C++ 2011 avancé

Table des matières

1 Extensions au langage C

A) Commentaires

```
/* commentaire C. avec des  // /// //
   sur plusieurs lignes */
// Commentaire C++, jusqu'à /* la fin */ de la ligne.
```

B) & et const

Esperluète : & (ampersand) ≡ alias sur une variable ou un objet.

```
int  i;
int& ri = i;   // déclaration d'une référence sur i = alias
int* pi = &i;  // pointeur sur i, &i renvoie ici l'adresse de i
int& rpi = *pi // *pi déréférence le pointeur et renvoie une référence
// ici i, ri, *pi et rpi référencent une même et unique variable int.
int k = *pi;   // k est un int indépendant, initialisé à la valeur de i
```

⚠ Une référence ne peut être `NULL`.

Passage de paramètres

```
void e(      T  i ) ;  // i est copié, la copie est utilisée
                       // par la fonction.
void f(      T& i ) ;  // les modifications sur i seront
                       // valables globalement.
void g(const T  i ) ;  // i est copié, la copie n'est utilisée
                       // qu'en lecture.
void h(const T& i ) ;  // i ne peut pas être modifié.
```

Déclarations constantes

```
int i;                          // Entier ordinaire
const int j = 13;               // Entier non modifiable
int * ip;                       // Pointeur
int * const cp = &i;            // Pointeur non modifiable
const int * cip;                // Pointeur sur un entier non modifiable
const int * const cicp = &ci; // Pointeur non modifiable
                                // sur un entier non modifiable
```

👍 Mnémotechnique : mettre une) avant l'*

C) Surcharge et arguments par défaut

Surcharge des fonctions : contrairement au C, deux fonctions peuvent avoir le même nom, pourvu que leurs signatures soient différentes.

```
int max(int i, int j) { return (i>j) ? i : j; }
char* max(char * s, char * t) {
  return (strcmp(s,t)>0) ? s : t; }
```

Argument par défaut d'une fonction : on peut fournir une valeur par défaut pour les derniers paramètres d'une fonction :

```
int ajouter(int a, int b=1) { return a+b ; }
ajouter( 2 ) ; // renvoie 3
```

D) New et Delete

`new` est un opérateur (donc un mot réservé), redéfinissable comme on le verra en section 9, qui calcule automatiquement la taille du type sur lequel il agit, appelle `malloc`, et renvoie un pointeur de type adéquat, épargnant ainsi le changement de type. Voici un exemple :

```
double *dp = new double;
// équivalent à : // double *dp = (*double) malloc(sizeof(double));
```

Pour un tableau, il suffit de préciser sa dimension :

```
double *table = new double[dim];
```

Dans une déclaration de tableau normale, dans la pile, la dimension du tableau doit être constante.
Au contraire, ici, `dim` peut être n'importe quelle expression, dynamique. Cependant, une taille explicite doit être déclarée, il ne faut pas laisser les crochets vides. Ainsi la mémoire supplémentaire est allouée, dans le tas.
Ce mécanisme fonctionne aussi pour les tableaux multidimensionnels :

 Lorsque les objets ainsi obtenus ne sont plus utiles, on peut appeler l'opérateur `delete` (affublé de crochets pour le cas des tableaux) qui les détruit, i.e. libère la mémoire ainsi allouée :

```
delete dp;
delete [] table;
```

 Comme dans le cas de free, le pointeur n'est pas remis à zéro, et il ne faut appeler `delete` que sur un bloc alloué par `new`.

2 Encapsulation : espace de nommage

`namespace` permet de définir un espace de nommage, i.e. un module (un paquetage en Ada ou en Java) pour structurer un programme en «unités logiques».

A) Spécification

```
namespace pile_de_char { // interface dans un fichier pile.h
  void empiler( char ) ;
  char depiler() ;
  bool est_vide() ;
}
```

 Il existe un espace de nommage anonyme, qui permet d'éviter des conflits de noms entre fichiers : il suffit de déclarer des fonctions dans un espace de nommage sans identificateur :

```
namespace
{
  //functions
}
```

 Il est possible de renommer des namespace avec : `namespace <nouveau> = <ancien>`.

B) Utilisation

Un fichier `test-pile.C` peut alors utiliser l'interface :

```
#include "pile.h"
#include <iostream>

int main() {
  pile_de_char::empiler( 'x' ) ;
  if (pile_de_char::depiler() != 'x')  std::cerr <<  "impossible." ;
}
```

Le préfixe `pile_de_char::` indique que `empiler()` et `depiler()` sont ceux du module `pile_de_char`.

C) Implémentation

L'implémentation peut être donnée lors de la spécification (dans le fichier `pile.h`) :

```
namespace pile_de_char {
  const int taille_max = 200 ;
  char la_pile [ taille_max ] ;
  int sommet = 0 ;

  void empiler ( char c ) {
    // Attention, pas de contrôle de débordement
    la_pile[ sommet ] = c ;
    ++sommet ;
  }

  char depiler() {
    // Attention, pas de contrôle de débordement
    return la_pile[ --sommet ] ;
  }

  bool est_vide() {
    return sommet == 0 ;
  }
}
```

Il est possible de donner l'implémentation d'une fonction séparément (ici dans `pile.C`), en préfixant le nom de la fonction par l'identificateur de son espace de nommage :

```
#include "pile.h"
namespace pile_de_char {
  const int taille_max = 200 ;
  char la_pile [ taille_max ] ;
  int sommet = 0 ;
  int depiler() ;
  void empiler ( char c ) ;
  bool est_vide ( ) ;
}

int pile_de_char::depiler() {
    // Attention, pas de contrôle de débordement
    return pile_de_char::la_pile[ --pile_de_char::sommet ] ;
}

void pile_de_char::empiler ( char c ) { ... } ;
bool pile_de_char::est_vide ( ) { ... } ;
```

D) Using namespace

Il est possible de spécifier que tous les noms doivent également être cherchés dans un `namespace` donné. Le préfixage par `pile_de_char::` peut alors être omis si il n'y a pas d'ambiguïté.

```
int main() {
    pile_de_char::empiler('a');
    using namespace pile_de_char;
    depiler();
    return 0;
}
```

E) Koenig lookup

Une fonctionnalité intéressante des espaces de nommage est le "Koenig lookup" : lors d'un appel à une fonction le type des arguments permet parfois de lever l'ambiguïté. Ceci peut être très utile pour faire des fonctions génériques modifiables par la suite (puisque un `namespace` est une structure ouverte à laquelle il est toujours possible d'ajouter des données ou des méthodes).

```
#include <iostream>
#include <list>
#include <vector>

namespace Listes {
        struct TypeDeDonnees { std::list<int> valeurs; };
        void affiche(TypeDeDonnees l) { std::cout << "Listes"; }
}

namespace Vecteurs {
        struct TypeDeDonnees { std::vector<int> valeurs; };
        void affiche(TypeDeDonnees l) { std::cout << "Vecteurs"; }
}

int main() {
        Vecteurs::TypeDeDonnees v;
        // Vecteurs::affiche est trouvée, même sans ‘‘using’’,
        // grâce à son paramètre de type Vecteurs::TypeDeDonnees.
        affiche( v ) ;

        return 0;
}
```

3 Classes

Définition d'un nouveau *type* au sens C++ : ensemble de *méthodes* (fonctionnalités) auxquelles sont ajoutés des *membres* (données).

```
class Point {
   // Deux membres, les coordonnées du point, inaccessibles de l'extérieur
   int _x, _y;

// Interface utilisateur, accessible de l'extérieur
public:

   // Méthodes d'accès (en lecture) aux membres
   int abscisse()  { return _x; }
   int ordonnee() { return _y; }

   // Méthodes de modification des membres
   void affect_x(int x) { _x = x; }
   void affect_y(int y) { _y = y; }
};
```

Une classe est une implémentation d'un type. Une variable de ce type (une instance de la classe) est appelée un *objet*.

A) public, private, protected

Par défaut, dans une classe, seules les méthodes de cette classe peuvent accéder aux autres méthodes et aux membres. On dit que ceux-ci sont en accès privé. Pour être utilisable, une classe doit donc rendre publiques certaines méthodes.

La spécification de `public:` (resp. `private:`) à l'intérieur de la classe change l'accès par défaut pour les membres et fonctions situés après cette déclaration. Plusieurs `public:` (resp. `private:`) peuvent apparaître dans une classe.

Troisième accès : `protected :` → l'accès est autorisé aussi aux classes dérivées, mais pas pour les utilisations externes.

B) class ou struct ?

Il existe une autre déclaration de classe : `struct` (accès public par défaut pour tous les membres et méthodes) ≈ `class` (accès privé par défaut pour tous les membres et méthodes).

```
struct Point {
   // CETTE FOIS-CI les coordonnées du point sont accessibles de l'extérieur
   int _x, _y;
private:
   // Ce qui suit, jusqu'au prochain "public:" n'est pas accessible
   ...
};
```

C) Constructeurs

Utilisés à la déclaration de l'objet pour le construire (affecter les membres, allouer la mémoire, etc.) à partir de différents paramètres.

```
class Point {
  int _x, _y;

public:
   // Intérêt de la notation _x
   Point() : _x(0), _y(0) { }
   Point(int x, int y) : _x(x), _y(y) { }
   Point(const Point& P) : _x(P._x), _y(P._y) { }

   // Autres méthodes ...
   int abscisse()  { return _x; }
   int ordonnee() ;
};

Point X;      // appel du constructeur vide
Point M(3,4); // appel du constructeur sur les entiers
Point P(M);   // appel du constructeur de recopie physique
```

⚠ Par défaut, si AUCUN autre constructeur n'est implémenté, les constructeurs vide et de copie sont générés automatiquement, et constitués des appels aux constructeurs des différents membres.

