans l'ordre naturel operator< des clés

Map : exemple

```
unordered_map<int,int> factCache;
int fact (int n) {
 if (n<=2) return n;
 auto it = factCache.find(n);
 if (it == factCache.end()) {
        // cache miss
        int res = fact (n-1) * n;
        factCache[n]=res;
        return res;
 } else {
        // cache hit
        return it->second;
 }
}
```

```
factCache.insert(make_pair(10,3628800));
factCache.insert({5,120});
cout << fact(10) ;
cout << fact(8) ;
```

Memory

<memory>

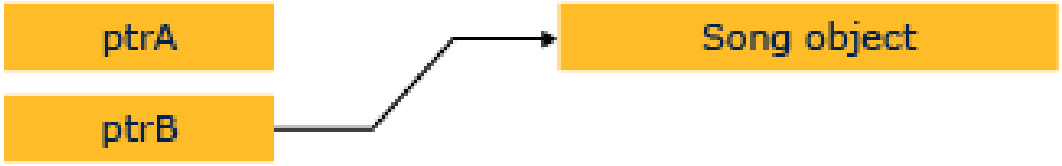
- Pointeurs intelligents : se comportent comme des pointeurs globalement, mais gèrent la mémoire pointée
  - unique_ptr : un bloc de mémoire qui n'est pas partagé, une seule utilisation à la fois (pas d'aliasing)
  - shared_ptr : un bloc de mémoire partagé muni d'un compteur de références, collecté quand il tombe à 0. En général on le veut non modifiable (const).
  - Il y en a quelques autres, éviter auto_ptr (obsolète)
- Evite pas mal de gestion mémoire à la main
  - Cf String constantes du TD 1
  - Pas un mécanisme de GC complet (cycles ?)
- Pour la mémoire dynamique le premier reflexe doit rester <vector>

unique_ptr<T>

- Un pointeur unique vers une zone « détenue »
  - Obtenue avec std::make_unique<T>(objet)
  - Utilisable avec * ou -> comme un pointeur normal
  - Détruit quand le point est détruit
  - Transféré quand le pointeur est affecté (sémantique move)
- Particulièrement utile en présence d'héritage
  - On détient les objets via des pointeurs dans ce cas en général

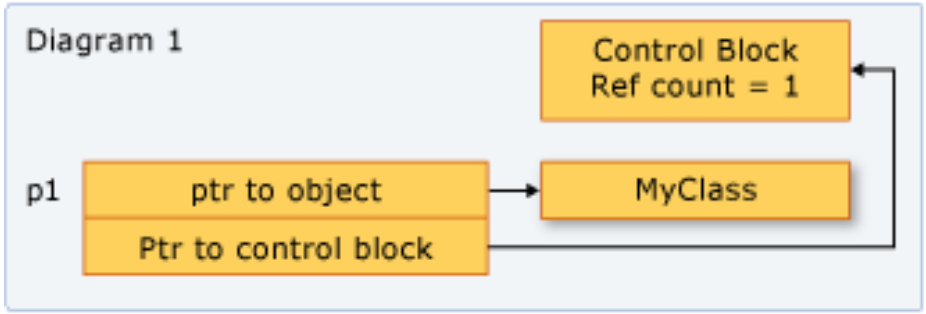
```
auto ptrA = make_unique<Song>(L"Diana Krall", L"The Look of Love");
```

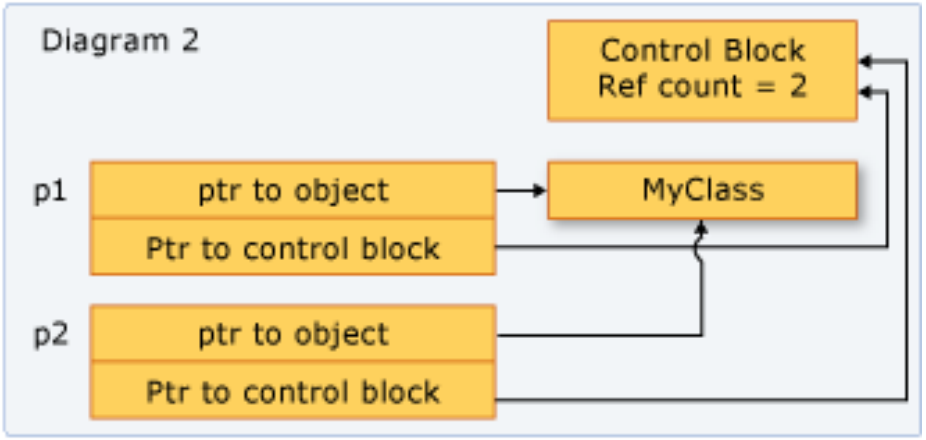

```
auto ptrB = std::move(ptrA);
```



shared_ptr<T>

- Un pointeur partagé + compteur vers une zone détenue
  - Obtenu avec std::make_shared<T>(objet)
  - Copiable, affectable : maintient un compteur à jour
  - En général manipulé en lecture seule
  - L'objet est détruit quand tous les pointeurs sont détruits (ne traite pas les cycles)
  - Aussi appelé smart pointer dans la littérature (pattern proxy GOF)





Algorithm, Function, Lambda

<algorithm>

- Grâce à l'abstraction fournie par les itérateurs, des algorithmes génériques sont fournis
  - Arguments fournis sous la forme de *range* délimités par deux itérateurs begin/end
- Un algorithme est souvent paramétrable
  - sort nécessite une relation d'ordre operator<
  - Configuration en passant un foncteur (classe qui redéfinit operator()), un pointeur de fonction, ou une lambda
- Algorithmes scindés selon l'exigence vis-à-vis des itérateurs
  - Séquentiel : ++, *
    - Deux sous-catégories const/modifiant
  - Tris, actions sur des zones triées (recherche, fusion)
  - Min/max, quelques autres fonctions utiles

