Programmation Avancée et Projet

Prise de notes basée sur le cours de M. Torri

Julien Khamphousone

18 janvier 2018

Table des matières

Première partie

1 Shared-ptr

A utre solution : utiliser make_shared qui crée un shared-ptr en allouant la mémoire. Ex :

```
f(make_shared<int>(42),g());
```

1.1 Fuite de mémoire : les cycles

Exemple :

```
class A
{
  shared_ptr<B> nom_B;
  ...
};
class B {
  shared_ptr<A> nom_A;
  ...
};
shared_ptr<A> a = new A;
shared_ptr<B> b = new B;
count << a.use_count() << b.use_count() << endl;
a->nom_b = b;
count << a.use_count() << endl;
b->nom_a = a;
count << b.use_count() << endl;
a.reset;
b.reset;
```

Affichage :

1 1

2

2

Après reset, les compteurs sont décrémentés de 1 et a et b ne sont plus accessibles ⇒ fuite de mémoire.

1.2 Allocation de mémoire

```
class C;
C * nom_alloc();
void nom_new(C*);
shared_ptr<C> (nom_alloc());
```

Plusieurs problèmes

- Si nom_alloc() fait plus que juste un new ⇒ fuite de mémoire.
- Si nom_alloc() utilise un autre allocateur de mémoire. ⇒ programme plante.

On utilise le 2e argument du constructeur du shared_ptr ;

```
shared_ptr<C> (nom_alloc(), &nom_new);
```

```
// Declaration de nom_new :
void nom_new(void*);
```

1.3 Inconvénients

- Ils ne gèrent pas les cycles
- Ils peuvent être moins performants que les ptr classiques
- Ils consomment plus de mémoire (ref_count)
- Gestion des ref_count peut être couteuse en terme de performance

weak_ptr Ils sont conçus pour résoudre le problème des cycles avec les shared_ptr.
Idée :

- Les shared_ptr : ils gèrent la durée de vie d'un ptr.
- les weak_ptr : avoir accès à la mémoire. Ils ne gèrent pas la durée de vie.

Utlisation de base : On peut créer un weak_ptr à partir d'un shared_ptr ou à partir d'un shared_ptr ou à partir d'un autre weak_ptr ;

Exemple :

```
shared_ptr<int> s(new int(42));
weak_ptr<int> w(s);
```

Pour accéder à l'objet : méthode lock() :

```
void f(weak_ptr<T> w){
  shared_ptr<T> data = w.lock();
  if (data)
  {
    //on travaille sur data
  }
  else
  // le ref_count de data vaut 0, l'objet est detruit.
}
```

Cassage de cycles

Exemple 1 :

```
class Tirroir
{
  vector<shared_ptr<chaussette>> coutenu;
};
class Chaussette
{
  weak_ptr<Tiroir> tiroir;
}
```

Exemple 2 :

```
class Tiroir
{
  vector<weak_ptr<chaussette>>coutenu;
};
class Chaussette
{
  shared_ptr<Tiroir>tiroir;
};
```

Deuxième partie

STL les conteneurs

STL : **S**tandard **T**emplate **L**ibrary namespace : std

But des conteneurs : gérer un ensemble d'objets (similaire à un tableau) tout en gérant la mémoire pour vous.

Avantages :

- Pour tout le contenu, il existe un moyen commun pour itérer sur ces contenus (les itérateurs).
- Pour tout le contenu, on peut lui appliquer un foncteur (c'est-à-dire un algorithme).
- Ce sont des classes génériques.

2 Les itérateurs

Un itérateur est une abstraction des pointeurs et permet de lister chacun des éléments d'un conteneur.

Méthodes :

- Pour déréférencer et obtenir l'élément.
- ++ pour accéder à l'élément suivant.

Chaque conteneur a un itérateur spécifique.

Méthode du conteneur :

- begin() : 1er élément
- end() : «dernier» élément

Exemple : Si «Conteneur» est votre contenu,

```
Conteneur.iterator i = instance_conteneur.begin();
while(i != instance_conteneur.end())
{
  f(*i);
  i++;
}
```

Remarque : Si on ne modifie pas l'élément, il faut utiliser :

```
const_iterator
```

Remarque : Si le contenu est bidirectionnel, pour aller à l'élément précédent, on utilise :

```
reverse_iterator (et ++ pour l element precedent)
ou
const_reverse_iterator
```

3 Comparaison de conteneurs

$$\left.\begin{array}{l} == \\ != \end{array}\right\}$$ Si 2 conteneurs sont égaux ou non

$<$ $<=$ $>$ $>=$ Si le template supporte ces opérations.

4 Séquences

Ce sont des conteneurs opérant de manière séquentielle (dans un certain ordre) sur leurs éléments. La STL en implémente beaucoup.

4.1 Les listes

Exemple :

```
std::list<double>l;
// La liste est remplie
void print_li(const list<double> &l)
{
  list<double>::const_iterator i = l.begin();
  while(i != l.end())
  {
    std::cout<<*i<<std::endl;
    i++;
  }
}
```

```
void f()
{
  std::list<double> l1(3,5); // 3 elements de valeur 5 chacun
  print_li(l1);
  std::list<double> l2=l1
}
```

Quelques propriétés :

- Liste doublement chaînée.
- Insertion et supression d'éléments en O(1). (Méthode insert() et erase())
- Insertion au début et à la fin. (respectivement)
  1. push_front
  2. push_back (respectivement)
- idem pour la supression :
  1. pop_front
  2. pop_back

Exemple :

```
typedef list<int> li;
void print_li(const li &l); // voir ci-avant
void f(){
  int val[] = {2,5,7,7,3,3,2,6,6,3,4};
  li l;
  for (int i = 0; i<sizeof(val)/sizeof(int); i++)
    l.push_back(val[i]);
  print_li(l);
  l1.reverse();
  print_li(l);
  l1.sort();
  print_li(l);
  l1.reverse(3); // enleve tous les 3 de la liste.
}
```

4.2 La classe «vector»

Sémantique proche des tableaux du C.

Propriétés :

- Unidirectionnel
- Insertion et suppression en O(n)
- Accès aléatoire. (on peut avoir directement accès à n'importe quel élément)
- Insertion peut faire une allocation de mémoire

Méthodes : idem que les listes plus :

```
operator[](int i)
```

```
at(int i)
```

renvoie l'élément numéro i

− > Peut renvoyer une exception. at() est moins performant que operator[]

Exemple :

```
int main()
{
  typedef vector<int> vi;
  vi v(10); // 10e element
  v.at(0) = 2;
  v.at(5) = 7;
  v.resize(11);
  v.at(10) = 5;
  v.push_back(13);
}
for (int i = 0; i < v.size(); i ++)
{
```

```
  std::count<<v[i] <<std::endl;
}
```

2	.	.	.	7	.	.	.	.	9	13

Propriétés :

— Accès aléatoire aux éléments

— Ajout et suppression au début et à la fin en O(1) sinon O(n)

Mêmes méthodes que vector.

4.3 Class stack et class queue

stack : pile LIFO queue : pile FIFO

4.4 conteneurs associatifs

— Ils associent une valeur à une cléf

— Recherche en O(ln(a)) ⇒ Appropriés quand on fait beaucoup de recherche.

