Notes de cours INFO702, M1 Informatique
Programmation générique et C++

Jacques-Olivier Lachaud
LAMA, Université de Savoie
jacquesolivierlachaud.github.io
(suivre Teaching/INFO702)

29 septembre 2023

Introduction

Ce document donne les éléments principaux de la programmation générique avec le langage C++. Ce cours s'adresse à des étudiants ayant acquis une licence en Informatique, ayant déjà une connaissance de la programmation impérative et de la programmation objet. Il présuppose une certaine connaissance du langage C (vous pouvez vous référer aux notes de cours de INFO505 - Programmation C, sur le même site), notamment sa syntaxe, ses mécanismes d'allocation mémoire (pile, tas), son organisation (fichiers sources, fichiers en-tête), son cycle de développement. Il présuppose aussi une certaine connaissance d'un langage objet, notamment sur le principe des classes, des méthodes, et du polymorphisme. Ici, on comparera souvent l'approche C++ avec l'approche JAVA, car c'est un langage objet très répandu et utilisé dans de nombreux domaines.

Il existe un nombre considérable de ressources sur Internet pour la programmation C++ ainsi que sur les principes de la programmation générique. On conseille notamment les livres écrits par B. Stroustrup (inventeur du C++, sa FAQ est aussi très intéressante sur son site), S. Meyers (effective C++), H. Sutter, ... Pour une introduction plus progressive au C++, les livres de Deitel et Deitel sont très accessibles. Un autre nom important en C++ est A. Stepanov, un des initiateurs de la bibliothèque standard C++ et grand défenseur et développeur de la programmation générique. On peut dire que c'est vraiment lui et ses co-auteurs qui ont réussi à imposer la programmation générique comme manière de modéliser un problème et de le programmer.

Il est conseillé aussi de bookmarquer les sites `www.cplusplus.com` et `www.cppreference.com`, qui contiennent notamment toute la référence à l'API de la bibliothèque standard C++ (dite **STL**).

Ce document ne donnera pas tous les éléments de la syntaxe du C++, loin de là. Il existe des ressources disponibles qui le font (cf. plus haut, voir aussi le tutoriel sur le site `www.cplusplus.com`).

Pour ceux qui veulent vraiment faire du C++, le site C++ Core Guidelines `http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines` donnent vraiment beaucoup d'éléments intéressants. Il est édité par l'inventeur du C++ (Bjarne Stroustrup) et un de ses experts les plus reconnus (Herb Sutter).

Nous ferons le choix de présenter principalement le standard ANSI-C++ (1998, révisé en 2003, dénommé C++03). Nous parlerons un peu des évolutions postérieures : le standard dit C++0x (planifié dès 2005 mais terminé en 2011) et le standard C++11 (aussi en 2011, qui finalise le précédent). Ces deux derniers standards apportent des tas de choses intéressantes (notamment le mot-clé **auto**, une bibliothèque pour le multi-threading enfin standardisé, les lambda expressions, les rvalues références). Enfin, nous n'aborderons pas du tout les deux normes C++14 et C++17, qui sont essentiellement des correctifs légers pour de la programmation très pointue. On note néanmoins qu'à partir de C++17, le parallélisme et les algorithmes parallèles sont intégrés au langage, et à partir de C++20, les concepts ont enfin intégrés au langage.

Une autre ressource importante de bibliothèques en C++ est BOOST (`www.boost.org`). D'une part une partie de ces bibliothèques finissent par faire partie du standard C++ au fil des ans, d'autre part certaines de ces bibliothèques sont justes très pratiques. Je ne citerai que `boost::program_options` pour vous faciliter la définition, la vérification et la lecture des arguments donnés à un programme, et `boost::compute` pour faire des calculs sur GPU en codant quasiment comme du CPU.

Plan

1. Quelques éléments du langage : syntaxe, variables, pointeurs, références, fonctions, classes
2. Programmation orientée objet, redéfinition vs surcharge
3. Mécanismes d'allocation mémoire
4. Fonctions et classes patrons : définition, utilisation de la STL
5. Généricité dans la STL : itérateurs, conteneurs
6. Ecrire générique : concepts, instantiation, spécialisation
7. Bonus : méta-programmation, lambda, etc.

1 Quelques éléments du langage : syntaxe, variables, pointeurs, références, fonctions, classes

Le langage C++ est un langage compilé typé fortement. Le premier terme indique que vous devez compiler votre programme pour une architecture spécifique avant de pouvoir l'exécuter, le deuxième terme indique que, pour être compilé, toutes les variables sont typées et le compilateur vérifie que vous utilisez les variables du bon type à tout moment. En pratique pour vous, cela veut dire que vous ne pourrez exécuter votre programme tant que toutes ses lignes sont consistentes. Cela change notamment de langage comme le PHP ou le python, où il faut que la machine exécute une ligne invalide pour qu'une erreur (exception) se produise.