👍 La notation `: membre(valeur)` permet d'initialiser certains membres en appelant leur constructeur par copie, ce qui est en général plus efficace que d'utiliser l'affectation (le constructeur vide ayant été appelé de toute manière).

Par ailleurs la syntaxe de construction des attributs hors du corps du constructeur est la seule façon d'initialiser des attributs `const` :

```
struct Couple {
  const int _a;
  int _b;

  // _a=a; est impossible dans le corps du constructeur
  // car l'affectation tenterait de modifier _a, qui est const
  Couple(int a, int b) : _a(a) { _b=b; }
};
```

D) Destructeur

Appelé automatiquement quand l'objet ne sera plus utilisé (fermeture d'accolade }), `delete` ou encore `delete []`.

```
class Tableau {
   int * _d;
public:
   Tableau(int s = 10) { _d = new int[s]; }
   ~Tableau() { delete [] _d; }
};

{  Tableau t; // ici de taille 10 par défaut
     ...
} // Appel de tous les destructeurs des objets déclarés dans ce groupe
```

⚠ Par défaut, une classe implémente un destructeur qui ne fait rien.

E) Pourquoi programmer objet ?

Constructeurs et Destructeur ⇒ Intérêt majeur de la programmation objet : la gestion de la mémoire est faite *par le programmeur une fois pour toutes* et non *par l'utilisateur à chaque utilisation*!

4 Optimisation par annotations

A) `const`

Pour permettre au compilateur de mieux optimiser le code, le mot-clef `const` peut être ajouté à une méthode pour indiquer que celle-ci s'engage à ne pas modifier les membres de la classe (vérifié à la compilation).

```
class Point {
   int _x, _y;
public: // Interface utilisateur
   int abscisse() const { return _x; }
   void affect_x(int x)  { _x = x; }
};
```

Les méthodes sans effet de bord (par exemple celles de lecture des membres privés) devraient le plus souvent être déclarées `const`.
⚠ En particulier, les méthodes `const` ne peuvent appeler que d'autres méthodes `const` de la classe.

B) `::` et `inline`

Il est possible de définir une méthode en dehors de la définition de la classe : il suffit de préciser la classe pour laquelle la méthode déclarée doit être implémentée.

```
class Point {
   int _x, _y;
public: // Interface utilisateur
   Point();
   int abscisse() const;
   void affect_x(int);
};

// Probablement dans un fichier point.cpp
Point::Point() : _x(0), _y(0) {  }
int Point::abscisse() const { return _x; }
void Point::affect_x(int x) { _x = x; }
```

Les méthodes définies à l'intérieur de la classe (*inline*) peuvent avoir leur corps recopié à chaque appel pour accélérer l'exécution. Ce n'est pas le cas de celles définies à l'extérieur.

Une méthode définie à l'extérieur, comme toute fonction, peut aussi être déclarée *inline*. Le compilateur ESSAIERA de recopier le corps si le code n'est pas trop long.

```
inline Point::Point() : _x(0), _y(0) {  }
inline int Point::abscisse() const { return _x; }
inline void Point::affect_x(int x) { _x = x; }
inline void f(int i) { return 2*i+1; }
```

5 this

Dans une méthode, ou un constructeur, le mot-clef this est un pointeur sur l'objet considéré.

```
Point& Point::copie(const Point& P) {
  // Ici la méthode recopie le point P dans l'objet appellant
  // puis retourne l'objet appelant

  if (this != &P) {
    // Ce n'est pas la peine de se copier soit même.
    // Attention, ce test vérifie l'égalité en adresse mémoire
    // et non l'égalité mathématique (celle des membres).
    _x = P._x;
    this->_y = P._y;
  }
  return *this;
}
```

 En pratique cela revient à dire que l'objet appelant une méthode est également implicitement passé en paramètre de la méthode.

6 friend

A) Méthode friend

Ce que l'on pourrait prendre pour une méthode déclarée friend est en fait une fonction classique : ce n'est absolument pas une méthode de la classe ! Le mot-clef friend permet simplement d'indiquer à la classe que cette fonction pourra accéder à toutes les données privées (membres ou méthodes). L'ambiguïté vient du fait que cette indication passe pour une déclaration.

```
class Point {
   int _x, _y;
public:
   friend void affiche(const Point& M) { printf("(%l,%l)",M._x,M._y); }
   void affiche_interne() { printf("(%l,%l)",_x,_y); }
};
...
Point M(3,4);
affiche( M ) ;       // affiche est une fonction globale classique
M.affiche_interne(); // affiche_interne est une méthode
```

B) Classe friend

De même, une classe peut être indiquée friend à l'intérieur d'une autre classe. Là encore cela ressemble à une déclaration précédée du mot-clef friend. Ainsi les méthodes de cette classe (implémentées ultérieurement) pourront manipuler les données privées de la classe dont elle est «amie».

7 static et extern

A) Accessibilité au niveau global

Le mot-clef `extern` est utilisé pour spécifier qu'une variable ou une fonction est déclarée dans un fichier différent. Quand le mot-clef `extern` est utilisé ; le compilateur n'allouera pas de mémoire pour cette variable.

Au contraire, pour limiter la portée d'un nom à l'unité courante, il faut utiliser le mot-clef `static`, ou alors utiliser un espace de nom anonyme ("nameless namespace").

```
      int a  ;//crée ici, implicitement   accessible depuis un autre source.
extern int b=0;//crée ici, explicitement   accessible depuis un autre source.
static int d  ;//crée ici, explicitement inaccessible depuis un autre source.
extern int c  ;//existe dans un quelconque source (peut-être le même).
```

B) static dans une classe

Le mot-clef `static` permet de définir des membres ou des méthodes «globales», tout en limitant leur portée. Un membre `static` est identique pour toutes les instances de la classe. Toute modification par un objet est reconnue par tous les autres objets.

Une méthode est normalement appliquée à (et par) un objet. Une méthode `static` est au contraire une méthode globale à la classe, non spécifique à un objet, faisant naturellement partie de la classe et offrant généralement un service partagé.

```
class Utilisateur {
  public:
    Utilisateur(int uid);
    char* getPassword() const { return password; }

  private:
    static const int TAILLE_PASSWORD = 6;
    static char* randomPassword();

    int userId;
    char* password;
};
```

On accède aux membres et méthodes statiques via l'espace de nom créé par la classe et la notation `nomClasse::`.

⚠ Bien qu'acceptée, la notation `object.membreOuVariableStatique` est déconseillée.

```
Utilisateur::Utilisateur(int uid) : userId(uid)
{
  password = Utilisateur::randomPassword();
}

char* Utilisateur::randomPassword()
{
  char* tmppass = new char[Utilisateur::TAILLE_PASSWORD];
  for (int i=0; i<Utilisateur::TAILLE_PASSWORD; ++i)
    tmppass[i] = 'a' + random() * 26 / INT_MAX;
  return tmppass;
}
```

L'utilisation de `static` est relativement rare mais parfois parfaitement adaptée. Par exemple, une ıconstante propre à une classe sera ainsi généralement déclarée comme `static const`.

⚠ Les méthodes et membres d'une classe, même static, sont visibles partout par défaut et ne peuvent donc être dissimulées aux autres unités qu'en déclarant la classe dans un espace de nom anonyme.

8 Exceptions

A) try, catch, throw

Les exceptions sont un moyen de réagir à des circonstances exceptionnelles (comme une erreur) dans un programme en transférant le contrôle à une fonction dédiée au traitement de cette circonstance.

Pour pouvoir effectuer ce transfert, la portion de code doit être incluse dans un bloc `try`. Si une exception est levée le contrôle est passé *immédiatement* au bloc de traitement, sinon tous les blocs de traitement sont ignorées.

Une exception est levée par le mot-clef `throw`, un bloc de traitement est déclarée par le mot-clef `catch` qui doit être placé immédiatement après le bloc `try`, ou immédiatement après un précédent bloc `catch`.

```
#include <iostream>
int main () {
  try {
    throw 42;
    // ce qui suit jusqu'à l'accolade fermante n'est pas executé ...
    // ...
  } catch (float f) {
    // ce bloc de traitement n'est pas utilisée car le type de son
    // paramètre ne correspond pas à celui de la valeur levée
    // ...
  } catch (int i) {
    std::cerr << "Une exception " << i << " a été levée" << std::endl;
  }
  return 0;
}
```

 Il est possible d'imbriquer autant de blocs `try` que nécessaire, de lever des exceptions dans les blocs de traitement, etc.

B) ...