Exemple :

map

multimap

pair

Exemple :

```
int main()
{
  typedef map<int, char*> mic;
  typedef list<pair<int, char*>> lp;
  lp l;
  l.push_back(lp::value_type(1,"un"));
  l.push_back(lp::value_type(2,"deux"));
  l.push_back(lp::value_type(3,"trois"));
  l.push_back(lp::value_type(4,"quatre"));

  mic i(l.begin(),l.end()); // 4 elements
  mic::iterator j = i.begin();
  while(j != i.end())
  {
    count << j->second() <<endl; //afficher la chaine de caracteres
          //de chaque element
  j++;
  }
  return 0;
}
```

4.5 Class valarray :

But : Faire des calculs d'algèbre linéaire de manière optimisée (utilise les instructions SIRD pour les calculs)

Troisième partie

C++11

Spécification sortie aux alentours de 2011 But : Rendre le C++ plus facile à apprendre et plus rapide.

Compilateurs : g++ clang++ vc++14 (Visual 2015) ont implémentés le C++11

5 Le spécificateur auto

Il permet la simplification d'une variable.

```
double d = 5.0:
```

Il serait inutile de dire que d est un `double` car 5.0 est un double.

On peut remplacer `double` par `auto`.

```
auto d = 5.0; // d est le type double
```

Remarque :

- Inutile pour les types de base
- En plus, si nous voulons un double et que l'on écrit :
  - `auto d = 5; // d est un int !`
  - `d2 = 8/d; // != de 8.0/5.0`
- Si on veut un float : `auto f = 5.0f;`

Utile quand le type est abominable. - Auto fonctionne aussi avec les types de retour de fonction :

```
int add(int x, int y){
  return x + y;
}
int f()
{
  auto sum = add(s,7); // sum de type int car add() retourne un int.
  return 0;
}
```

Utile avec les itérateurs :

```
for (std::vector<int>::const_iterator iter = nom_vect_cbegin();
iter != nom_vector_cend(); iter++)
{
  =
}
```

Pas forcément évident de trouver le type iter, d'où :

```
for (auto iter = nom_vect.cbegin(); iter!=nom_vect.cend(); iter++){
  =
}
```

Remarque : **c**begin() et **c**end() pour des itérateurs **constants**

6 decltype

decltype permet de trouver le type d'une expression au moment de la compilation.

Remarque : decltype est utilisée pour la méta-programmation (voir ci-après).

Exemple :

```
decltype(5) x; // x est de type int
decltype(x) y = 6; // y est de type celui de x, i.e. int
auto z = x; // z est de type int.
T& operator [ ](int);
T operator [ ](int) const;
```

Remarque : decltype et auto peuvent ne pas déduire le même type.

```
const std::vector<int> v(5);
auto a = v[0]; // a est de type int car operator [ ] renvoie un int;
decltype (v[0]) b = 1; // b est de type : const int &;
```

7 Les classes enum

Les énumérations ne sont pas fortement typées.

Exemple :

```
enum couleur
{
  ROUGE,
  VERT,
  BLEU,
};

enum fruit
{
  BANANE,
  POMME
};
Couleur a = ROUGE;
fruit b = BANANE;
if (a==b){
  =
}
```

```
else {
  =
}
```

Une "enum" est de type int. Donc le compilateur peut comparer a et b. Mais en terme de logique, on ne peut pas comparer des fruits et des couleurs.

Le C++11 a introduit les classes enum pour rendre les énumérations fortement typées (sorte de namespace).

```
enum class couleur
{
  ROUGE,
  VERT,
  BLEU
};

enum class fruit
{
  BANANE,
  POMME
};
couleur a = couleur::ROUGE;
fruit b = fruit::BANANE;
if (a==b) // erreur de compilation
{
  =
}
```

Remarque : Si vraiment on veut == entre 2 classes enum, il faut surcharger operator ==

```
enum class RGB
{
  ROUGE,
  VERT,
  BLEU
};
enum class HSV
{
  HUE,
  SATURATION,
  VALUE
};
```

(⇒ Légitimité de surcharger l'opérateur ==)

8 Boucle et domaine de valeur (Range-base for statement)

Pour les itérateurs, le spécificateur auto permet de simplifier les déclarations. C'est une utilisation tellement courante qu'il existe une autre syntaxe plus simple pour la boucle for !

Exemple :

```
typedef vector<int> nom_vect; nom_vect v; // v est initialise
for(auto x : v)
{
  count<<x;
}
```

x parcourt tout v. Pour modifier le vecteur :

```
for (auto &x : v) // x est modifiable
  count << ++x;
```

Remarque : cette syntaxe fonctionne avec :

- Tableaux du C
- Les itérateurs de la STL
- le type std : :string

9 Constructeurs délégués

Cadre :

```
class vect
{
  int sz_;
  double *data_;
  vect(int sz)
  {
    sz_ = sz;
    data_ = new double [ sz ] ;
  }
  vect (int sz, double val){
    {
      sz_ = sz;
      data_ = new double [ sz ] ;
      for (auto i : data_)
        data_ [ i ] = val;
    }
  }
}
```

Le même code est écrit dans les 2 constructeurs :

1re solution : On utilise une fonction intermédiaire.

2e solution : Constructeurs délégués.

```
class vect
{
  int sz_;
  double * data_;
  vect(int sz)
```

```
  {
  sz_ = sz;
  data_ = new double [ sz ] ;
  }
  vect(int sz, double val) : vect(sz)
  {
    for(auto i : data_)
      data_ [ i ] = val;
  }
};
```

Avantages :

— Pas de duplication de code

— Meilleure lisibilité (surtout comparé à l'utilisation de fonctions intermédiaires)

10 La sémantique move et les références de rvalue

Avant le C++11, il y avait un problème de performance dans la création d'objets temporaires. Quelques fois, le compilateur peut optimiser mais ce n'est pas suffisant.

Exemple :

```
vector<int> double_val (const vector <int> &v)
{
  vector<int> new_val;
  new_val.reserve(v.size [ ] ); // meme nombre d'elements pour v et new_val
  for(auto iter=v.begin(); iter != v.end(); iter++)
  {
  new_values.push_back(2* *iter);
  }
  return new_values;
}
int menu()
{
  vector<int> v;
  for(int i = 0; i < 100; i++){
    v.push_back(i);
  }
  v = double_val(v);
}
```

Problème : Il existe une copie de l'objet retournée et stockée dans une variable temporaire. Elle est au niveau de v = ...

— Quand double_val est appelé, il construit une copie v de main().

— Il multiplie par 2 les valeurs de cette copie de v.

— Quand il retourne la valeur new_val, il y a aussi une copie du vecteur.

2 copies de vecteurs.

Au niveau des performances, c'est préjudiciable.

Les objets temporaires sont crées et détruits «inutilement».

Exemple :

```
int i = j + k;
```

l = j + k i = l l est détruit

Idée : «Déplacer» une variable dans une autre plutôt que :

- La variable copiée dans tmp
- modifier tmp
- copier tmp dans la 2^{e} variable

⇒ Opérateur de déplacement

10.1 rvalue et lvalue

v = val

v : lvalue

val : rvalue

Une lvalue (left value) est une partie de la mémoire qui est (semi) permanente dans laquelle on peut stocker des valeurs.

Remarque : semi : pour une variable qui est déclarée dans une bloc. Remarque : Ce n'est pas que pour les variables.

Exemple :

```
int x;
int &get_ref()
{
  return x;
}
get_ref() = 4; // x vaut 4.
```

Ici get_ref() retourne une adresse vers une partie de mémoire permanente.