1.1 Le programme "Hello world !"

En C++, il ressemble à ceci (fichier appelé `hello.cpp`) :

```
#include <iostream>
int main( int argc, char** argv )
{
  std::cout << "Hello World !" << std::endl;
  return 0;
}
```

Cela ressemble à du C, mais la gestion de l'affichage sur la sortie standard est plus agréable qu'en C. Ici `std::cout` désigne une variable globale `cout` définie dans l'espace de nom `std`, c'est-à-dire la bibliothèque standard C++ ou STL. Le type de cette variable est un *flux* de sortie (spécialisation de la classe `ostream`). On remarque tout de suite une spécificité du C++ par rapport au C : l'opérateur `<<`, qui désigne habituellement le décalage à gauche des bits d'un entier, désigne ici un moyen pour envoyer la chaîne de caractères `"Hello World !"` sur le flux `cout`.

Autrement dit, en C++, les opérateurs usuels peuvent être redéfinis pour chaque type déclaré : c'est la *surcharge* des opérateurs.

NB : En C, on ne peut même pas surcharger une fonction (un nom de fonction = une fonction), alors encore moins surcharger des opérateurs.

Questions : En JAVA, peut-on surcharger les opérateurs ? Les fonctions ? Même question en Python ? Quelle est la différence avec le Python ?

On retrouve sinon le *préprocesseur* du C, avec les macros `#include`. Ici, la macro inclut le fichier en-tête `iostream`. Celui-ci contient toutes les déclarations propres à la gestion des flux pour les entrées/sorties standards. Pour les flux liés à des fichiers, on inclut `fstream`.

On écrira pour compiler ce fichier sur votre shell :

```
(you@machine) g++ -c hello.cpp
```

Cela vous rend la main sans rien dire si le programme s'est compilé sans problème. Normalement, un fichier `hello.o` a été créé. Il contient du code machine binaire parfaitement illisible pour vous. Néanmoins, vous ne pouvez pas l'exécuter. En effet, votre code a bien été compilé (en environ 2500 octets de code machine), mais il lui manque le lien avec toutes les fonctions écrites dans le système que vous appelez dans votre code. Ici, vous utilisez l'opérateur `«` surchargé de la classe `basic_stream`. Pour faire l'édition des liens avec la bibliothèque système C++. On écrira :

```
(you@machine) g++ hello.o -o hello
```

Cela fabrique un exécutable `hello`, que l'on peut exécuter, ce qui donne :

```
(you@machine) ./hello
Hello the World !
```

Ici, la commande `g++` (sans le `-c`) fait l'édition des liens avec toute la bibliothèque système C++, qui contient entre autres les fonctions d'entrées/sorties sur la console.

NB : Si vous voulez compilé avec un standard précis (très souvent C++11), alors, si votre compilateur est un peu ancien il faut spécifier :

```
(youmachine) g++ -std=c++11 hello.cpp -o hello
```

1.2 Les variables ; portée d'une variable

Comme en C, les variables désignent un couple (type, identifiant). Physiquement, une variable n'est finalement qu'un alias pour une zone mémoire (donc une adresse et un certain nombre d'octets). La portée d'une variable est soit globale (on peut accéder à la variable par son nom en différents points du programme), soit locale (on ne peut accéder à la variable par son nom que dans la fonction où elle est définie). On peut *définir* ou seulement *déclarer* une variable globale. Les variables locales sont *définies* seulement. Sauf cas particuliers argumentés, il n'y a pas de raison de définir ou déclarer des variables globales. Par défaut, débrouillez-vous pour toujours n'utiliser que des variables locales.

1.2.1 Définition d'une variable.

Une variable est *définie* en écrivant un *type* suivi d'un *identifiant*, qui sera le nom de la variable. La définition d'une variable entraîne la réservation d'un espace mémoire réservé pour cet identifiant. La taille mémoire réservée dépend du type voulu. Par exemple, un entier nécessite entre 1 et 8 octets, tandis qu'un long texte peut prendre quelques milliers d'octets.

```
void f() {
  int i; // Définition d'une variable de type ``int'', non initialisée
  int j = 3; // Définition d'une variable de type ``int'', initialisée à 3
}
```

1.2.2 Déclaration d'une variable.

Déclarer une variable (globale) est une indication pour le compilateur qu'une telle variable (globale) existe ailleurs dans le programme. C'est souvent seulement à l'*édition des liens* que cette existence supposée est vérifiée. Il faut utiliser le mot-clé `extern`.

```
// Fichier toto.hpp
extern const double mon_pi; // déclaration.
...
```

```
// Fichier toto.cpp
const double mon_pi = 3.14159; // définition.
```