Lorsque qu'une exception est levée avec un paramètre de type `T`, si aucun bloc de traitement `catch` n'a le paramètre `T`, aucun traitement n'est appliqué et l'exception est propagée au bloc supérieur (le programme est abandonné s'il n'y a pas de bloc supérieur). Il est néanmoins possible d'ajouter un traitement prenant le contrôle pour n'importe quel paramètre (il est donc a priori le dernier bloc `catch`.)

```
try{ /* etc. */ }
catch (float f) { /* etc. */ }
catch (int i)   { /* etc. */ }
catch (...)     { // traitement utilisé si une exception est levée de
                  // type différent de int ou float
                  /* etc. */ }
```

C) Restriction de levée d'exception

En déclarant une fonction, il est possible de restreindre les type d'exceptions qu'elle peut lever par le mot-clef `throw` suivi d'une liste de types autorisés entre parenthèses.

```
int f (int a) throw(int,char); // exceptions de type int ou char seulement
int g (int a) throw();         // aucune exception autorisée
int h (int a);                 // toute exception autorisée
```

D) Exceptions standard

 La bibliothèque standard fournit un type d'exceptions spécialement conçu pour être levé : `exception`. Cette classe possède principalement une méthode `what()` qui renvoie une chaîne de caractères décrivant l'exception levée.

```
class exception {
  public:
    exception() { }
    virtual ~exception();
    virtual const char* what() const;
};
```

Plusieurs type dérivés d'exception sont également fournis :

`bad_alloc`	levée par `new` en cas de problème d'allocation
`bad_cast`	levée par `dynamic_cast` en cas de problème de référence
`bad_exception`	levée quand aucun `catch` n'a le paramètre requis
`bad_typeid`	levée par `typeid`
`ios_base::failure`	levée par des `iostream`

9 Opérateurs

Un opérateur est une fonction ou une méthode avec un appel particulier.

A) Exemple

```
class Complexe {
  float _re, _im;
public:
    // une méthode peut être un opérateur ...
  Complexe& operator+= (Complexe x) {
    _re += x._re;
    _im += x._im;
    return *this;
  }
};

  // ... tout comme une fonction classique
Complexe operator+ (Complexe x, Complexe y) {
  Complexe r = x;
  return r += y;
}

void f(Complexe x, Complexe y, Complexe z) {
  Complexe r1 = x + y + z; // r1 = operator+(x, operator+(y,z) )
  Complexe r2 = x;
  r2 += y;                 // r2.operator+=( y )
  r2 += z;                 // r2.operator+=( z )
}
```

Les opérateurs suivants peuvent être (re)définis :

```
+   -   *   /   %   ^   &   |   ~   !   =   <   >
+=  -=  *=  /=  %=  ^=  &=  |=  <<  >>  ==  <<= >>=
&&  ||  !=  >=  <=  ++  --  ->* ,   ->  []  ()
new     new[]   delete  delete[]
```

⚠ les opérateurs `::` (résolution de nom), `.` (sélection de membre) et `.*` ne peuvent cependant pas être redéfinis.

⚠ l'opérateur d'affectation (`operator=`) est prédéfini et appelle par défaut les opérateurs d'affectation des membres de l'objet. Il peut être redéfini.

⚠ Dans le cas où la classe possède des pointeurs, l'opérateur `=` ainsi que le constructeur par copie doivent être redéfinis si on ne souhaite pas partager les objets pointés.

B) Opérateur parenthèses

L'opérateur parenthèses permet de donner un type à une fonction (de définir une fonction-classe). Ainsi les fonctions peuvent être manipulées plus facilement (comme des objets).

```
class Incrementeur {
  int _inc;
public:
  Incrementeur(int i) : _inc(i) {}
  int operator() (int i) { return i+_inc; }
};

main() {
  Incrementeur plus_un(1), plus_deux(2); // Appel du constructeur
  int a = plus_deux(3);  // a <-- 5, par l'appel de la méthode ()
}
```

⚠ Contrairement aux autres opérateurs, l'opérateur parenthèses a la particularité de pouvoir être défini avec un nombre quelconque d'arguments. Ce doit être une méthode (i.e. cela ne peut pas être une fonction `friend`).

C) Opérateurs d'incrémentation et de décrémentation

operator++ et operator−− sont ambigus : s'agit-il de la définition de l'opérateur suffixé ou préfixé ? Un paramètre artificiel `int` dans la définition permet de lever cette ambiguïté.

```
class Pointeur {
  int * p;
public:
  Pointeur& operator++ ();       // préfixe
  Pointeur  operator++ (int);    // suffixe
  Pointeur& operator-- ();       // préfixe
  Pointeur  operator-- (int);    // suffixe

  int operator*() { return *p; } // déréférencement
};
```

ou alors comme fonction globale en dehors de la classe

```
class Pointeur {
  int * p;
public: int operator*() { return *p; }   // déréférencement
};

Pointeur& operator++ (Pointeur&);        // préfixe
Pointeur  operator++ (Pointeur&,int);    // suffixe
Pointeur& operator-- (Pointeur&,);       // préfixe
Pointeur  operator-- (Pointeur&,int);    // suffixe
```

D) Conversions

Les conversions sont possibles dans une classe, par constructeur dans un sens, par opérateur de type (transtypage) dans l'autre, le constructeur étant prioritaire.

La syntaxe d'opérateur de transtypage consiste en le mot-clef `operator`, sans type de retour, suivi du nom du type ou de la classe vers laquelle s'opère la conversion.

```
struct GaussInt {
  int a, b;
  GaussInt () {}
  GaussInt(int f, int g) : a(f), b(g) {}
  GaussInt(const Complexe& c) ;
  operator Complexe() ;
};
```

Pour l'implémentation, la conversion à la C peut être utilisée : `float re = (float)a`.

En C++ deux syntaxes sont néanmoins préférées, car plus précises :
– `float re = float(a)` est un appel au constructeur de float, de paramètre un entier.
– `float re = static_cast<float>(a)` est un transtypage plus contraint, donc plus précis.

En effet, en C++, il existe trois autres transtypages possibles :
– `dynamic_cast` qui permet de réaliser des conversions valides, vérifiées à l'exécution, mais que le compilateur n'arrive pas à faire (car il ne connaît pas encore le véritable type – un template, par exemple –).
– `reinterpret_cast` est utilement atroce (permet par exemple de transformer un int en une adresse de pointeur).
– `const_cast` permet d'ajouter ou de retirer un qualificatif `const` (le comportement ensuite peut-être non spécifié).

```
GaussInt::GaussInt(const Complexe& c) : a( int(c.re) ), b( int(c.im) ) {
  std::cerr << "Cstor GaussInt from Complexe" << std::endl;
}

GaussInt::operator Complexe() {
  std::cerr << "Cast GaussInt to Complexe" << std::endl;
  return Complexe( float(this->a), float(this->b) );
}
```

```
int main() {
  Complexe c (1.2, 3.4), d;
  GaussInt g (5  , 7  ), f;

  d = (Complexe)g;
  d = Complexe(g);
  d = static_cast<Complexe>(g);

  f = (GaussInt)c;
  f = GaussInt(c);
  f = static_cast<GaussInt>(c);

  return 0;
}
```

s'exécute en

```
Cast GaussInt to Complexe
Cast GaussInt to Complexe
Cast GaussInt to Complexe
Cstor GaussInt from Complexe
Cstor GaussInt from Complexe
Cstor GaussInt from Complexe
```

10 Entrées et sorties par flots

La librairie standard <iostream> permet de faire des entrées-sorties par flots. Les flots de sorties (de type `std::ostream`) sont `std::cout` et `std::cerr` pour le standard et l'erreur respectivement. Le flot d'entrée (de type `std::istream`) est `std::cin`. Comme l'indique le préfixe `std::`, tous ces objets sont dans le namespace `std`.

```
int numero;
std::cout << ``Entrez un nombre plus petit que 10 : '';
std::cin >> numero;
if (numero > 10)
  std::cerr << `` *** Erreur nombre trop grand !!!'' << std::endl;
else
  std::cout << `` Merci.'' << std::endl;
```

Il est possible de définir les opérateurs << et >> pour n'importe quel type et ainsi chaque objet peut être facilement manipulé par les flots.

```
class Complexe {
  float _re, _im;
  ...

  friend std::ostream& operator<< (std::ostream& o, const Complexe& c);

  friend std::istream& operator>> (std::istream& i, Complexe& c) ;
};

std::ostream& operator<< (std::ostream& o, const Complexe& c) {
  return o << c._re << '+' << c._im << 'i';
}

std::istream& operator>> (std::istream& i, Complexe& c) {
  char t;
  return i >> c._re >> t >> c._im >> t;
}
```

 Noter que l'objet `ostream` (resp. `istream`) retourné par référence par ces opérateurs permet leur enchainement.

 chaque << est un appel de fonction. Ces opérateurs sont donc très commodes mais parfois moins efficaces que les entrées-sorties C (`scanf`, `printf`) qui peuvent ne faire qu'un seul appel pour plusieurs objets.

11 Héritage

Une classe `B` peut dériver d'une classe mère `A` : tout objet de type `B` hérite alors des attributs et fonctions membres d'un objet parent. En C++, l'héritage peut être multiple : une classe fille peut avoir plusieurs classes mères.