Définition : une expression est une rvalue si elle résulte d'un objet temporaire.

Exemple : (variables)

```
int i = j + k;
/* j + k est une rvalue
il faut calculer l = j+k
i = l
l est detruit
l est l'objet temporaire
```

```
*/
```

Exemple : (fonctions)

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

Le return ⇒ un objet temporaire

Résumé :

```
int & get_ref(); // lvalue a cause du &
int get_val(); // rvalue
```

Autre exemple :

```
std::string get_name()
{
  return "foo";
}
std::string n=get_name();
-> /* get -Name renvoie "foo" dans une string \Rightarrow objet temporaire std::string tmp(foo)
copie de tmp dans m
*/
```

10.2 Détection des objets temporaires avec les références de rvalue

Le C++11 introduit les références de rvalue pour détecter si une expression résulte d'un objet temporaire.

Avant le C++11, on n'avait que des références lvalue.

Les références de rvalue font le lien entre un objet mutable (non constant) et une rvalue.

Syntaxe : &&

Exemple :

```
const std::string && name = get_name();
std::string && name = get_name();
```

Exemple : (fonctions)

```
print_ref(const std::string &str)
{
  cout << str;
}
print_ref(const std::string &&str){
  count << str;
}
str::string foo("foo");
print_ref(foo); // 1er print_ref qui est appele
print_ref(get_name()); // 2e print_ref qui est appele

print_ref(get_name());
/* 2e print_ref qui est appele car get_name renvoie une rvalue donc la surcharge choisit
le print_ref qui a une reference de rvalue en parametre
*/
```

10.3 Constructeur de déplacement

Comme le constructeur de copie, mais pas d'allocation de mémoire. On «vole» les données de l'objet «copié» car l'utilisation des r-value références détecte que c'est un objet temporaire.

- Si l'objet temporaire est un type de base, on fait une affectation
- Si l'objet temporaire est un pointeur, on ne fait pas d'allocation de mémoire, on fait une affectation de pointeur. Le pointeur de l'objet temporaire DOIT être mis à NULL.

Exemple :

```
class Tab
{
  int size_;
  double *data_;
  public :
  Tab(int n)
  {
  size_ = n;
  data_ = new double [n];
  }
  Tab(const Tab &t)
  {
    size_ = t.size_;
    data_ = new double [size_];
    for (int i = 0; i < size_; data_[i] = t.data[i]; i++) {}
  }
  ~Tab()
  {
    delete [] data;
  }
  // Constructeur de deplacement
  Tab(Tab &&t)
```

```
  {
    // On "vole" les donnees de t.
    size_ = t.size_;
    data_ = t.data_
    // IMPORTANT :
    t.size_ = 0;
    t.data_ = NULL;
  }
};
Tab operator+(const Tab &t1, const Tab &t2)
{
   =
}

//Utilisation :
t3 = t1 + t2;
```

Étapes :

1. Création de l'objet et temporaire tmp
2. t est le résultat de l'opérator+(t1,t2)
3. le compilateur détecte que tmp est un objet temporaire

   ⇒ il utilise le constructeur de déplacement.

   C'est-à-dire : $\begin{cases} t3.size\_ = tmp.size\_; \\ t3.data\_ = tmp.data\_; \\ tmp.data\_ = NULL; \\ tmp.size\_ = 0; \end{cases}$
4. Destruction de tmp (appel du destructeur) :

   delete[] tmp.data_ ;

Comme tmp.data_ vaut NULL, delete ne fait rien.

Remarque :

Si dans le constructeur de déplacement, on met : t.data_ = NULL ;

Le destructeur est appelé sur tmp ⇒ delete[] tmp.data_.

Puis quand t3 est détruit, le destructeur est appelé sur t3 : delete [] t3.data_ qui vaut tmp.data_ et qui a déjà été détruit ⇒ le programme plante.

2 choses importantes :

- Le paramètre ne peut pas être une rvalue référence constante :
  on modifie l'objet courant
- IL FAUT définir à NULL le pointeur en tant que membre de données de l'objet temporaire (t.data_ = NULL ;)

Conséquences : La surcharge du constructeur de déplacement ⇒

- Moins de consommation de mémoire
- Gain en rapidité

11 Initialiseur de listes

En C ou en C++98, on peut initialiser un tableau au moment de sa déclaration. Si la taille du tableau n'est pas indiquée, sa taille est celle de l'initialiseur.

Exemple :

```
int v[ ] : {1, 3, 5, 7, 9};
int v[5] = {1, 3, 5}; // vaut {1, 3, 5, 0, 0}
int v[2] = {1, 3, 5}; // erreur
```

Pour initialiser un std : :vector, on peut passer par un tableau statique :

Exemple :

```
int val[ ] = {1, 3, 5, 7, 9};
std::vector<int> v (val, val+ sizeof(size)/sizeof(int));
// sizeof(size)/sizeof(int) vaut 5, nombre d'éléments de val;
void fct(int i)
{
   std::cout << "i␣:" << std::endl;
}
for_each(v.begin(); v.end(); fct);
```

But : uniformiser la syntaxe et utiliser , , , avec d'autres types que des tableaux :

Le C++ introduit le type std:: initializer_list <T>

```
void cout_all (std::initialiszer_list <int> l)
{
  for_each(_______________);
}
cout_all({1, 3, 5, 7, 9};
```

On peut l'utiliser dans un constructeur :

```
struct A
{
   std::vector <int> v;
   A(std::initalize_list<T> l)
   {
     v.resize(l.size);
     std::copy(l.begin(); l.end(); ~.begin());
   }
};

Utilisation :
A a({1, 2, 3, 4});
A a{1, 2, 3, 4};
A a({1, 2, 3 ,4}; // idem sans =
```

Tous les conteneurs (vectors, list,...) les string et les regex ont été mis à jour.

Pour des structures + complexes, on peut embboîter les initialiseurs :

```
list <pair <string, int>> val =
{
   {"1er", 1};
   {"2e", 2};
   {"3e", 3}
};
```

Conversions implicites :

```
vector <double> v = {1, 2, 3}; // v vaut {1.0, 2.0, 3.0}
```

/!\ : L'utilisation des initialisateurs peut introduire soit des confusions soit des erreurs :

```
vector<int> v1(7); // 7 éléments mis à 0;
vector<int> v2{7}; // 1 élément valant 7;
vector<int> v3 = 7; // erreur
vector<int> v4 = {7}; // 1 élément valant 7;
vector<int> v5 = vector<int> (5); // idem v1;
vector<int> v6 = vector<int> {5}; // idem v2;
vector <int> v7;
vector <int> v8;
v7 = 5; // erreur idem v3
v8 = {5}; // idem v4
range_based for :
for(const auto i : {1, 3, 5, 7, 9})
   count << i << endl;
```

12 Expressions constantes

— En C++, 3+4 est une expression constante et évaluée au moment de la compilation :

— Les tableaux doivent avoir pour taille une expression constante (les fonctions ne sont pas autorisées)

```
int fct() \{return 7;\}

int tab[ fct() + 5]; // pas autorisée
```

Le C++11 introduit le mot clé constexp garantissant qu'une fonction ou un constructeur est une constante au moment de la compilation.

Exemple :

```
constexp int fct()
{
  return 7;
}
int tab[ fct() + 5 ]; // tableau à 12 éléments
```

1) Restrictions :

- La fonction doit retourner une valeur
- Le corps de la fonction ne doit pas retourner de variable
- Le corps de la fonction ne doit pas définir de nouveaux types
- La fonction doit retourner une expression constante
- La fonction doit avoir uniquement des arguments qui sont des expressions constantes. (doivent avoir constexp devant).