1.2.3 Variables globales.

Une variable *définie* en dehors de toute fonction à une portée dite *globale*. Elle est accessible directement à partir de toute fonction définie *après* l'endroit où cette variable était définie.

```
void f()
{
  ma_variable_globale = 7; // Erreur, non encore déclarée
}
// Définition d'une variable globale de nom ``ma_variable_globale''
int ma_variable_globale;
...
void g()
{
  f();
  std::cout << "elle vaut: "
            << ma_variable_globale    // valide
            << std::endl;
}
```

Si vous voulez vous servir d'une variable globale, soit avant dans le même module, soit dans un autre module, on utilise le mot-clé **extern** pour *déclarer* la variable. Cela donnerait pour l'exemple ci-dessus :

```
// Déclaration d'une variable globale de nom ``ma_variable_globale''
extern int ma_variable_globale;
...
void f()
{
  ma_variable_globale = 7; // Tout va bien.
}
...
```

```
// Définition d'une variable globale de nom ''ma_variable_globale''
int ma_variable_globale;
...
```

1.2.4 Variables locales.

Une variable *définie* dans une fonction à une portée dite *locale*. Plus précisément, les variables locales peuvent être soit les paramètres formels (entre les parenthèses), soit définies dans le bloc de code (entre les accolades). Elles sont instanciées automatiquement (réifiées) lorsque le fil d'exécution rentre dans la fonction. Elles sont détruites automatiquement lorsque le fil d'exécution quitte la fonction. On peut accéder (lecture/écriture) à la valeur stockée dans une variable dans la fonction elle-même simplement en écrivant l'identifiant (le nom) de la variable.

```
int mult2( int i )
{ // i est une variable locale à la fonction mult2.
  return 2*i;
}
int carre( int i )
{ /* i est une variable locale à la fonction carre. Elle porte le
  même nom que la variable i de la fonction mult2, mais n'a
  strictement rien à voir. */
  return i*i;
}
// Cette fonction retourne (j+1)*(j+1) + 2*j + 1.
int f( int j )
{
  int val = carre( j+1 );
  val += mult2( j ) + 1;
  return val;
}
```

La *portée* d'une variable est limitée à sa fonction de définition. Sa durée de vie coïncide avec la période où le fil d'exécution est dans la fonction (de l'entrée jusqu'à la sortie). On note donc que la variable `j` (locale à la fonction `f`) existe toujours pendant les exécutions des fonctions appelées dans la fonction `f`), c'est-à-dire dans `carre` puis `mult2`. Néanmoins, cette variable n'est pas accessible via son identifiant `j`. On verra plus loin qu'on peut utiliser les pointeurs pour y accéder.

NB : par convention, on ne peut utiliser une variable dans une fonction avant la ligne où elle est définie. En pratique, les compilateurs prévoient l'espace de toutes les variables locales d'une fonction dès l'entrée dans la fonction.

1.3 Pointeurs ; tableaux

Le C++, comme le C, permet d'utiliser le mécanisme des *pointeurs*, notamment pour accéder à des variables ou des zones mémoire en dehors des variables locales propres à la fonction courante. Modulo le mot-clé `const`, utilisé pour contraindre le champ d'utilisation d'un pointeur, la gestion des pointeurs en C++ est identique à celle du langage C.

```
// [Utilisation de pointeurs (méthode C... ou C++)]
// fonction qui échange les valeurs pointées par pi et pj
void swap(int* pi, int* pj)
{ // pi pointe sur la case mémoire stockant i
  // pj pointe sur la case mémoire stockant j
  int tmp = *pi;
  *pi = *pj;
  *pj = tmp;
}

int main()
{
  int x = 1;
  int y = 2;
  swap( &x, &y ); // après, x = 2, y = 1
}
```

On peut contraindre les pointeurs à pointer sur des zones non-modifiables (type `const T*`) ou à ne pouvoir être déplacé (type `T* const`) ou les deux à la fois (type `const T* const`).