A) Types d'héritage

Comme pour les membres, une classe mère `A` peut être dérivée de manière `public`, `protected` ou `private` :

```
class X: public A { ... } ;    // X est un sous-type de A.

class Y: protected A { ... } ; // Héritage d'implémentation: Y et Z
class Z: private A { ... } ;   // restreignent l'interface de A.
```

- `X: public A` : tous les membres publics et protégés de `A` peuvent être utilisés par n'importe quelle fonction manipulant un `X`. C'est la dérivation la plus commune : "`X` *est un* `A`".
  Tout objet de type `X*` peut être converti en `A*`.
- `Y: protected A` : les membres publics et protégés de `A` ne peuvent être utilisés que par les fonctions membres et amies de `Y` ou des classes `YY` qui dérivent de `Y`.
  Seules ces fonctions peuvent convertir un objet de type `Y*` en `A*`.
- `Z: private A` : les membres publics et protégés de `A` ne peuvent être utilisés que par les fonctions membres et amies de `Z`.
  Seules ces fonctions peuvent convertir un objet de type `Y*` en `A*`.

B) Construction de l'objet parent

Les constructeurs d'une classe mère ne sont pas hérités. Lors de la construction d'un objet d'une classe dérivé, il est possible de construire l'objet parent en appelant le constructeur de la classe mère. C'est la seule façon de construire les attributs hérités.

```
class A {
  private:
    int a ;
  public:
    A( int n ) { a = n ; }
};

class B: public A {
  public:
    B( int n ) : A(n) { /*...*/ }
};
```

C) Fonctions membres virtuelles

Une classe dérivée peut remplacer les fonctions membres de sa classe mère. En C++, on parle de membre virtuel : `virtual`.

```
struct A {
    int zero() { return 0 ; }
    virtual int un() { return 1 ; }
};
struct B: public A {
    int zero() { return 1 ; }
    int un() { return 0 ; }
};
int main() {
  A a ;
  B b ;
  A &ref_A = a ;   ref_A.zero() ;  // retourne 0
                   ref_A.un() ;    // retourne 1
  A *pt_A ;
  pt_A = &b ;    // possible car la dérivation est publique.

  pt_A->zero() ; // retourne 0: la fonction est prise dans A car pt_A est
                 //             un pointeur sur un objet de type A
                  //             et zero n'est pas virtuelle.

  pt_A->un() ;   // retourne 0: la fonction est virtuelle et prise dans B
                  //             car pt_A pointe en fait sur un B.

  pt_A = &a ;    pt_A->zero() ; // retourne 0.
                 pt_A->un() ;   // retourne 1.
}
```

D) Polymorphisme par sous-typage (dérivation)

L'idée est de partir d'un type et de le modifier. Par exemple, en C++ on peut créer une classe de base, puis faire des classes dérivées, modifiant ou étendant son comportement.

```
struct Forme { virtual float Aire() = 0; };
struct Carre : public Forme {
   virtual float Aire() { return m_cote*m_cote; }
private:
   float m_cote;
};
struct Cercle : public Forme {
   virtual float Aire() { return 3.1415926535*m_rayon*m_rayon; }
private:
   float m_rayon;
}
```

Grâce aux fonctions virtuelles, on peut faire un algorithme en n'utilisant que la classe de base qui va automatiquement appeler les fonctions des classes dérivées :

```
float AireTotale(Forme* tabl, int nb) {
   float s=0;
   for(int i = 0; i<nb; ++i) {
      s+= tabl[i].Aire(); // par la table des fonctions virtuelle
   }
   return s;
}
...
Forme* tableau[3] = {new Carre, new Cercle, new Carre};
AireTotale(tableau,3);
```

En proposant d'utiliser un même nom de méthode pour plusieurs types d'objets différents, le polymorphisme permet une programmation beaucoup plus générique. Le développeur n'a pas à savoir, lorsqu'il programme une méthode, le type précis de l'objet sur lequel la méthode va s'appliquer. Il lui suffit de savoir que cet objet implémentera la méthode de façon adaptée.

Si une fonction virtuelle n'est pas redéfinie, c'est celle de la classe mère qui est utilisée.

E) Fonctions virtuelles pures

Une classe mère peut définir une fonction membre sans en donner d'implémentation : cette implémentation est laissée aux classes filles. On parle alors de fonction virtuelle pure.

```
class A {
  public:
    virtual void f (int i) = 0 ; // f est virtuelle pure.
};

class B: public A {
  public:
    void f ( int i) { /* Implémentation de f */ }
};
```

Une classe dont un membre au moins est virtuel pur est dite abstraite ; on ne peut créer d'objets de cette classe (seulement des pointeurs peuvent être manipulés).
Une classe dont tous les membres sont virtuels purs sert à définir une interface.

F) Héritage multiple

Une classe peut hériter de plusieurs classes. La gestion des ambiguïtés se fait explicitement en utilisant `::` ou en indiquant les classe mères moins prioritaires par `virtual`.
Dans le cas d'une telle dérivation virtuelle, les attributs hérités d'une même classe mère virtuelle suivant deux branches d'héritage distinctes ne sont pas dupliqués. Le constructeur d'une classe mère virtuelle est appelé une seule fois lors de la construction d'un objet dérivé (il n'y a qu'un objet de la classe mère).

```
class A {
  public:
    int a;
    // ...
};

class B: public virtual A { // ... };

class C: public virtual A { // ... };

class D: public virtual B, public virtual C {
  // L'objet n'a qu'un objet parent de type A (donc un seul attribut a)
}
```

12 Généricité

Une classe ou une fonction peut être générique par rapport à un type ou une valeur entière. La déclaration de la classe doit alors être préfixée par `template`. Un fonctionnement générique peut ainsi être appliqué à des types différents sans duplication de code.

A) Déclaration de classe générique

La déclaration se fait en précédant la classe d'une déclaration `template`, qui spécifie les types, ou variables, qui ne seront connus qu'à la compilation.

```
// Une pile d'objets générique, de taille maximale 200 par défaut.
template <typename T, int taille_max = 200>
  // Le paramètre entier max_size a la valeur 200 par défaut.
struct  pile {
    typedef T value_type;
  private:
    T* la_pile ;
    int sommet ;

  public:
    pile() : sommet(0) { la_pile = new T [taille_max] ; }
    ~pile() { delete [] la_pile ; }

    void empiler ( T c ) { la_pile[ sommet++ ] = c ; }

    // ...
};
```

La déclaration d'un objet se fait alors avec le nom de classe suivi des paramètres `template`, cette fois connus, entre $<$ et $>$.

⚠ dans le cas de template imbriqués, séparer les $>$ par des espaces, pour ne pas confondre avec les opérateurs d'entrées-sorties.

```
int main() {
  // Déclaration d'un type pile composé d'au plus 26 char
  typedef pile< char, 26 > Alphabet;

  Alphabet p ; // instanciation d'une pile de 26 caractères.
  for (Alphabet::value_type c = 'a'; c < 'z'; ++c) {
     p.empiler( c ) ; }
}
```

☝ Le compilateur vérifiera à la compilation que le type passé en paramètre `template` possède bien les méthodes utilisées dans la classe ou la fonction générique.

B) typename

Le mot-clef `typename` indique que le mot suivant doit être considéré comme un type. C'est le cas dans une déclaration `template` (où `typename` peut aussi être remplacé par `class`) et dans les cas où il y aurait ambiguïté ou si le type n'est pas encore instancié (paramètre d'un autre type, lui-même template).

```
template <class P, int taille_max = 200>
class SacADos {
   // P::value_type n'est pas encore un type puisque T, n'a pas été instancié
   // Il faut donc l'indiquer au compilateur
   typedef   typename P::value_type   T_values;

   ...
};
```

C) Fonctions et méthodes génériques

`template` permet aussi de déclarer des fonctions génériques, voire des fonctions membres génériques d'une classe générique. Cependant, ici, la spécification entre $<$ et $>$ n'est pas requise à l'appel car déduite des paramètres effectifs.

```
class  pile {
     template <typename T>
     void empiler(const T& a);
     ....
};


template<typename X>
void explorer(const X& p) { ... };

int main() {

  pile p ; // instanciation d'une pile.

  char c = 'a';

  // Ces appels sont valides.
  // Dans le premier cas ``char'' est déduit du type de c.
  p.empiler(c);
  p.empiler<char>(c) ;

  explorer( p );
  explorer<pile>( p );
}
```

D) Spécialisation

Tout objet `template` peut être spécialisé. C'est-à-dire que pour une classe particulière, par exemple, on peut définir un corps différent pour l'objet template. La spécialisation se fait en indiquant `template<>`.

```
// La même version pour tous les objets de type X
template<typename X>
void afficher(const X& p) {
   std::cout << p << std::endl;
}

// Pour les int on peut aller plus vite avec printf
template<>
void afficher<int>(const int& p) {
    printf(``%d\\n'',p);
}
```

E) Un peu de template meta-programmation

Les templates définissent en fait un langage en lui-même. En outre comme ils sont traités avant toute compilation, il est possible de les programmer et de faire calculer le compilateur ! Prenons par exemple le calcul de la fonction factorielle et faisons calculer le compilateur de la manière suivante :

```
template<int N> struct Factorielle {
        static const int valeur = Factorielle<N-1>::valeur * N;
};

template<> struct Factorielle<1> {
        static const int valeur = 1;
};

#include <iostream>

int main(int argc, char ** argv) {
        std::cout << Factorielle<6>::valeur << std::endl;
        return 0;
}
```

Dans cet exemple, les seuls calculs font intervenir uniquement des constantes. Ainsi, comme aucune instance de la classe n'est construite, aucun code n'est actuellement produit ! Au contraire, le calcul est effectué par le compilateur qui remplace au fur et à mesure (même récursivement comme ici) les valeurs des constantes, directement dans le code généré.