2) Cas d'une variable :

```
constexp a = 7.8;
constexp b = a/8.7; // possible car a est une expression constante
```

3) Cas d'un constructeur : Un constructeur constant :

- doit définir les membres de données avec des expressions constantes
- ne doit pas définir de variables
- ne doit pas définir de nouveaux types

Ceci permet d'évaluer des fonctions ou des méthodes au moment de la compilation.
Une application classique : la méta programmation.

Exemple :

Calcul de la puissance d'entiers au moment de la compilation en utilisant les `constexpr` et les templates (méta programmation)

```
#include <iostream>
// version expression constante :
constexpr unsigned int pow_constexpr(constexpr unsigned int base,
                  constexpr unsigned int puiss)
{
   return(puiss == 0)? 1 : pow_constexpr(base, puiss-1);
}
// version template (voir plus loin : la méta programmation)
template<unsigned int base, unsigned int puiss>
struct pow_template
{
   enum
   {
     value = base* pow_template<base,puissance - 1>::value;
   };
};

template <unsigned int base>
struct template_pow <base, 0>
{
   enum{value = 1};
};

int main()
{
   std::count << "pow_constexpr(5,3) = " << pow_constexpr(5,3) << std::endl;
   std::count << "pow_template(5,3) = "pow_template<5,3>::value << std::endl;
```

```
    return 0;
}
```

13 Initilisation de membres de données non statiques

Jusqu'à présent, on pouvait initialiser des membres de données statiques (nom statique : dans le constructeur)

Exemple :

```
class A
{
    static const int i1 = 7; // OK
    const int i2 = 7; // Erreur
    static int i3 = 7; // Erreur
    static const int i4 = var; // Erreur
};
// En C++11 c est possible :
class A
{
    int i = 42;
};
// Équivalent à :
class A
{
    int i;
    public :
    A() {i = 42;}
};
// Fonctionnalité intéressante avec plusieurs constructeurs :
class A
{
    int a_;
    int b_;
    std::string s_;
    public :
    A() : a_(7), b_(5), s_("toto"){}
    A(int a) : a_(a), b_(5), s_("toto"){}
    A(D g); a_(7), b_(g()), s_("toto"){}
};
\\ Devient :
class A
{
    int a_ = 7;
    int b_ = 5;
    std::string s_ = "toto";
    public:
    A(){} // Initialise a_, b_ et s_ à respectivement 7, 5 et "toto"
    A(int a) : a_(a){} // Initialise b_ et s_ respectivement à 5 et "toto"
    A(D g) : b_(g(){}) // Initialise a_ et s_ respectivement à 7 et "toto"
};
```

14 Surcharge explicite de la virtualité

Lors de l'héritage il est facile de se tromper sur la déclaration d'une méthode de la classe file (on re-déclare une méthode alors que l'on ne veut pas).

Exemple :

```
class A
{
   virtual void f();
   virtual void g() const;
   virtual void k(char);
   void k();
};

class B : public A
{
   void f(); // hérite de A::f()
   void g(); // n'hérite pas de A::g() car pas const
   virtual void k(char); //hérite de A::k
   void k(); // n'hérite pas de A::k car A::k n'est pas virtuelle
}
```

Le C++11 introduit 2 mots clefs pour essayer de limiter les erreurs :

— final : la méthode n'est pas héritable

— override : la méthode doit hériter d'une autre méthode

Exemple :

```
class A
{
   virtual void f() final;
   virtual void g() const;
   virtual void k(char);
   void k();
};

class B : public A
{
   void f(); // Erreur car A::f n'est pas héritable
   void f() override; // Erreur : g doit hériter de A::g, ce qui n'est pas le cas
   virtual void k(char); // Hérite de A::k,
}
```

15 Fonction lambda (ou anonyme)

Dans la STL, les algorithmes sont des fonctions qui généralement itèrent sur des conteneurs. Principe : on itère à partir d'un élément de ce conteneur jusqu'à un autre élément et on applique à chacun de ces éléments, soit une fonction classique (dite libre), soit une fonction (un objet-fonction).

Remarque : La déclaration de la fonction ou du foncteur dépend de l'algorithme.

Exemple :

```
std::vector<int> = {50, -10, 20, -30};
// tri par défaut :
std::sort(v.begin(),v.end()); // v vaut {-30, -10, 20, 50}
// tri par valeur absolue avec une fonction libre :
bool val_abs(int i, int j)
{
   return abs(i) < abs(j);
}
std:sort(v.begin(), v.end(), val_abs); // v vaut -10, 20, -30, 50
//tri avec un foncteur :
struct Abs
{
   bool operator()(int i, int j)
   {
     return abs(i) < abs(j);
   }
};
Abs foncteur;
std::sort(v.begin(), v.end(), foncteur); // v vaut {-10, 20, -30, 50}
```

Un des buts des fonctions lambda est de créer facilement des foncteurs. Syntaxe des fonctions lambda : [liste de capture] (paramètre) retour //code

[] **obligatoires**

Remarque : elle n'a pas de nom.

Description de la syntaxe :

1. liste de capture :
   liste de noms de variables (éventuellement séparées par des virgules). Ce sont des variables externes à la fonction lambda utilisables dans le corps de la lambda (exemple : une variable globale)
2. paramètres (optionnels) :
   Paramètres renvoyés par l'algorithme
3. retour (optionnel)
   - type de retour de la lambda :
     - Soit rien (void par défaut si la fonction ne retourne rien, ou le type de l'expression après return).
     - Soit void
     - Soit $->$ type
4. Code : La définition de la fonction lambda.

Pour le tri par valeur absolue :

1. rien
2. 2 paramètres (ceux demandés par l'algorithme)
3. détecté automatiquement

```
[ ] (int i, int j) { return abs(i) < abs(j);}
```

Utilisation : directement dans l'algorithme.

```
std::sort(v.begin(), v.end(), [ ](int i, int j) {return abs(i)<abs(j);});
```

Syntaxe pour 1 :

[] : ne capture rien

[k] : capture toutes les variables par référence

[=] : capture toutes les variables par copie

[& count] : capture la variable count par référence

[= count] : capture la variable count par copie

[= i, &j]

Exemple :

```
std::vector<int> v = {150, -10, 20, -30}
std::vector<int> index (v.size());
int count = 0;
generate(index.begin(), index.end(), [&count] () {return ++ count;});
// index vaut {1, 2, 3, 4}
```

Autres syntaxes possibles pour la fonction lambda :

```
generate(index.begin(), index.end(), [&count] ()->int(return ++count;});
generate(index.begin(), index.end(), [&count] ->decltype(++count){return ++count;});
```

Exemple : (dans une méthode)

```
std::vector<int> list{1, 2, 3, 4, 5};
  int total = 0;
  int value = 5;
  std::for_each(list.begin(),list.end(), [&,value, this ] (int x)
  {
    total += x * value * this->ma_methode();
  }
  );
```

Remarque : on peut utiliser les fonctions lambda comme argument de la fonction (par exemple 1, mais son nom n'existe qu'en interne au compilateur)

On utilise auto pour déterminer son type :

```
auto lambda1 = [&] (int x){//code };
auto lambda2 = new auto([=] (int x){//code });
```

16 Méta-programmation

But : interpréter du code non pas à l'exécution du programme mais à la compilation du programme.