Sur l'exemple ci-dessus on aurait pu écrire :

```
void swap(int* const pi, int* const pj) // le reste inchangé
```

Un *tableau* est déclaré ou défini avec des crochets. Entre crochets, on place la taille du tableau. Comme en C, l'identifiant du tableau est en fait une expression dont la valeur est un pointeur (non déplaçable) vers la première case du tableau. Un tableau est un moyen de définir un grand nombre de variables du même type d'un coup, toutes contigües en mémoire. Ces variables sont accessibles via un entier (appelé un indice) entre 0 et la taille du tableau moins 1.

```
char c[ 10 ]; // tableau de 10 caractères, non initialisé, modifiable, non
    déplaçable
char d[] = "Bonjour"; // d est un tableau de 8 caractères, modifiable, non
    déplaçable
// dessous, déprécié à partir de C++11
char* e = "Bonjour";  // e est un tableau de 8 caractères, non modifiable,
    déplaçable
// il faudrait écrire en C++11
const char* e = "Bonjour";  // e est un tableau de 8 caractères, non modifiable,
    déplaçable
int t[] = { 5, 4, 3, 2, 1 }; // tableau de 5 entiers, modifiable, non déplaçable
```

On note que la définition `T t[]` implique le type `T* const` pour `t`. Les tableaux ont donc une adresse de base non modifiable. L'adresse d'un tableau `t` est `t` lui-même, i.e. l'écriture `&t` est équivalente à `t` tout court.

NB : Une différence majeure entre tableau et pointeur est que la définition d'un tableau (avec une taille spécifiée) entraîne la définition d'autant de variables en mémoire.

1.4 Références

Le C++ dispose de plus du mécanisme de *référence*, qui simplifie dans beaucoup de cas l'écriture du code C++ par rapport au C, tout en ayant la même efficacité. Une référence est souvent utilisée lors d'un passage en **entrée/sortie** ou en **sortie** d'un paramètre. Le paramètre formel devient un *alias* de l'argument d'appel dès sa construction. Ensuite, tout se passe comme si le paramètre formel **est** l'argument d'appel. Toute modification de l'une modifie l'autre. Les références se déclarent avec le symbole `&`.

```
// [Utilisation de références (méthode C++)]
// fonction qui échange les valeurs de i et j
void swap(int& i, int& j)
{ // i est la variable x, j est la variable y
  int tmp = i;
  i = j;
  j = tmp;
}

int main()
{
  int x = 1;
  int y = 2;
  swap( x, y ); // après, x = 2, y = 1
}
```

On peut aussi déclarer des *références non-modifiables* via le mot-clé `const`, par exemple pour passer en **entrée** un paramètre de taille importante. Il n'est ainsi pas recopié, mais vous pouvez y accéder comme si c'était une variable locale.

```
struct Personne {
  char nom[ 100 ];
  char prenom[ 100 ];
};
void affiche( const Personne & p ) // rapide: passage de l'adresse de P1 en
    réalité
{
  std::cout << p.prenom << " " << p.nom << std::endl;
}
int main()
{
  Personne P1 = { "Jean", "Bon" }; // Miam...
```

```
  affiche( P1 );
}
```

On ne peut déclarer de références sans les initialiser immédiatement. Ainsi :

```
int a = 7;
int& b = a; // valide, b est un alias de a
int& c; // invalide, il faut désigner un variable du bon type.
b = 3; // à la fois a et b valent 5 (b est la variable a)
```

Puisqu'une référence désigne en fait une autre variable dès sa création, on ne peut donc changer la variable désignée par cette référence par la suite.

Les références peuvent apparaître :

1. dans un bloc de programme, pour se donner une autre nom pour une variable :

```
Personne P[ 100 ]; ...
Personne& p10 = P[ 10 ];  // p10 est la 10eme personne de P
```

2. dans les paramètres de fonction, pour simuler des passages en entrées et/ou sorties :

```
void metAuCarre( double& x ) { x *= x; }
```

3. comme donnée membre d'une classe ou structure ; attention à l'initialiser dans le constructeur

```
struct Donnees { ... };
struct VueDonnees {
  Donnees& myData;
  int myStart, myEnd;
  // Constructeur, requiert une Donnees
  VueDonnees( Donnees& data, int s, int e )
  : myData( data ) // nécessairement initialisé ici
  { myStart = s; myEnd = e; } // peut être fait là ou sur la ligne au-
      dessus
};
```

4. comme valeur de retour d'une méthode (quasiment jamais d'une fonction), pour référencer une donnée membre par exemple.

```
struct Donnees {
  Personne myP[ 100 ];
  // surcharge de l'opérateur [ entier ] pour accéder en écriture au
      tableau
  Personne& operator[]( int i )
  { return myP[ i ]; }
};
```

Comme toute variable, les références sont à la fois des *lvalue* (utilisables à gauche dans une affectation) et des *rvalue* (utilisables à droite dans une affectation).