Voici par exemple un extrait du code assembleur produit par "g++ -S factorielle.C" sur un PIII, la valeur de 6! = 720 est copiée directement pour le `cout`, sans calcul :

```
.LCFI2:
        andl    $-16, %esp
        movl    $0, %eax
        subl    %eax, %esp
        movl    $720, 4(%esp)
        movl    $_ZSt4cout, (%esp)
```

⚠ Attention, la récursivité des template est limitée. Avec `g++`, il est possible de l'augmenter avec l'option `-ftemplate-depth-NN`, où NN est la profondeur désirée.

Les possibilités sont très grandes : outre les calculs de constantes, tous les calculs de types sont possibles, les prédicats `if`, `for`, etc. sont faisables. Voici, par exemple, comment dérouler totalement une boucle ...

```
#include <iostream>

template<int N> struct Boucle {
        static void body() {
                std::cout << "Boucle " << N << std::endl;
                Boucle<N-1>::body();
        }
};


template<> struct Boucle<0> {
        static void body() {
                std::cout << "Boucle " << 0 << std::endl;
        }
};

int main (int argc, char **) {

        Boucle<10>::body();

        return 0;
}
```

... ou encore un test sur les types :

```
#include <iostream>

template<bool B> struct If;

template<> struct If<true> {
        static void body() { std::cout << "Vrai"; }
};
template<> struct If<false> {
        static void body() { std::cout << "Faux"; }
};


template<typename X, typename Y> struct Egaux {
        static const bool valeur = false;
};

template<> template<typename X> struct Egaux<X,X> {
        static const bool valeur = true;
};

int main(int argc, char ** argv) {

        If< Egaux<int, long>::valeur >::body();

        return 0;

}
```

Pour plus de détails, les articles de Todd Veldhuizen sur le sujet font référence :
- Todd L. Veldhuizen. Expression templates, C++ Report, 7(5), pp. 26-31, June 1995.
- Todd L. Veldhuizen. Using C++ template metaprograms, C++ Report, 7(4), pp. 36-43, May 1995.

F) Initialisation template par liste

En C++11 (le standard paru en 2011), les listes d'initialisations (par accolades sont généralisées. Depuis le C on peut initialiser dans la pile à l'aide des accolades :

```
int tab[3] = { 7, 11, -5};

struct Mixte {
    float a; int b;
};

Mixte couple = {0.43, 10};
```

C++11 généralise le concept grâce au template `std::initializer_list` qui permet de passer des listes d'éléments entre accolades à des fonctions, des constructeurs, etc.

```
void f(std::initializer_list<float> liste);

f( {1.0, -3.45, -0.4} );
```

G) Initialisation uniforme

En C++11 l'initialisation de liste est généralisée à toutes les constructions pour réaliser une initialisation uniforme des objets avec des accolades :

```
struct Mixte { float a; int b; };
struct MixteCstor {
   float a; int b;
   MixteCstor(float x, int y) : a{x}, b{y} {}
   // a et b sont initialisés chacun par une liste à 1 élément
};

Mixte      M { -3.42, 17 }; // initialisation de liste
MixteCstor N { -7.86, 58 }; // initialisation par le constructeur
```

13 Standard Template Library

La STL (Standard Template Library) fournit différents types génériques implémentant des containers d'objets. Un *container* peut être vu comme une séquence d'objets de type `T` qui peut être parcourue en utilisant des *itérateurs*.

A) Séquences

La STL propose trois séquences de base :
- `std::vector< T >` : vecteur à une dimension ; permet l'accès rapide à n'importe quel élément en temps constant. L'insertion et la destruction d'un élément sont coûteuses.
- `std::list< T >` : liste doublement chaînée. L'insertion et la destruction sont rapides ; l'accès à un élément aléatoire coûteux.
- `std::deque< T >` : séquence pour laquelle l'ajout en tête ou en queue est aussi efficace que pour `std::list`, et où l'accès à n'importe quel élément est presque aussi efficace que pour un vecteur. L'insertion et la destruction en milieu sont coûteux.

Les méthodes communes à ces trois séquences sont les suivantes :
- size() : renvoie le nombre d'éléments
- resize(int i) : change le nombre d'éléments
- begin() délivre un "pointeur" sur le premier élément `C[0]`
- end() délivre un "pointeur" sur le premier élément qui n'appartient plus au container (juste après `C[N-1]`)
- front() le premier élément `*(C.begin())`
- back() le dernier élément `*(--C.end())`
- push_back(T x) ≡ `a.insert(a.end(), x)`
- pop_back() ≡ `a.erase(--a.end())`

```
#include <vector>
std::vector<int> notes;

// Remplissage sans savoir le nombre de valeurs dans le fichier
while ( !file.isAtEnd() )
{
  int i
  file >> i;
  notes.push_back(i);
}
```

Par ailleurs, il existe trois containers de base optimisés, qui peuvent être vus comme des vecteurs :
- `std::basic_string` et `std::string` : chaîne de caractères ;
- `std::bitset` : vecteur de bits (opérateurs logiques et de décalages) ;
- `std::valarray` : vecteur optimisé pour les calculs numériques (vectoriels).

En C++11, les vecteurs de la bibliothèque standard acceptent l'initialisation de liste et comme tout type l'initialisation uniforme :

```
std::vector<int> u = { 1, -2, -3, 4 }; // u est un vecteur de taille 4
std::vector<int> v{ 7 };               // v est un vecteur de taille 1
std::vector<int> w{ 5, 7 };            // w est un vecteur de taille 2
std::vector<int> x( 7 );               // x est un vecteur de taille 7
std::vector<int> y( 5, 7 );            // y est un vecteur de taille 5
```

B) Itérateurs

De manière générale, un container `C` est une séquence de N objets de type :
`T : { C[0], ..., C[N-1] }`.

Tout container contient des itérateurs qui permettent de le parcourir, en renvoyant l'équivalent de pointeurs sur ses éléments.
– `C.begin()` délivre un *itérateur* sur le premier élément `C[0]` ;
– `C.end()` délivre un *itérateur* sur le premier élément qui n'appartient plus au container (juste après `C[N-1]`) ;

Ces éléments sont de type `const_iterator` (pointeur sur un élément non modifiable) ou `iterator`. Voici un exemple de parcours :

```
for (std::vector<int>::const_iterator iter = vec.begin();
     iter != vec.end(); ++iter)
  printf("%d ", *iter);
```

Trois principaux itérateurs sont proposés :
– `Forward iterator  : C.begin(), C.end().`
– `Backward iterator : C.rbegin(), C.rend().`
– `Random iterator   : C[i]` (sans vérification de débordement) ou `C.at(i)` (avec vérification).
Il est donc également possible de parcourir le conteneur à l'envers avec `C.rbegin()` et `C.rend()`. L'accès à un élément arbitraire est possible via `C[i]` (sans vérification de débordement) ou `C.at(i)` (avec vérification).

Voici un exemple d'écriture d'un opérateur d'affichage générique de conteneurs.

```
#include <iostream>
template<class T, class A, template <class X, class Y> class Container>
std::ostream& operator<< (std::ostream& o, const Container<T,A>& C) {
  typename Container<T,A>::const_iterator refs =  C.begin();
  for(  ;  refs != C.end() ;  ++refs ) {
      o << *refs << ' ' ;
  }
  return o << std::endl;
}
```

Une fois cet opérateur générique défini, il est possible d'afficher tout conteneur de cette façon.

```
#include <vector>
#include <list>
int main() {

  std::vector<int> v(2); v[0] = 1; v[1] = 2;
  std::list<char> l; l.push_back('f'); l.push_front('g');

  std::cout << "vecteur : " << v;
  std::cout << "list : " << l;

  return 0;
}
```

C) Algorithmes génériques

La STL fournit en standard différents algorithmes génériques qui peuvent être appliqués à de nombreux containers :
– `find, search, sort, for_each, copy, rotate, random_shuffle,` ...
– `min, max, lexicographical_compare,` ...
– opérations sur les ensembles : `includes, union, intersection,` ...
– construction d'une structure de tas : `make_heap, push_heap, pop_heap, sort_heap`
– Permutations : `next_permutation, prev_permutation`.

D) Piles et Files

Trois adapteurs sont fournis sur les containers de base `std::vector` et `std::deque` :
– `std::stack< T >` : pile LIFO
– `std::queue< T >` : file FIFO
– `std::priority_queue< T, cmp >` : file avec priorité.