Exemple : de bibliothèques utilisant la méta-programmation :

- Boost (multi plate-forme)
- Spirit (parser)
- blitz++

Pour cela, on utilise les templates

```
template<typename T>
const T & Max (const T &x, const T &y)
{
  return(x>y) ? x : y;
}
int i = Max<int> (5,4);
double d = Max<double>(8.7, 1.2);
std::string s = Max<std::string> (std::string("Foo"), std::string("Bar"));
```

Le compilateur (à la compilation) va interpréter T comme si c'était un int, un double ou un std : :string
La méta-programmation sera exactement ceci : interpréter du code à la compilation avec pour but du calcul ou de la génération de code
L'exécution du programme utilisera les résultats fourni par la compilation :

Inconvénients :

- La compilation est plus lente
- Le code de la méta-programmation DOIT être connu au moment de la compilation
- Beaucoup de mémoire peut être utilisée

Avantages :

- Exécution très rapide

7 Décembre 2017

Ce que permet la Méta-programmation :

- Faire des calculs mathématiques plus rapides
  - Algorithmes de tris
  - Calculs matriciels
  - ...
- Générer du code automatiquement à la compilation
  - Écriture de parser
  - Analyse syntaxique

16.1 Calculs Mathématiques

Basé sur les classes template et les énumérations :

Exemple : (factorielle)

Remarque : on n'utilise pas de boucle. (Car la boucle se fait à l'exécution et non à la compilation). ⇒ On utilise un calcul récursif

```
template<unsigned int N>
struct Fact
{
```

```
  enum{Value = N*Fact <N - 1>::Value};
}
template<> struct Fact <0>
{
  enum{Value = 1};
};
// Utilisation
unsigned int x = Fact<3>::Value;
```

Quand ce code est compilé, x vaut 6.

Fact<3>::Value est remplacé par 3*Fact<2>::Value

Fact<2>::Value est remplacé par 2*Fact<1>::Value

Fact<1>::Value est remplacé par 1*Fact<0>::Value

Fact<0>::Value vaut 1. Donc les principes de base :

— Utilisation de templates entiers

— Utilisation d'une forme de récursivité

— On n'utilise aucune boucle

16.1.1 Classes enum

On n'utilise plus de fonctions (exception : voir plus loin).

— Les fonctions sont remplacées par des classes ou structures.

— Les variables/types de retour sont remplacés par des énumérations.

Exemple : (fonction)

```
int id(int x)
{
  return x;
}
int x = id(5); // Valeur retournée par id comme à l'exécution
```

Exemple :(metaprog)

```
template<int x>
struct Id
{
  enum{Value = x};
};
int x = Id<5>::Value;
```

16.1.2 Fonctions en ligne (inline)

Ce sont des fonctions dont l'appel est remplacé par le code au moment de la compilation.

Utilité : utilisation de données fournies par l'utilisateur.

Remarque : Le remplacement de l'appel de la fonction en ligne par son code est contrôlé par le compilateur.

Exemple :

```
inline int add(int x, int y)
{
  return x+y;
}
int sum = add(2,3);
```

Remarque :

— Préférable d'utiliser une fonction statique : `static inline int` add(...) {−−}

— inline peut ne pas être reconnu par tous les compilateurs

1. C83 : – inline –
2. VC++ : – inline ou – force inline
3. gcc (clang) inline ou –inline–

```
#ifdef - WIN32
#define Inline --forceinline
#else
#define Inline inline
#endif
```

— Les compilateurs ont des options pour contrôler le caractère en ligne des fonctions.

Exemple : (metaprog)

g++ -O3

↪ Entre autre, rend en ligne toute fonction

Remarque : si une fonction en ligne s'exécute rapidement, le programme prend plus de place sur le disque. (⇒ plus de temps de chargement en mémoire du programme).

Exemple avec la méta-programmation : (toujours la récursivité)

```
static inline int add(int x, int y)
{
  return y = 0? x : add(x,y - 1) + 1;
}
int x : add(5,3);

// Remplacé par :
add(5,2) + 1
// Remplacé par :
add(5,1) + 1
// Remplacé par :
add(5,0) + 1
||
5
//i.e. 8
```

16.1.3 Test

Les tests if/else sont interdits :

Solution : Class template avec un bool.

Exemple :

```
template <Bool condition> struct Test {};
template<> struct Test <true>
{
  static void Do()
  {
    fait_quelque_chose();
  }
}

template<> struct Test<false>
{
  static void Do()
  {
    fait_autre_chose();
  }
}

Test <true>::Do(); // Exécute fait_quelque_chose();
Test <false>::Do(); // Exécute fait_autre_chose();
```

Remarque : si on utilise cond variable de type bool, Test<cond>::Do(); Cond doit être connu au moment de la compilation.

Remarque : programmation classique :

```
if (cond)
  fait_quelque_chose();
else
  fait_autre_chose();
```

On peut utilisateur l'opérateur ternaire :

Exemple :

```
template <int i> struct NumTest
{
  enum
  {
  EST_PAIR = (i % 2 ? false : true);
  EST_NUL = ((i == 0) ? true : false);
  };
};
bool b1 = NumTest<45> :: EST_PAIR; //false
bool b2 = NumTest<0>:: EST_NUL // true
```

16.1.4 Boucles :

On ne peut pas utiliser de boucles for ou while ou do-while ⇒ Récursivité

Exemple :

```
// itération :
template <int Being, int End>
struct Loop
{
  static void Do()
  {
  fait_quelque_chose();
  Loop <Begin + 1, End>::Do();
  }
};
// arrêt
template <int n> struct Loop <n,n>
{
  static void Do(){}
};
```

Utilisation : Loop <5,15>::Do();

Utilisation classique :

```
for (i = 5; i < 15; i++)
{
  fait_quelque_chose();
}
```

16.1.5 Variables temporaires

On utilise des enum dans les classes génériques.

Exemple :

```
template<int x, int y> struct Calcul
{
  enum
  {
    tmp1 = x + z + y;
    tmp2 = y + z + x;
    tmp3 = x < y ? 0 : 1;
    value = tmp3? tmp1:tmp2;
  };
};
```

16.2 Réécriture de quelques fonctions mathématiques

Il faut commencer par savoir comment calculer récursivement ces fonctions.