1.5 Fonctions

La déclaration et la définition des fonctions en C++ suivent un mécanisme très similaire à celui du C. Les fonctions sont *déclarées* sous la forme

TypeValeurDeRetour NomFonction(*TypeParam1 NomParam1*, ..., *TypeParamN NomParamN*);

Les fonctions sont *définies* sous la même forme, en rajoutant le code associé :

TypeValeurDeRetour NomFonction(*TypeParam1 NomParam1*, ..., *TypeParamN NomParamN*)
{
  .../* code */
}

On déclare en général les fonctions dans les fichiers en-tête (pour que d'autres modules puissent s'en servir), on définit en général les fonctions dans les fichiers sources (les autres modules n'ont pas besoin de savoir comment la fonction est écrite). On peut bien sûr aussi écrire des fonctions récursives, comme celle-ci :

```
// mymath.hpp: Ma bibliothèque de math...
// Calcule x^n en temps log(n).
double power( double x, int n );
```

```
// mymath.cpp: Voilà son secret...
// Calcule x^n en temps log(n), pour tout n entier.
double power( double x, int n )
{
  if ( n < 0 ) return 1.0 / power( x, -n );            // x^(-n) = 1/x^n
  else if ( n == 0 ) return 1.0;                       // x^0 = 1
  else if ( n == 1 ) return x;                         // x^1 = x
  else if ( n % 2 == 0 ) return power( x * x, n / 2 ); // x^(2n) = (x^2)^n
  else return power( x * x, n / 2 ) * x;               // x^(2n+1) = ((x^2)^n)*x
}
```

Il est légitime en C++ de *surcharger* des fonctions, c'est-à-dire de définir des fonctions de même nom mais de signature différente (les paramètres sont différents). C'est le compilateur qui choisit la fonction la plus appropriée par rapport aux types que vous donnez en paramètres. On note aussi que le C++ fait des conversions implicites de type dans certains cas pour essayer de trouver la fonction adéquate.

```
double power( double x, int n ); // (1)
float power( float x, int n );   // (2) légitime, on peut faire une version pour
    les float
...
double x = power( 1.4, 3 ); // appelle (1)
float y  = power( 1.4f, 3 ); // appelle (2)
float z  = power( 1, 3 ); // ERREUR: le compilateur ne peut choisir entre (1) et
    (2)
```

On note que le compilateur n'utilise pas le type de la valeur de retour pour choisir quelle fonction il appelle. Le type de la valeur de retour peut varier entre fonctions surchargées.

Comme autre exemple, on peut envisager d'additionner des éléments variés. On peut donc surcharger une fonction `add` pour divers paramètres, ou même surcharger l'opérateur binaire `+`.

```
struct Point { double abs; double ord; };
double add( double a, double b )
{
  return a+b
};
Point add( Point p1, Point p2 )
{
  Point q = { p1.abs + p2.abs, p1.ord + p2.ord };
  return q;
}
Point operator+( Point p1, Point p2 )
{
  return add( p1, p2 );
}
```

NB : attention, les fonctions et classes *patron* (ou *template*) sont définies aussi dans les fichiers en-tête. Cela est dû à une certaine incapacité des compilateurs actuels à gérer la compilation de ces fonctions et classes paramétrées sans connaître leur code tout entier.

1.6 Structures et classes

Comme beaucoup de langages, le C++ offre un moyen de définir de nouveaux types complexes en agrégeant les types déjà créés via des structures (enregistrements, *record*, classes). Il offre deux mot-clés très similaires pour le faire : `struct` et `class`.

La différence réside uniquement dans le fait que, par défaut, une structure est *publique*, et que donc des fonctions extérieures à la structure peuvent accéder aux champs et aux méthodes de la structure, tandis qu'une classe est par défaut *privée* et interdit à toute fonction extérieure d'accéder à ses données : seules ses méthodes ont ce droit-là. On voit donc que les deux mot-clés ont été conservés uniquement pour que les structures aient par défaut les mêmes propriétés qu'en langage C, tandis que les classes poussent le programmeur à préserver l'*encapsulation* des données.

1.6.1 Déclaration des structures et classes

On prend l'exemple standard de la définition d'un point dans le plan.

```cpp
// Point.hpp
#ifndef _POINT_HPP_  // évite d'inclure deux fois un fichier en-tête
#define _POINT_HPP_

class Vecteur; // déclare l'existence d'une classe Vecteur

class Point {
  double coord[ 2 ]; // privé, accessible à partir des méthodes.
public:
  Point(); // déclare un point à l'origine du repère (0,0)
  Point( double aX, double aY ); // déclare un point (aX,aY)
  double x() const; // const = indique que la méthode ne
  double y() const; // change pas les données membres.
  // crée un nouveau point translaté (donc méthode const)
  Point operator+( const Vecteur & v ) const;
  // Translate ce point du vecteur spécifié (donc méthode non-const)
  Point& operator+=( const Vecteur & v );
};
#endif
```

Listing 1 – Fichier Point.hpp

```cpp
struct Point {
  double x;
  double y;
};
...
Point p;
p.x = 17; // valide
```

```cpp
class Point {
  double x; // en fait privé
  double y; // en fait privé
};
...
Point p;
p.x = 17; // INVALIDE
```

Ici, nous avons simplement déclaré qu'un `Point` est l'agrégat de deux `double`. Le C++ contient donc tout le mécanisme du C de construction de types.