Les méthodes communes sont (la notion de "en tête" est relative au container : FIFO, LIFO ou plus prioritaire) :
– `push(const T& a)` : ajoute un élément
– `T& top()` : retourne l'élément en tête
– `pop()` : supprime l'élément en tête

Dans le cas d'une file de priorité, la priorité est définie par la fonction classe `cmp`, permettant de comparer des éléments, et qui doit avoir la spécification :

```
struct cmp{
  bool operator()( const T& x, const T& y) { ... }
};
```

Par défaut, la fonction classe `less` est utilisée. L'ordre des priorités est défini par rapport aux entiers. Sur les `int`, `less` implémente la comparaison au sens de l'opérateur "`<`" : l'élément le plus prioritaire est alors le plus grand.

E) Containers associatifs

Un container associatif (ou carte d'association) est un container où chaque élément est associé à une clef (on parle aussi de dictionnaire). Étant donnée une clef, il est possible d'accéder à l'élément associé.

Un container associatif est générique par rapport à 3 paramètres `template` et se déclare via `std::map< Key, T, cmp >` :
– le type `T` de ses éléments
– le type `Key` des clefs
– la fonction classe `cmp` qui définit la comparaison entre deux clefs. Par défaut, la fonction classe `less` (correspond à l'opérateur `<`) est utilisée.

```
template<typename _Tp>
struct less : public binary_function<_Tp, _Tp, bool> {
   bool operator()(const _Tp& __x, const _Tp& __y) const {
      return __x < __y;
   }
};
```

A une clef correspond (au plus) un unique élément : `C[k]` retourne une référence sur l'objet (de type `T`) dont la clef est `k` (de type `Key`).

`std::multimap< Key, T, Cmp >` permet d'associer plusieurs éléments à une clef.

F) Ensembles

Les ensembles d'objets sont gérés par un `std::set< Key, Cmp >`. Un ensemble implémente donc un arbre équilibré de clefs. Il ne fournit pas l'opérateur `[ ]`, mais le test d'appartenance d'un objet à l'ensemble est optimisé.

`std::multiset< Key, Cmp >` : comme `set`, mais une même clef peut apparaître plusieurs fois.

G) Lambda

En C++11, il est possible d'utiliser des fonctions classes de manière anonyme, on les appelle alors `lambda`. Elles sont utiles par exemple pour simplifier l'utilisation des algorithmes génériques.

```
std::vector<int> v { 1, 3, -2, 11 };
std::sort(v.begin(), v.end(), less<int>() );

std::vector<int> w { 1, 3, -2, 11 };
// Ne nécessite pas une définition préalable de la fonction classe
std::sort(w.begin(), w.end(), [](int x, int y)->bool { return x<y; }  );
```

H) auto et decltype

En C++ 11 (compilation avec le drapeau `--std=c++11`), le mot-clef `auto` a une nouvelle sémantique : `auto` prend la place du type dans la déclaration. Le type sera alors automatiquement décidé par correspondance avec le type retourné par l'objet utilisé pour l'initialisation de la variable.

⚠ Les variables étant déclarées avec `auto` devront donc impérativement être initialisées.

```
float fun();

auto a = 5;     // a est de type int
auto b = fun(); // b est du type de retour de la fonction fun
```

decltype permet d'obtenir le type d'un objet pour en déclarer un autre.

```
template<typename Tableau> void f(const Tableau& t) {
  decltype(t[0]) a; // a du type des éléments du tableau
}
```

Ces inférences automatique de type sont particulièrement utiles pour les itérateurs, car cela simplifie largement leur déclaration, comme on peut le voir sur l'exemple de l'opérateur de sortie générique :

```
template<class T, class A, template <class X, class Y> class Container>
std::ostream& operator<< (std::ostream& o, const Container<T,A>& C) {
  for(auto refs=C.begin();  refs != C.end() ;  ++refs ) {
      o << *refs << ' ' ;
  }
  return o << std::endl;
}
```

Il existe également une syntaxe pour le parcours de container supprimant totalement les accès visibles à l'itérateur et renvoyant directement la valeur déréférencée :

```
template<class T, class A, template <class X, class Y> class Container>
std::ostream& operator<< (std::ostream& o, const Container<T,A>& C) {
    for(auto vals: C) o << vals << " " ;
    return o << std::endl;
}
```

14 Optimisations de compilation, Profilage, gestion de la mémoire

A) Options d'optimisation pour g++

Générales : `-O, -O3, -O5`

Non automatiques :

```
-funroll-loops              : déroule les boucles dont le nombre
d'itérations est connu à la compilation (moins de tests) ...

-funroll-all-loops          : toutes les boucles sont déroulées ...
                              (souvent plus lent que le précédent)

-frerun-loop-opt  -floop-optimize          : ... et redéroulées

-fexpensive-optimizations   : quelques optimisations mineures, qui
                              ralentissent la compilation

-felide-constructors       : Pas de temporaire créé pour initialiser
un objet (T x = y; est remplacé par T x(y);)

et aussi :
-fstrict-aliasing -fomit-frame-pointer -fprefetch-loop-arrays
-malign-double -falign-functions -falign-labels
-falign-loops -falign-jumps
-fschedule-insns2 -fforce-addr -fforce-mem -fstrength-reduce
-ffast-math
```

Spécifiques à la machine :

```
-mcpu=ultrasparc -mtune=ultrasparc  : sur Sun Ultra
-mcpu=rs6000 -mtune=rs6000          : sur IBM
-march=pentium; -march=pentiumIV    : sur PC
-march=k8 -mtune=k8
-march=core2 -mtune=core2

-march=native                       : architecture automatique
```

La liste des options peut être obtenue par l'appel à `g++ --help=optimize`.

Pour les spécificités du C++ 11, on pourra utiliser les drapeaux de standard :

```
--std=c++0x        : ancien code de C++ 11
--std=c++11        : utilise le standard C++ 11
```

B) Débogueur : `gdb`

1. Compiler votre programme avec l'option `-g`
2. Passer l'exécutable à `gdb` : `gdb ./monprogram`
3. Lancer l'exécution du programme dans l'environnement du débogueur : `(gdb) run`

`gdb ./monprogram core` permet le débogage correspondant à un "core" créé et `gdb --help` décrit les options de ligne de commande.

 Il est possible de passer des arguments au programme débogué, après la commande `run` (par exemple si `./monprogram` peut être lancé avec des arguments comme `3`, `22` ou encore `fichier.txt`) : `(gdb) run 3 fichier.txt 22`.
Si ces arguments sont toujours les mêmes pour toutes les exécutions successives de `run` à l'intérieur de la session déboguée, il est possible de lancer le débogueur avec l'option `--args` de la façon suivante : `gdb --args ./monprogram 3 fichier.txt 22` puis d'utiliser ensuite un simple `run`.

`where` permet ensuite par exemple de donner la pile d'appels au moment de l'arrêt.

Les points principaux de débogage sont la mise en place de points d'arrêt, le contrôle d'éxecution et l'affichage des variables :

break [file :]line	set breakpoint at line number [in file]
break [file :]func	set breakpoint at func [in file]
break *addr	set breakpoint at address addr
break	set breakpoint at next instruction
tbreak ...	temporary break ; disable when reached
info break	show defined breakpoints
clear	delete breakpoints at next instruction
clear [file :]fun	delete breakpoints at entry to fun()
clear [file :]line	delete breakpoints on source line
delete [n]	delete breakpoints [or breakpoint n]
disable [n]	disable breakpoints [or breakpoint n]
enable [n]	enable breakpoints [or breakpoint n]
enable once [n]	enable breakpoints [or breakpoint n] ; disable again when reached
enable del [n]	enable breakpoints [or breakpoint n] ; delete when reached
ignore n count	ignore breakpoint n, count times

TABLE 1 – Points d'arrêts

continue [count]	continue running ; if count specified, ignore this breakpoint next count times
step [count]	execute until another line reached ; repeat count times if specified
stepi [count]	step by machine instructions rather than source lines
next [count]	execute next line, including any function calls
nexti [count]	next machine instruction rather than source line
until [location]	run until next instruction (or location)
finish	run until selected stack frame returns
jump	line
jump	*address
set var=expr	evaluate expr without displaying it ; use for altering program variables

TABLE 2 – Contrôle de l'exécution

Display print [/f] [expr]	show value of expr [or last value $] according to format f : (x, d, u, o, t, a, c, f)
disassem [addr]	display memory as machine instructions
display [/f] expr	show value of expr each time program stops [according to format f]
display	display all enabled expressions on list
undisplay n	remove number(s) n from list of automatically displayed expressions
disable disp n	disable display for expression(s) number n
enable disp n	enable display for expression(s) number n
info display	numbered list of display expressions
expr	an expression in C, C++
file : :nm	a variable or function nm defined in file
{type}addr	read memory at addr as specified type
$	most recent displayed value
$n	nth displayed value
$$	displayed value previous to $
$$n	nth displayed value back from $
$_	last address examined with x
$__	value at address $_
$var	convenience variable ; assign any value
show values [n]	show last 10 values [or surrounding $n]
show conv	display all convenience variables

TABLE 3 – Affichage

`kill` ou `quit` permettent de sortir de l'exécution.