16.2.1 Factorielle

```
n! = n*(n-1)!
template <unsigned int N>
struct Fact
{
  enum
  {
    Value = N*Fact<N - 1>::Value
  };
};
template<> struct Fact<0>
{
  enum{Value = 1};
};
```

Remarque : pour N>12 : utiliser des doubles.

```
template<int i> inline double Fact()
{
  return i*Fact <i - 1>();
}
template<> inline double Fact<0>()
{
  return 1;
}
```

16.2.2 Puissance

$$a^b (b\ entier) \tag{1}$$

$$b > 0\ ;\ a^b = a \times a^{b-1} \tag{2}$$

```
template<int n> inline double puiss(double x)
{
  return x*puiss<n - 1>(x);
}
template<> inline double puiss<0> (double x)
{
  return 1;
}
```

16.2.3 Exponentielle

$$e^x = \sum_{k=0}^{+\infty} \frac{x^n}{n!} \tag{3}$$

$$S_N = \sum_{k=0}^{N} \frac{x^n}{n!} \tag{4}$$

$$S_N = \frac{x^n}{n!} + S_{N-1} \tag{5}$$

```
template<int n> inline double expo(double x)
{
  return expo<N-1>(x) + puiss<i>(x) / fact<i>();
}
template<> inline double expo<0> (double x)
{
  return 1;
}
template<int n> inline double Exp(double x)
{
  return (x > 0) ? expo<n>(x) : expo<n>(-x);
}
```

16.2.4 Cosinus

$$\cos(x) = \sum_{k=0}^{+\infty} (-1)^k \frac{x^{2k}}{(2k)!} \tag{6}$$

$$S_n = \sum_{k=0}^{n} (-1)^k \frac{x^{2k}}{(2k)!} \qquad \text{et} \qquad S_n = (-1)^n \frac{x^{2n}}{(2n)!} + S_{n-1} \tag{7}$$

```
template <int n> inline double Cos(double x)
{
  return Cos<n-1>(x) + ((n%2)? -1 : 1)*puiss<2*n>(x)/fact<2*n>();
}
template<> inline double Cos<0> (double x)
{
  return 10;
}
```

16.2.5 Tangente hyperbolique

$$\arctan(x) = \sum_{n=0}^{+\infty} \frac{x^{2n+1}}{(2n+1)} \tag{8}$$

```
template <int n> inline double arctan(double x)
{
  return arctan<n-1>(x) + puiss<2*n+1>(x)/(2*n+1);
}
template<> inline double arctan<0> (double x)
{
  return x;
}
```

Remarque :

— Ces séries convergent toutes lentement

— ⇒ optimiser les calculs pour rendre la partie compilation plus rapide

16.3 Génération automatique du code

les typelist

Les templates ne sont plus des types entiers mais des classes

```
template <class T1, class T2>
struct TypeList
{
  typedef T1 tete;
  typedef T2 queue;
};
```

La construction de la liste sera récursive.

```
Typelist<Type1, TypeList<Type2, ... ... TypeList<TypeN, TypeNULL>>...>
struct TypeNULL{};
```

Utilisation : pour avoir la liste de [Double, `float`, `int`]

`typedef` TypeList <`double`, TypeList<`float`, TypeList<`int`,TypeNULL>>>

Ceci génère :

```
struct TL
{
  typedef double tete;
  typedef struct TL
  {
    typedef float tete;
    typedef TL
    {
      typedef int tete;
      typedef struct {} queue;
    } queue;
  } queue;
};
```

16.4 Bibliothèque

Modern C++ Design (Andrei Alexandrescu, Addison-Wesley)

17 Calcul parallèle

17.1 Tâche (thread)

Toute application lance un processus qui exécute une série d'instructions

Un processus exécute en particulier certaines instructions dans le thread dit principal. Toutes les autres sont exécutées dans d'éventuels threads secondaires.

Exemple :

Les instructions du thread principal sont celles de la fonction main().

Différences : entre processus et thread :

— Un processus a sa propre mémoire virtuelle

— Les threads d'un processus se partagent la mémoire viruelle

— Chaque thread a sa propre pile d'appel et peut gérer des variables indépendemment des autres thread (TLS : Thread Local Storage)

Le processus d'un ordinateur a plusieurs cœurs, chaque thread s'exécute sur un cœur ⇒ permet d'exécuter du code simultanément sur chaque cœur. (encodage et décodage de vidéos, calculs mathématiques...)

Remarque :

— Un thread exécute une fonction bloquante (boucle infinie s'arrêtant dans un cas spécifique)

Mais dans ce cas seul, un thread secondaire est bloqué, le processus ne l'est pas.

17.2 API

Le C++11 permet la création de threads et la synchronisation de ceux-ci (la possibilité d'attendre qu'un thread se termine pour en lancer un autre).

17.2.1 Threads

Classe : std::thread

Inclure : thread

Un constructeur par défaut.

Un constructeur prenant une fonction et des données passées à cette fonction. (nom de cette fonction : callback). On aussi peut utiliser une lambda.

Exemple :

```
#include <iostream>
#include <thread>

// Fonction qui sera exécutée dans le thread :
void affiche(int debut, int nombre)
{
  for (int i = debut; i < debut + nombre; i++)
```

```
  {
    std::count << i << ",";
  }
}

int main()
{
  // thread qui lance la fonction affiche
  std::thread t1(affiche, 0, 5);
  // thread qui lance la lambda
  std::thread t2([](){affiche(5, 5);});
  t1.join();
  t2.join();
  return 0;
}
```

Sortie possible :

```
0,1,2,3,4,5,6,7,8,9,
```

17.2.2 Méthode join()

Cette méthode bloque jusqu'à ce que le thread se termine. Remarque : Un thread se termine quand la fonction qu'il lance se termine.

17.3 Synchronisation des threads

Les threads étant exécutés en parallèle, l'ordre de ceux-ci n'est pas assuré. → Il faut un mécanisme pouvant, si nécessaire, garantir l'ordre d'exécution des threads. Ce sont les mutex.

Remarque : Quand il y a des problèmes de synchronisation de thread, on dit qu'il y a une "race-condition" (situation de compétition).

Exemple :

```
#include <iostream>
#include <thread>

void disp(int o)
{
  for (int i = 0; i < 10; i++)
  {
    std::cout << (i*3) + o << "␣";
  }
}

int main()
{
  std::thread t1(disp, 0);
  std::thread t2(disp, 1);
  std::thread t3(disp, 2);
  t1.join();
```

```
  t2.join();
  t3.join();
}
```

Sortie : Avec 2 exécutions successives :

```
0 3 1 2 6 4 5 9 7 8 ...
0 3 6 9 1 2 12 15 4 5 ...
```

Pour afficher dans l'ordre, on utilise, un std::mutex. Il existe trois autres types de mutex :

std::recursive_mutex

std::timed_mutex

std::recursived_timed_mutex

17.3.1 mutex

C'est un objet qui peut être verrouillé (locked) une fois. Il existe 2 méthodes :

— lock() qui verrouille le mutex

— unlock() qui déverrouille le mutex.

Si on vérouille un mutex qui est déjà verrouillé, lock() renvoie false. Pour tester si un mutex est verrouillé ou non, on utilise la méthode trylock() qui renvoie true si la mutex est vérouillé et false sinon.

Exemple : (modification de l'exemple)

```
#include <iostream>
#include <thread>
#include <mutex>

void display(int o, std::mutex lock)
{
  lock.lock();
  std::cout << "Thread #" << o << std::endl;
  for(int i = 0; i < 10; i++)
  {
    std::cout << (i*3) + o << " ";
  }
  std::cout << std::endl;
  lock.unlock();
}

int main()
{
  std::mutex lock;
  std::thread t1(display, 0, lock);
  std::thread t2(display, 1, lock);
  std::thread t3(display, 2, lock);
  t1.join();
  t2.join();
  t3.join();
  return 0;
}
```

Explication :

— t1 est lancé

⇒ display() qui est lancée

⇒ donc le mutex lock est verrouillé

— t2 est lancé

⇒ display() qui est lancée

⇒ lock.lock() est appelé. Mais comme il est déjà verrouillé, cette fonction bloque.

— t3 est lancé

⇒ idem : lock.lock() est blonquante

— Quand t1 se termine, lock.unlock() est appelé ⇒ lock est déverrouillé.

⇒ la fonction display() de t2 est exécutée. (display() de t3 est toujours bloquée).

— Quand t2 se termine, lock.unlock() est appelé et le display de t3 peut se finir.