1.6.2 Méthodes dans les classes ; surcharge d'opérateurs

Le C++ rajoute la possibilité de définir des fonctions attachées à une classe ou à une instance de cette classe : on parle de *méthode*. On retrouve le principe de la programmation objet en C++. On note aussi l'existence de méthodes particulières :

- les *constructeurs*, qui portent le nom de la classe (mettons `T`). Le constructeur adéquat est appelé lors de l'instanciation d'une variable de ce type, autrement dit lors de la création de l'*objet*.
- le *destructeur*, qui porte le nom `~T`. Le destructeur est appelé juste avant la disparition de la variable de ce type.

On peut aussi redéfinir l'opérateur d'affectation (`operator=`), mais aussi quasiment tous les opérateurs usuels pour les adapter à la classe.

On donne sur les Listings 1, 2, 3, 4 et 5, un exemple assez complet de création de classes `Point` et `Vecteur` du plan. Il y a beaucoup de choses à comprendre dans ces exemples :

- différence entre déclaration et définition d'une classe
- une classe seulement déclarée (ici `Vecteur`) n'est accessible que par référence ou pointeur (sinon le compilateur doit connaître sa taille).
- les méthodes peuvent être `const` ou non ; les premières garantissent que l'objet n'est pas modifié par l'exécution de la méthode.
- lorsqu'une méthode est définie dans le fichier source, il faut rappeler qu'elle appartient à la classe `T` en écrivant `T::` devant le nom de la méthode.
- le pointeur `this`, dans une méthode, est un pointeur vers l'instance actuelle, i.e. c'est l'adresse de l'objet en mémoire.
- on peut redéfinir les opérateurs arithmétiques et logiques usuels pour les adapter à chaque classe. C'est très pratique quand on modélise des objets mathématiques.

On remarque certes que le langage C++ est assez verbeux dans la définition des méthodes de classes. On note néanmoins que l'on pourrait inclure le code directement dans le fichier en-tête comme

```cpp
// Point.cpp
#include "Point.hpp"   // on a besoin des déclarations de
#include "Vecteur.hpp" // ces deux classes

Point::Point()
{
  coord[ 0 ] = 0.0; // initialisation forcée à zéro des membres.
  coord[ 1 ] = 0.0;
}
Point::Point( double aX, double aY )
{
  coord[ 0 ] = aX;
  coord[ 1 ] = aY;
}
double Point::x() const
{
  return coord[ 0 ];
}
double Point::y() const
{
  return coord[ 1 ];
}
Point Point::operator+( const Vecteur & v ) const
{ // Crée directement le point avec les bonnes coordonnées
  return Point( x() + v.x(), y() + v.y() );
}
Point& Point::operator+=( const Vecteur & v )
{ // Crée directement le point avec les bonnes coordonnées
  coord[ 0 ] += v.x();
  coord[ 1 ] += v.y();
  return *this; // retourne une référence à soi-même
}
```

Listing 2 – Fichier Point.cpp

```cpp
// Vecteur.hpp
#ifndef _VECTEUR_HPP_
#define _VECTEUR_HPP_

class Vecteur {
  double comp[ 2 ];
public:
  Vecteur(); // déclare le vecteur nul (0,0)
  Vecteur( double u, double v ); // déclare un vecteur (u,v)
  double x() const; // const = indique que la méthode ne
  double y() const; // change pas les données membres.
  // les vecteurs s'additionnent
  Vecteur operator+( const Vecteur & v ) const; // v3 = v1 + v2
  Vecteur& operator+=( const Vecteur & v );     // v2 += v1
  // les vecteurs se multiplie par un scalaire
  Vecteur operator*( double k ) const; // permet v2 = v1 * k
  Vecteur& operator*=( double k );     // permet v1 *= k
};
Vecteur operator*( double k, const Vecteur & v ); // permet v2 = k * v1

#endif
```