C) Couverture de tests : `gcov`

1. Compiler votre programme avec l'option `-fprofile-arcs -ftest-coverage` , sans oublier `-fprofile-arcs` à l'édition de liens.
2. Faire une exécution de votre programme (`./monprogram`)
3. Exécuter : `gcov -a ./monprogram > Stats.gcov`

Les fichiers `*.hpp.gcov`, `*.cpp.gcov`, `*.inl.gcov` contiennent les informations de couverture de votre éxecution (nombre de passages dans les différents blocs, etc.). Le fichier `Stats.gcov` contient le pourcentage de lignes éxécutées dans chaque fichier de source.

```
File 'Agora_case.h'
Lines executed:94.37% of 71
Branches executed:100.00% of 22
Taken at least once:90.91% of 22
Calls executed:100.00% of 7
Agora_case.h:creating 'Agora_case.h.gcov'

File 'light_container.h'
Lines executed:100.00% of 14
Branches executed:71.43% of 14
Taken at least once:64.29% of 14
Calls executed:66.67% of 6
light_container.h:creating 'light_container.h.gcov'

File 'agora_coupspossibles.inl'
Lines executed:100.00% of 62
Branches executed:100.00% of 96
Taken at least once:98.96% of 96
Calls executed:100.00% of 47
agora_coupspossibles.inl:creating 'agora_coupspossibles.inl.gcov'

File 'agora_alphabeta.inl'
Lines executed:95.00% of 120
Branches executed:100.00% of 60
Taken at least once:81.67% of 60
Calls executed:100.00% of 65
agora_alphabeta.inl:creating 'agora_alphabeta.inl.gcov'

File 'jgdagora.C'
Lines executed:82.35% of 17
Branches executed:100.00% of 6
Taken at least once:66.67% of 6
Calls executed:83.33% of 18
jgdagora.C:creating 'jgdagora.C.gcov'
...
```

⚠ Attention certaines optimisations peuvent combiner des lignes de code. Pour s'assurer de la couverture, il peut donc être nécessaire de désactiver les options d'optimisation.

D) Profilage : `gprof`

1. Compiler votre programme avec l'option `-pg`
2. Faire une exécution de votre programme (`./monprogram`)
3. Le fichier `gmon.out` contient les statistiques
4. Exécuter : `gprof ./monprogram > Stats.gmon`

Le fichier `Stats.gmon` contient des informations sur les fonctions appelées par votre programme : temps passé dans chaque fonction, nombre d'appels, etc.

Une première partie liste tous les appels de fonctions avec leur temps total. Une deuxième partie donne le détail des sous-fonctions appelées par chaque fonction.

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
 28.05     15.72    15.72 53155300     0.30     0.67  Plateau::UpdatePM(PossibleMove &, short,
 14.56     23.88     8.16 68153700     0.12     0.64  Plateau::UpdatePM(PossibleMove &, short,
 14.26     31.87     7.99 24279986     0.33     0.43  vector<Position, allocator<Position> >::
  8.78     36.79     4.92  1362949     3.61    35.78  Plateau::UpdatePM(PossibleMove &, short,
  8.62     41.62     4.83    19771   244.30  2814.00  Plateau::ennemi_ab(int, double, double,
  4.59     44.19     2.57 48559972     0.05     0.05  __uninitialized_copy_aux__H2ZP8PositionZ
  3.73     46.28     2.09 12722007     0.16     0.25  Coup::Coup(Coup const &)
  2.36     47.60     1.32 15437022     0.09     0.09  Position::BR(Position &) const
  2.32     48.90     1.30 15414874     0.08     0.08  Position::TL(Position &) const
  2.28     50.18     1.28 15836194     0.08     0.08  Position::TR(Position &) const
  2.23     51.43     1.25 15451775     0.08     0.08  Position::BL(Position &) const
  2.03     52.57     1.14  4137040     0.28     0.28  Plateau::unmove(vector<short, allocator<
  1.98     53.68     1.11  4137040     0.27     0.31  Plateau::move(vector<short, allocator<sh
  1.91     54.75     1.07 12722315     0.08     0.08  __uninitialized_copy_aux__H2ZPC8Position
  1.12     55.38     0.63   651778     0.97     1.61  Plateau::addjump(PossibleMove &, short,
  0.59     55.71     0.33 10779582     0.03     0.03  __uninitialized_fill_n_aux__H3ZP8Positio
  0.29     55.87     0.16  4090653     0.04     0.04  vector<short, allocator<short> >::insert
  0.25     56.01     0.14  3532540     0.04     0.04  vector<Coup, allocator<Coup> >::insert(C
  0.05     56.04     0.03                             main
  0.00     56.04     0.00     2931     0.00     0.01  vector<Position, allocator<Position> >::
  0.00     56.04     0.00     1721     0.00     0.00  Possibility::~Possibility(void)
  0.00     56.04     0.00     1528     0.00     0.37  __uninitialized_copy_aux__H2ZP4CoupZP4Co
  0.00     56.04     0.00      992     0.00     0.01  Possibility::Possibility(Possibility con
  0.00     56.04     0.00      992     0.00     0.01  __uninitialized_copy_aux__H2ZPC4CoupZP4C
  0.00     56.04     0.00      764     0.00     0.99  vector<Coup, allocator<Coup> >::_M_inser
  0.00     56.04     0.00      246     0.00     0.00  Timer::operator=(Timer const &)
  0.00     56.04     0.00      246     0.00     0.00  Timer::operator+(Timer const &) const

...
```

Une deuxième partie donne le détail des sous-fonctions appelées par chaque fonction. La notation [ ] référence les fonctions, classées par temps passé décroissant.

```
granularity: each sample hit covers 4 byte(s) for 0.20% of 4.95 seconds
index % time    self  children    called     name
[1]    100.0    0.00    4.95                 main [1]
                0.36    4.54    1768/1768        Plateau::ennemi_ab(int, double, double, short
                0.00    0.04    1083/100553      Plateau::UpdatePM(PossibleMove &, short, shor
                0.00    0.00    2900/314904      Plateau::move(vector<short, allocator<short>
                0.00    0.00    2900/314904      Plateau::unmove(vector<short, allocator<short
                0.00    0.00     550/5028200     Plateau::UpdatePM(PossibleMove &, short, int,
                0.00    0.00     172/1017564     Coup::Coup(Coup const &) [7]
                0.00    0.00      13/13          vector<PossibleMove, allocator<PossibleMove>
                0.00    0.00      10/1927129     vector<Position, allocator<Position> >::_M_in
                0.00    0.00       5/174         vector<Coup, allocator<Coup> >::_M_insert_aux
                0.00    0.00     238/238         vector<Position, allocator<Position> >::opera
                0.00    0.00      10/313222      vector<short, allocator<short> >::insert(shor
                0.00    0.00     104/209         Possibility::~Possibility(void) [46]
                0.00    0.00      35/257902      vector<Coup, allocator<Coup> >::insert(Coup *
                0.00    0.00      22/22          Timer::operator+(Timer const &) const [48]
                0.00    0.00      22/22          Timer::operator=(Timer const &) [47]
                0.00    0.00      18/18          PossibleMove::PossibleMove(void) [51]
                0.00    0.00      17/17          vector<double, allocator<double> >::insert(do
                0.00    0.00      14/14          vector<vector<short, allocator<short> >, allo
                0.00    0.00       1/1           Plateau::~Plateau(void) [54]
-----------------------------------------------
                                84719            Plateau::ennemi_ab(int, double, double, short
                0.36    4.54    1768/1768        main [1]
[2]     99.0    0.36    4.54    1768+84719   Plateau::ennemi_ab(int, double, double, short, sh
                0.37    4.00   99470/100553      Plateau::UpdatePM(PossibleMove &, short, shor
                0.09    0.02  312004/314904      Plateau::move(vector<short, allocator<short>
                0.07    0.00  312004/314904      Plateau::unmove(vector<short, allocator<short
                                84719            Plateau::ennemi_ab(int, double, double, short
-----------------------------------------------
...
-----------------------------------------------
                0.00    0.00      10/1927129     main [1]
                0.00    0.00   12359/1927129     Plateau::addjump(PossibleMove &, short, Coup
                0.74    0.11 1914760/1927129     Plateau::UpdatePM(PossibleMove &, short, int,
[6]     17.2    0.74    0.11 1927129         vector<Position, allocator<Position> >::_M_insert
                0.11    0.00 3854258/3854258     __uninitialized_copy_aux__H2ZP8PositionZP8Pos
                0.00    0.00       3/20          __default_alloc_template<true, 0>::_S_refill(
```

⚠ Ne pas faire `strip` avant l'exécution (car strip permet de réduire la taille de l'exécutable en retirant tous les attributs nécessaires au débogage et au profilage !).
⚠ Penser à désactiver les sorties d'erreur pour ne pas polluer les statistiques.
⚠ Si vos fonctions sont trop `inline`, elles peuvent ne pas apparaître, essayez de compiler avec les options suivantes : `-fno-default-inline -fno-inline -finline-limit-0`
⚠ `_mcount` est une fonction interne à gprof.

E) Gestion de la mémoire : `valgrind`

Le programme suivant compile et s'exécute parfaitement.