Affichage

```
thread #0
0 3 6 9 12 ...
thread #1
1 4 7 10 12 ...
thread #2
2 5 8 11 14 ...
```

17.3.2 recursive_mutex

Même chose qu'un mutex, mais il peut être verrouillé plusieurs fois dans le même thread.

17.3.3 Les versions "timed"

Le mutex est bloqué pendant un certain laps de temps.

17.3.4 Situation de concurrence (race condition)

Situation classique :

On a m1 et m2, deux mutex et on a t1 et t2, deux threads. t1 lock m1 et m2 dans cet ordre. t2 lock m2 et m1 dans cet ordre.

Exemple :

```
#include <iostream>
#include <thread>
#include <mutex>

std::mutex m1, m2;
std::thread t1( [&m1, &m2] ()
{
  t1.lock();
  t2.lock();
  std::cout << "T1" << std::endl;
  m2.unlock();
```

```
  m1.unlock();
});
std::thread t2([&m1, &m2]()
{
  m2.lock();
  m1.lock();
  std::cout << "T2" << std::endl;
  m1.unlock();
  m2.unlock();
});
```

Scénario : t1 verrouille m1.

t2 verrouille m2.

t1 verrouille m2 ⇒ on bloque ici

t2 verrouille m1 ⇒ on bloque ici

⇒ les 2 threads sont bloqués (dead lock)

Cas classique où on a un inter-blocage : quand une exception qui est lancée au mauvais moment (avant un unlock())

lock_guard : Pour éviter ce problème où une exception est lancée avant le unlock, on peut utiliser un lock_guard qui verrouille le mutex et le déverrouille dans il est détruit.

Exemple :

```
std::mutex m;
{
  std::lock_guard lock(m);
  // m est verrouillé
}
// ici lock est détruit => m est déverrouillé automatiquement.
```

Exemple : (modification de l'exemple précédent

```
void display(std::mutex m, int o)
{
  std::lock_guard lock(m);
  std::cout << "Thread #" << o << std::endl;
  for (int i = 0; i < 10; i++)
  {
    std::cout << (i*3) + o << " ";
  }
  std:cout << std::endl;
} // lock est détruit => m est déverrouillé
```

unique_guard : Similaire à lock_guard mais déverrouille le mutex s'il possède un verrou.

Exemple :

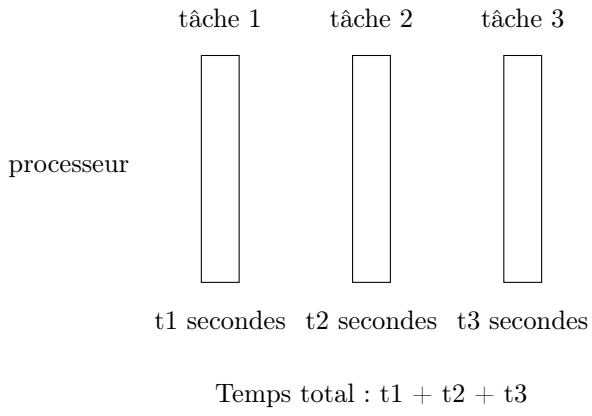
```
{
  std::mutex m;
  // ici code
  std::unique_lock <std::mutex> lock(m, std::defer_lock);
  if(!lock.try_lock())
  {
    // Le verrouillage n'est pas fait => on sort
    return;
  }
}
// m est déverrouillé car lock est détruit.
```

Utiliser les threads pour des calculs qui sont indépendants (ils doivent être exécutés dans des threads différents)

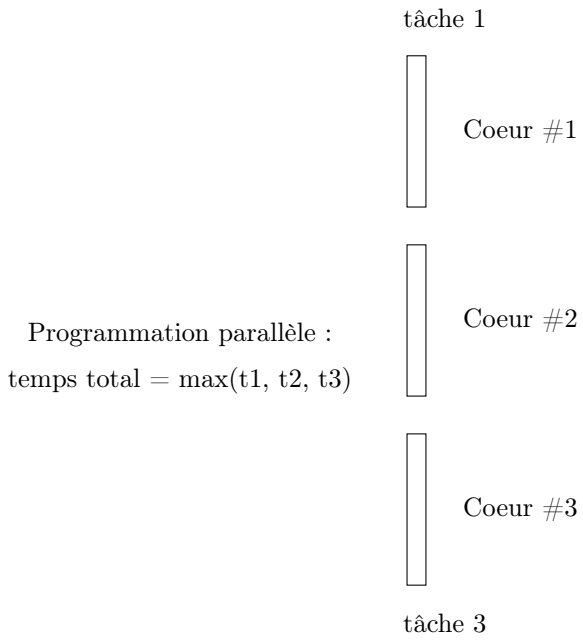
18 Tableau de valeurs

La programmation classique fournit une exécution séquentielle ou fournit des données, elles sont traitées par des fonctions dans un ordre précis et on obtient le résultat. La rapidité de calcul dépend de la puissance du processeur. Pour augmenter la vitesse de calcul 3 solutions se présentent :

- On augmente la vitesse du processeur
  Limitations physiques (vitesse limitée) et ça a un coût
- On augmente le nombre de processeurs => calculs parallèles
  Problème : il faut que les opérations soient indépendantes les unes des autres
  Séquentiel :



Temps total : t1 + t2 + t3



Le modèle de programmation parallèle s'appelle MIND (Multiple Instructions, Multiple Data).

— Solution intermédiaire : modèle de programmation SIND (Single Instruction, Multiple Data). Les données sont traités par bloc, avec un unique algorithme.

Les instructions sont MMX, ANDNow, SSE, 1, 2, 3, 4, AVX1, 2, ...

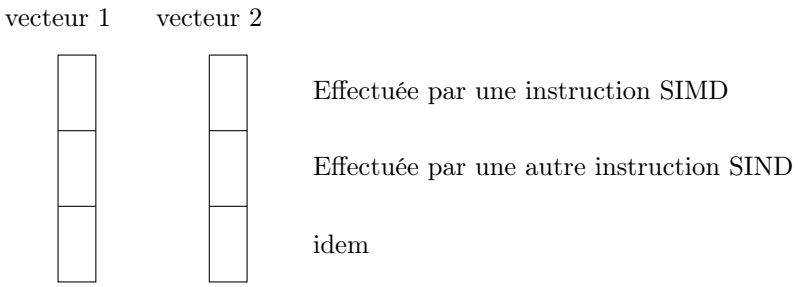
Les calculs pouvant trier fait de ces instructions sont ceux qui répètent le même calcul sur des données différentes.

(Domaines : codecs audio et vidéo)

Contraintes du SIND : il faut obtenir les types «complexes» sur lesquels on peut appliquer les algorithmes. Les vecteurs ou matrices sont adaptées.

Exemple :

La somme de 2 vecteurs :



Ces 3 tâches sont effectuée parallèlement

Le C++ introduit un type : valarray qui trie à partir de ces instructions. C'est une classe template. Elle permet de stocker un tableau de valeur.

Le paramètre générique est le type des valeurs.

valarray est dans la STL

Son implémentation dépend du compilateur utilisé (vc++, g++, clang++, icc et acc, STLport, ...)

Remarque : certaines implémentations utilisent une parallélisation du code.

Exemple :

Si le paramètre générique est un int, le code utilisera des instructions spécifiques aux entiers. Idem si c'est un double.