Listing 3 – Fichier Vecteur.hpp

```cpp
// Vecteur.cpp
#include "Vecteur.hpp"

// déclare le vecteur nul (0,0)
Vecteur::Vecteur()
{
  comp[ 0 ] = comp[ 1 ] = 0.0;
}
// déclare un vecteur (u,v)
Vecteur::Vecteur( double u, double v )
{
  comp[ 0 ] = u;
  comp[ 1 ] = v;
}
// accesseur à la composante x
double Vecteur::x() const
{
  return comp[ 0 ];
}
// accesseur à la composante y
double Vecteur::y() const
{
  return comp[ 1 ];
}
// les vecteurs s'additionnent (v3 = v1 + v2)
Vecteur Vecteur::operator+( const Vecteur & v ) const
{
  return Vecteur( x() + v.x(), y() + v.y() );
}
// Permet v2 += v1
Vecteur& Vecteur::operator+=( const Vecteur & v )
{
  comp[ 0 ] += v.x();
  comp[ 1 ] += v.y();
  return *this;
}
// les vecteurs se multiplie par un scalaire (v2 = v1 * k)
Vecteur Vecteur::operator*( double k ) const
{
  return Vecteur( k * x(), k * y() );
}
// permet v1 *= k
Vecteur& Vecteur::operator*=( double k )
{
  comp[ 0 ] *= k;
  comp[ 1 ] *= k;
  return *this;
}
// On note qu'il s'agit d'une faction, pas d'une méthode de la classe
// Vecteur. Permet v2 = k * v1
Vecteur operator*( double k, const Vecteur & v )
{
  return Vecteur( k * v.x(), k * v.y() );
}
```

Listing 4 – Fichier Vecteur.cpp

```cpp
#include <iostream>
#include "Point.hpp"
#include "Vecteur.hpp"

int main()
{
  Point p( 2, 1 );
  Vecteur u( 3, 1 );
  p += u;
  std::cout << "p=(" << p.x() << "," << p.y() << ")"
            << std::endl; // affiche p=(5,2)
  u *= 3; // u=(9,3)
  Point q = p + u;
  std::cout << "q=(" << q.x() << "," << q.y() << ")"
            << std::endl; // affiche q=(14,5)
  Vecteur v = -1 * u; // v=(-9,-3)
  q += 2 * v; // appelle operator*(double,Vecteur) puis Point::operator+=
  std::cout << "q=(" << q.x() << "," << q.y() << ")"
            << std::endl; // affiche q=(-4,-1)
  return 0;
  // les destructeurs de p, u, q, et v sont automatiquement appelés.
}
```

Listing 5 – Fichier point-vecteur.cpp

en JAVA. Il y aurait cependant une augmentation du temps de compilation moyen ainsi que de la taille de l'exécutable produit. En revanche, il est très intéressant de voir comment la définition des deux classes `Point` et `Vecteur` permet des écritures compactes lorsqu'on les utilise. Ceci est dû à la surcharge possible des opérateurs algébriques usuels.

1.6.3 Surcharge des opérateurs flux

On peut aussi surcharger les opérateurs de flux de sortie (`operator«`) et d'entrée (`operator»`), afin de faciliter la sérialisation ou l'affichage d'un objet. On rajouterait par exemple dans le fichier `Point.hpp` les lignes suivantes :

```cpp
ostream& operator<<( ostream & out, const Point & p )
{
  out << "(" << p.x() << "," << p.y() << ")";
  return out;
}
```

On pourrait alors réécrire le Listing 5 sous la forme plus compacte :

```cpp
...
p += u;
std::cout << "p=" << p << std::endl; // affiche p=(5,2)
u *= 3; // u=(9,3)
Point q = p + u;
std::cout << "q=" << q << std::endl; // affiche q=(14,5)
Vecteur v = -1 * u; // v=(-9,-3)
q += 2 * v; // appelle operator*(double,Vecteur) puis Point::operator+=
std::cout << "q=" << q << std::endl; // affiche q=(-4,-1)
...
```

On note surtout que l'on utilisera de la même façon les flux pour afficher, pour sauvegarder dans un fichier, ou pour créer une chaîne de caractères.

1.7 Espace de nommages : namespace

Le C++ offre un autre moyen logique pour classer les différentes définitions de variables, fonctions, classes. Ce mécanisme s'appelle les *espaces de nommages*, via le mot-clé `namespace`. Cela permet notamment de ne pas utiliser par inadvertance une fonction/classe définie quelque part, tout en pensant en fait utiliser une autre.

Ainsi, par défaut, toutes vos définitions sont dans l'espace de nommage global. En revanche, toutes les variables, fonctions et classes de la bibliothèque standard C++ sont rangés dans l'espace de nom

`std`. Pour accéder à celles-ci, il faut donc utiliser l'opérateur de résolution de portée `::`, qui permet de spécifier où le compilateur doit chercher l'identifiant.