```
int main() {
  char * p = new char[13];
  delete p;
  int * a = new int (2);
}
```

Il recèle néanmoins des fuites de mémoire détectées par l'outil `valgrind` :

```
> valgrind --leak-check=full a.out
==23228== Memcheck, a memory error detector.
...
==23228==
--23228-- DWARF2 CFI reader: unhandled CFI instruction 0:50
--23228-- DWARF2 CFI reader: unhandled CFI instruction 0:50
==23228== Mismatched free() / delete / delete []
==23228==    at 0x401CCBC: operator delete(void*) (vg_replace_malloc.c:244)
==23228==    by 0x80484EE: main (memory.cpp:3)
==23228==  Address 0x4282028 is 0 bytes inside a block of size 13 alloc'd
==23228==    at 0x401D7C1: operator new[](unsigned) (vg_replace_malloc.c:195)
==23228==    by 0x80484E0: main (memory.cpp:2)
==23228==
==23228== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 19 from 1)
==23228== malloc/free: in use at exit: 4 bytes in 1 blocks.
==23228== malloc/free: 2 allocs, 1 frees, 17 bytes allocated.
==23228== For counts of detected errors, rerun with: -v
==23228== searching for pointers to 1 not-freed blocks.
==23228== checked 109,140 bytes.
==23228==
==23228== 4 bytes in 1 blocks are definitely lost in loss record 1 of 1
==23228==    at 0x401DB31: operator new(unsigned) (vg_replace_malloc.c:163)
==23228==    by 0x80484FA: main (memory.cpp:4)
==23228==
==23228== LEAK SUMMARY:
==23228==    definitely lost: 4 bytes in 1 blocks.
==23228==      possibly lost: 0 bytes in 0 blocks.
==23228==    still reachable: 0 bytes in 0 blocks.
==23228==         suppressed: 0 bytes in 0 blocks.
==23228== Reachable blocks (those to which a pointer was found) are not shown.
==23228== To see them, rerun with: --show-reachable=yes
```

À noter, `memory.cpp:3` qui détecte la destruction d'un pointeur, qui devrait plutôt celle d'un tableau : `delete [] p;`. Également en `memory.cpp:4` un bloc n'est pas détruit : `delete a;` a été omis dans le programme ci-dessus.

`valgrind` peut aussi être utilisé pour le profilage par l'intermédiaire de `callgrind` :

```
> valgrind --tool=callgrind a.out bb79.bwb 2
==12090== Callgrind, a call-graph generating cache profiler
==12090== Copyright (C) 2002-2010, and GNU GPL'd, by Josef Weidendorfer et al.
==12090== Using Valgrind-3.6.1 and LibVEX; rerun with -h for copyright info
==12090== Command: a.out bb79.bwb 2
==12090==
==12090== For interactive control, run 'callgrind_control -h'.
(0,0)>(1,0)
==12090==
==12090== Events    : Ir
==12090== Collected : 105263691
==12090==
==12090== I   refs:      105,263,691
Profiling time alarm
```

L'intérêt d'utiliser `callgrind` au lieu de `gprof` est que l'on peut observer le profilage pendant son exécution par `callgrind_control -b`.

Au final, de la même manière que pour `gprof`, on peut obtient un compte-rendu des appels dans `gmon.out` et `callgrind.out.<pid>`. Ces appels peuvent être visualisés en mode texte par `callgrind_annotate` ou en mode graphique par `kcachegrind` :

1. Compiler votre programme avec les options : `-g -pg`
2. Lancer l'exécution du programme dans l'environnement `valgrind` :
   `valgrind --tool=callgrind ./monprogram [arguments]`
3. Visualiser l'exécution avec : `callgrind_control -b`
4. Visualiser les résultats collectés avec : `kcachegrind callgrind.out.<pid>`

15 Threads C++11

Il s'agit de la gestion des threads POSIX en C++11, accessible par la bibliothèque standard.

⚠ Pour accéder à l'implémentation : `#include <thread>`, sans oublier `-lpthreads` à l'édition de liens).

A) Lancement de threads

Pour créer un processus distinct il suffit de créer un objet de type `std::thread` et de lui donner en construction la fonction qu'il doit exécuter (soit un pointeur de fonction classique, soit une lambda).

B) Synchronisation

La méthode `join()` termine lorsque l'exécution du thread est terminé.
Cela permet de se synchroniser : l'exécution du thread qui appelle cette méthode sur un objet est bloquée jusqu'à ce que cet objet ait terminé.
Après un appel à cette fonction, l'objet thread devient non-joignable et peut être détruit en toute sécurité.

C) this_thread

Il est possible d'accéder au thread courant via `std::this_thread` et ses méthodes :

`get_id`	Numéro (unique)
`yield`	Laisser la main aux autres
`sleep_until`	Patientier jusqu'à ...
`sleep_for`	Patientier pendant ...

Les durées pour `sleep_until` et `sleep_for` sont exprimées via `std::chrono`.

```
void hello(){ std::cout << "Dans " << std::this_thread::get_id() << '\n'; }

int main(){
    std::vector<std::thread> threads;
    for(int i = 0; i < 5; ++i) {
        threads.push_back(std::thread(hello));
    }

    for(auto& thread : threads){
        thread.join();
    }
    return 0;
}
```

D) Exclusion mutuelle

L'utilisation la plus simple consiste à entourer les opérations sensibles d'un verrou, de type `std::mutex` (ne pas oublier `#include <mutex>` :

```
std::mutex mutex;
mutex.lock()
// ... ici une action courte sera effectuée en séquence
mutex.unlock();
```

⚠ Si des exceptions peuvent être lancées pendant l'action vérouillée, il peut être plus sûr d'utiliser un sémaphore protégé (qui peut-être cependant plus lent qu'un simple mutex) :

```
{
  std::lock_guard<std::mutex> guard(mutex);
  counter.increment();
} // la gestion du dévérouillage est faite par le destructeur de 'guard'
```

Enfin il existe des vérrous récursifs (`std::recursive_mutex`) si un verrou peut être pris plusieurs fois ; et des vérrous à temps limité (`std::timed_mutex` et `std::recursive_timed_mutex` pour limiter le temps de blocage.

À noter également la possibilité de rendre une variable atomique (en accès par exclusion mutuelle unique), simplement en la déclarant de type `std::atomic<T> var;`. Ceci est en général beaucoup plus rapide (car spécialisé) que d'utiliser des vérrous.

E) Appel unique

`std::call_once` est l'équivalent de `static` en mode multi-thread : tout appel de fonction réalisé le sera une unique fois quel que soit le nombre d'appels :

```
std::once_flag flag;
void faireuntruc() {
  std::call_once(flag, [](){ std::cout << "exécuté une seule fois\n"; });
  std::cout << "exécuté par tous les processus\n";
}
```

16 Introduction à OpenMP

OpenMP est un modèle pour la programmation parallèle en mémoire partagée supportant le parallélisme de données.

⚠ Pour compiler un programme OpenMP on ajoutera le drapeau `-fopenmp` à la compilation par `g++`.

A) Accès aux numéros de processus

La fonction `omp_get_num_threads` fournit le nombre de processus dans l'ensemble exécutant la région parallèle dans laquelle elle est appelée.

```
int omp_get_num_threads(void);
```

La fonction `omp_get_thread_num` fournit le numéro de processus, toujours compris entre 0 et `omp_get_num_threads()-1`, à l'intérieur de l'ensemble. Le processus maître a le numéro 0.

```
int omp_get_thread_num(void);
```

Il est également possible de définir le nombre de processus qu'OpenMP peut créer simultanément. Soit dynamiquement par la fonction `omp_set_num_threads`, soit statiquement en fixant ce maximum avant l'exécution, en donnant une valeur numérique à la variable d'environnement `OMP_NUM_THREADS`.

```
void omp_set_num_threads(int)
```

B) Parallélisation de boucles

La parallélisation s'effectue de manière incrémentale en ajoutant des directives `#pragma`. Par exemple, la parallélisation d'une boucle `for` par découpage des itérations en blocs indépendants, nécessite uniquement l'ajout de la directive suivante :

```
#pragma omp parallel for
for(int i=0; i< N; ++i)
  Calculer( i );
```

Il est possible de forcer l'exécution séquentielle d'une partie d'une boucle parallèle :

```
#pragma omp parallel for
for(int i=0; i< N; ++i) {
  Calculer( i );         // Les appels à 'Calculer' sont parallèles
  #pragma omp ordered {
        EnSequence( i ); // Les appels à 'EnSequence' sont séquentiels
  }
}
```

C) Déclaration de sections indépendantes et synchronisation

Il est possible de définir également des blocs de programmes pouvant s'exécuter simultanément. Chaque section de code opèrera dans un seul processus.

```
#pragma omp parallel sections  {

  #pragma omp section {
    // Un bloc de calculs
  }

  #pragma omp section {
    // Un deuxième bloc de calculs, indépendants de ceux du premier bloc
  }

}
```

Pour synchroniser l'arrêt des sections parallèles, il faut attendre la terminaison de tous les processus. Pour cela, on utilise `barrier` qui attend la terminaison de *tous* les processus en cours avant de continuer.

```
#pragma omp barrier
```

D) Zone d'exécution simultanée

Il est également possible de faire s'exécuter la même séquence de code sur *tous* les processus :

```
#pragma omp parallel {
 // ...
}
```

Une telle zone peut également contenir des directives `omp for` ou `omp ordered` pour les boucles, `omp section` ou `omp single` pour des zones à n'effectuer que sur un seul processus (mais pas nécessairement le maître dans ce cas).