Pointeur de Fonction

Rappels du C :

```
int foo(int x)
{     return x; }

int main()
{
    int (*fcnPtr)(int) = foo; // assign fcnPtr to function foo
    (*fcnPtr)(5); // call function foo(5) through fcnPtr.
    fcnPtr(5); // implicit dereference
    return 0;
}
```

Foncteur

- Toute classe qui définit operator() est un foncteur
  - Dualité : Objet + Fonction

```
class Delta {
        int max;
public:
        Delta(int max) :max(max) {};
        int operator() (int val) { return max - val; }
};

int test () {
        Delta delta(42);
        int val = 10;

        int diff = delta(val);
        return diff;
}
```

Lambda

- Définition d'une fonction anonyme
  - *Contexte* visible dans la fonction : [var1,&var2]
    - « Capture clause », par valeur ou par référence
    - [] vide, [=] par copie tout, [&] par référence tout
  - Arguments typés,
    - Comme une fonction normale
    - Type de retour optionnel, notation « trailing » -> type
  - Corps de la fonction, comme un bloc normal.

```
int test2() {
        int max = 42;
        int val = 10;

        int diff = [max](int v) -> int  { return max - v; } (val);
        return diff;
}
```

Lambda (2)

Exemple : fournir un critère de tri à « sort »

Les arguments du lambda peuvent utiliser « auto » (C++14)

```
#include <algorithm>
#include <cmath>

void abssort(float* x, unsigned n) {
    std::sort(x, x + n,
        // Lambda expression begins
        [](float a, float b) {
            return (std::abs(a) < std::abs(b));
        } // end of lambda expression
    );
}
```

Algorithmes Séquentiels

- Lecture/recherche d'élements satisfaisant un prédicat (const)
  - all_of, any_of, none_of :
    - rendent un bool
  - find, find_if, find_if_not, find_end, find_first_of, adjacent_find
    - rendent un itérateur à la position trouvée ou end
    - ite find_if(ite begin, ite end, UnaryPredicate pred)
    - Le prédicat est une fonction : bool (*) (const T &)
    - Ou tout objet disposant de : bool operator() (const T &)
  - count, count_if
    - Compter les élements
  - for_each
    - Appliquer une fonction à tous les éléments

Exemple : find_if, remove_if, erase

```
class Person {
public :
        string name;
        int age;
        Person(const string & name, int age) : name(name), age(age) {}
};

int algosTest () {
  vector<Person> vec;
  vec.emplace_back("toto", 12);
  vec.emplace_back("titi", 42); //...

  auto it = find_if(vec.begin(), vec.end(), [](const auto & p) { return p.age > 20;  });
  if (it != vec.end()) {
    cout << it->name << std::endl;
  }
  auto finzone = std::remove_if(vec.begin(), vec.end(),
                        [](const auto & p) { return p.age >= 20; });
  vec.erase(finzone,vec.end());
}
```

Exemples

```
 vector<int> v; // + ajout contenu
// tri des 5 premiers éléments, en ordre décroissant
 sort(v.begin(), v.begin()+5, [] (int a, int b) { return b > a;} );
// recherche d'un élément de valeur 6 => rend un itérateur
 auto it = find(v.begin(),v.end(),6);
// l'itérateur vaut end ssi. on ne l'a pas trouvé. Sinon *it vaut 6 ici.
 if ( it != v.end() ) { /*...*/ }
// on recherche dans le vecteur un élément qui vaut la moitié de 6 (*it)
 auto it2 = find_if(v.begin(),v.end(),[&it](auto a){ return a*2==*it; });
```

Définition de Prédicats

- Plusieurs approches possibles pour les définir
  - Avec une fonction
    - bool compare (Etu a, Etu b) { return a.note < b.note ;}
  - Avec une classe et son operator()

```
struct EtuComp {
  int mod;
  bool operator() (const Etu & a, const Etu & b) {
          return a.note % mod == b.note % mod ;  }
}
```

  - Avec une lambda expression

```
[] (const Etu & a, const Etu & b) { return a.note < b.note ;}
```

  - Dans le contexte d'une lambda auto fonctionne

```
[] (const auto & a, const auto & b) { return a.note < b.note ;}
```

Algorithmes Séquentiels non const

- Modifications d'éléments (non const)
  - replace, replace_if, remove, remove_if : modifie
    - Attention remove_if s'utilise avec erase
  - shuffle, rotate : change l'ordre
  - swap, swap_range : échanger des cellules
- Certains algorithmes ont besoin en plus de begin/end décrivant la zone de travail, d'un itérateur en écriture pour le résultat
  - copy, copy_if : copie (à travers les conteneurs de types différents)
  - Utiliser : std::back_inserter(container), std::inserter(cont,ite)
- Les tris
  - sort, stable_sort : nécessitent des RandomIterator. On peut leur passer un prédicat : bool (*less) (const T & a, const T&b)

Exemples

```
// un map, associant des entiers à des string
unordered_map<string,int> frequency;
// insertion de paires dans le map
 frequency.insert({"toto",3});frequency.insert({"tata",6}); //...
// un vecteur de paires, pour loger le contenu de la map
 vector<pair<string,int> > entries;
// on alloue la bonne taille tout de suite
 entries.reserve(frequency.size());
// copie du contenu du map dans le vecteur (ordre arbitraire)
// on note le “back_inserter” modifiant
 copy(frequency.begin(),frequency.end(),back_inserter(entries));
// tri des paires par valeur du deuxième élément (fréquence)
 sort(entries.begin(),entries.end(),
    [](const auto & a, const auto &b) { return a.second > b.second; });
 cout << entries.begin()->first << endl; // le mot le plus fréquent !
```