Remarque : #include <valarray>

18.1 Constructeur

Constructeur sans paramètre

```
valarray(size_t s);
vlarray(const T & val, size_t s);
valarray(T * val, size_t s);

valarray <double> v1;
valarray <int> v2(4); // 4 entiers
valarray <float> v3(3,7); // 7 floats valant 3
double tab[ ] = {1, 2, 8, 4};
valarray <double> v4 (tab, sizeof(tab)/sizeof(double));

valarray(const mask_array <T> & ma);
        indirect
        slice
        gslice
// Voir + loin l'utilisation
```

18.2 Modification de la taille

Exemple :

```
#include <iostream>
#include <valarray>
int main()
{
  valarray<double> v(4);
  cout << "taille␣:" << v.size() << endl;
  v.resize(8, 2.7); ———————————— // Complète les nouveaux éléments avec 2.7
  cout << "nouvelle␣taille" << v.size() << endl;
  return 0;
}
```

18.3 Accès aux éléments

```
T & operator[ ](size_t idx); // pour v[i] = 7
T operator[ ](size_t idx) const; // pour d = v[i]
```

18.4 Opérateurs arithmétiques

Toutes les opérations arithmétiques ($+$, $-$, $*$, $/$, $\%$) sont supportées.

Restrictions :

— Les 2 opérandes doivent être des valarray de même taille

— Le paramètre générique doit implémenter ces opérateurs

Les opérations sont faites éléments par éléments.

Opérateur d'incrémentations internes :

```
valarray <T> & operator *= (const T & val); //multiplication de tous les éléments par val
valarray <T> @ operator *= (const valarray<T> & va);
//multiplication éléments par éléments par va

idem +=, -=, /=, et %=, ^=, (xor) |=, <<=, >>=
```

18.5 Opérateurs mathématiques

Les fonctions mathématiques de la STL (ex : cos, sin, ...) peuvent être appliquées à un valarray. Si va est un valarray, cos(va) renvoie un valarray dont les éléments sont les cosinus des éléments de va.

Exemple :

```
void disp(const valarray<double> &va)
{
  for(int i = 0; i < va.size(); i++)
  {
    cout << va[i] << "␣";
    count << endl;
  }
}
int main()
{
  double v1[] = {1, 2, 3};
  double v2[] = {4, 5, 6};
  valarray <double> va1(v1, 3);
  valarray <double> va2(v2, 3);
  valarray <double> res = va1 + va2;
  disp(res); // vaut 5, 7, 9
  res = cos(va1);
  disp(res); // vaut cos(1), cos(2), cos(3)
  return 0;
}
```

18.6 Opérateurs de comparaison

< <= > >= == !=

/ ! Ça ne renvoie pas un bool, ça renvoie un valarray de bool

18.7 Opérateurs spécifiques (méthodes)

```
T sum() const;
T min() const;
T max() const;
valarray <T> shift(int) const; // décalage
valarray <T> cshift(int) const; // décalage circulaire
```

En paramètre, le nombre d'éléments décalés.

Si le paramètre est > 0, on décale vers la gauche

Si le paramètre est < 0, on décale vers la droite

Les éléments nouveaux ont pour valeur celle du constructeur de T par défaut.

Exemple :

```
int main{}
{
  double v1[5] = {1, 2, 3, 4, 5}
  valarray <double> va1(v1, 5);
  valarray <double> res(5);
  res = va1.shift(2);
  res = va1.shift(2);
  disp(res); // vaut 3, 4, 5, 0, 0
  res[3] = 0;
  res[4] = 3;
  disp(res); // vaut 3, 4, 5, 8, 9
  res = res.cshift(-2);
  disp(res); // vaut 8 9 3 4 4
  return 0;
}
```

Autre méthode valarray <T> apply(T fct(T)) const ; // Applique à chaque élément du valarray courant la fonction et renvoie un nouveau valarray

18.8 Sélection multiple d'éléments

operator[] permettent de sélectionner un élément. Il existe 4 autres moyens plus sophistiqués pour sélectionner des groupes d'éléments.

La syntaxe ressemble à celle de Scilab concernant l'exécution de sous-matrice.

(Rappel : v1 = [1:100], v2(1 :10 :100))

Base de cette sélection : choix d'un des 4 filtres disponibles.

18.8.1 Filtrage (ou sélection) par masque

Le plus simple des filtrages est un masque de booléens.

— On crée un valarray de booléens pour la sélection (de type mask_array)

— On utilise operator[] pour faire le filtrage

Exemple :

```
int main()
{
  int v[] = {1, 2, 3, 4, 5, 6, 7, 8};
  valarray <int> va(v,8);
  va [ (va3)==0] *= valarray<int> (2, va.size());disp(va); // vaut 1 2 \underline{6} 4 5 \underline{12} 7 8//
6 -> car 33 == 0
  // 12 -> car 63 == 0
```

18.8.2 Filtration par indexation explicite

Avantage : le masque n'a pas forcément le même nombre d'éléments que le valarray. Donc on fournit directement un valarray avec les indices que l'on souhaite.

Remarque : les indices peuvent ne pas être dans l'ordre croissant.

Exemple :

```
int main()
{
  int v[] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
  valarray <int> va(v,9);
  size_t idx[] = {2, 5, 7}; // les indices sélectionnés
  va[idx] *= valarray<int>(2, va.size());
  disp(va); // vaut 1 2 6 4 5 12 7 16 9
  return 0;
}
```

18.8.3 Filtration par indexation implicite

Les 2 précédentes filtrations peuvent être trop simples

Les 2 nouvelles filtrations sont utilisées quand il y a un "motif".

(comme [1 :10 :100] en scilab) Ce sont slice et gslice.

On indique l'indice de dépôt, le nombre d'indice et le pas d'incrémentation.

Slice : On utilise `operator[]`

```
valarray <T> operator [] (slice s) const
slice_array operator [] (slice s);
```

Exemple :

```
int main()
{
  int v[8] = {1, 5, 9, 4, 3, 7, 21, 32};
  valarray <int> va(v,8);
  slice sel(1, 3, 3); // 1 4 7 (très similaire à Scilab)
  // 1 : indice de départ
  // 1er 3 : nombre d'indices
  // 2e 3 : pas d'incrémentation
  va[sel] *= valarray <int>(2, va.size());
  disp(va); // vaut 1 10 9 4 6 7 21 64
}
```

gslice C'est un slice multidimensionnel.

```
gslice(size_t debut
    const valarray<size_t> & nbr;
    const valarray<size_t> & pas);
```

Ordre des indices : on fait varier en 1er les dernières variables caractérisées par la construction du gslice.

Exemple de gslice :

nbr : 2 3 5 1re étape

pas : 3 1 2 2e étape

debut : 2 3e étape

nbr 5 pas 2

→

nbr 3 pas 1 ↓

2	↓ 4	↓ 6 ↓	8 ↓	10
3	↓ 5	↓ 7 ↓	9 ↓	11
4	↓ 6	↓ 8 ↓	10 ↓	12

2 rectangles, le 1er est le 2e obtenu par pas de 3

5	7	9	11	13
6	8	10	12	14
7	9	11	13	15

Sortie de cube d'indice

Les indices peuvent apparaître plusieurs fois

Comme pour le slice, on utilise `operator[ ]` pour la sélection.

operator possibles sur les (g)slices mask_array et indexed_array

operator : =

∗ =

/ =

% =

+ =

− =

=

& =

| =

<<=

>>=