Cela explique pourquoi on écrit `std::cout` pour accéder à la variable globale `cout` placé dans l'espace de nom `std`.

Vous pouvez vous-même définir vos fonctions/classes dans votre propre espace de nommage, par exemple :

```
namespace heroes {
  class Superman { ... };
  class Batman { ... };
  class Bicyclerepairman { ... };
}
...
heroes::Superman clark_kent;
heroes::Batman bruce_wayne;
heroes::Bicyclerepairman michael_palin;
```

Dans un fichier source (mais ce **n'est pas** recommandé dans un fichier en-tête), on peut utiliser la directive `using namespace` pour que le compilateur aille rechercher tout seul dans cet espace de nom les identifiants spécifiés. Sur l'exemple précédent, on aurait pu écrire :

```
using namespace heroes;
...
Superman clark_kent;
Batman bruce_wayne;
Bicyclerepairman michael_palin;
```

1.8 Chaînes de caractères

On peut utiliser en C++ le même mécanisme que le C pour les chaînes de caractères, c'est-à-dire les tableaux de caractères terminés par le caractère 0 (ou `'\0'`). Néanmoins, on sait que ce n'est pas toujours pratique, notamment si on veut augmenter ou diminuer la taille de la chaîne. La STL fournit une classe `string` (dans le namespace `std`) extrêmement pratique pour manipuler les chaînes de caractères. Grâce à elle, il n'y a aucun risque de dépassement de capacité, et on peut concaténer, découper, dupliquer les chaînes très facilement.

```
// string.cpp
#include <string>
#include <iostream>

using namespace std; // par flemme de mettre std::

int main()
{
  string s = "Bonjour";
  s += string(" la ") + string("compagnie");
  cout << s << endl; // "Bonjour la compagnie"
  cout << s.length() << endl; // 20
  cout << s.find( "la", 0 ) << endl; // 8
  cout << s[1] << s[2] << endl; // "on"
  const char* str = s.c_str(); // conversion vers une chaîne C
  cout << str << endl; // "Bonjour la compagnie"
  return 0;
}
```

L'accès à un caractère se fait évidemment avec l'opérateur [], comme si c'était un tableau de caractères.

1.9 *lvalues* et *rvalues* en C++

Le standard (C++03 3.10/1) dit : "Toute expression est soit une *lvalue* soit une *rvalue*." C'est important de se rappeler que cette propriété (*lvalue* contre *rvalue*) est une propriété des expressions et non des objets.

Les expressions appelés *lvalue* sont des valeurs qui persistent même après que le flot d'exécution soit passé dessus. Par exemple, un identifieur de variable `obj`, l'indirection d'un pointeur `*ptr`, une case d'un tableau `ptr[index]` ou une variable pré-incrémentée `++x` sont tous des *lvalues*. Les valeurs

correspondantes sont en fait toujours en mémoire quelque part même après que ces expressions aient été exécutées.

Les *rvalues* sont des temporaires qui s'évaporent à la fin d'une expression complète dans laquelle ils vivent (l'expression se termine au prochain "`;`"). Par exemple, une constante entière `1729`, une expression arithmétique `x + y`, un objet non nommé `std::string("meow")` , ou une variable post-incrémentée `x++` sont tous des *rvalues*.

Un moyen de détecter *a posteriori* si on est en présence d'une *lvalue* ou d'une *rvalue* est l'opérateur d'adresse. L'opérateur d'adresse (`operator&`) requiert que son opérande soit une *lvalue*. Donc, si l'on peut prendre l'adresse d'une expression, c'est une *lvalue*, sinon c'est une *rvalue*.

```
A obj;
&obj; // valide
&12;  // invalide
A & a = obj;  // valide
int & i = 12; // invalide
const int & j = 12; // valide (durée de vie jusqu'à la fin de j
```

On peut faire une référence à une *lvalue* mais pas à une *rvalue*. On note que l'on peut faire une *référence constante* à une *rvalue* (et bien sûr à une *lvalue* aussi).

2 Programmation orientée objet, redéfinition vs surcharge

Le C++ offre (presque) tous les mécanismes classiques de la programmation dite objet : méthodes de classes, méthodes d'instance, héritage (simple ou multiple), classes abstraites, polymorphisme (dynamique), encapsulation, transtypage. il n'offre pas naturellement tous les mécanismes d'introspection ou de méta-classes, même si l'utilisateur peut vérifier dynamiquement les types de ses objets à l'exécution.

Contrairement au Python par exemple, les méthodes associées à une classe ne sont pas modifiables à l'exécution. Sur l'exemple du Listing 3, la méthode `Vecteur::x() const` ne peut être changée pendant l'exécution du programme.

2.1 Encapsulation

On dispose de trois domaines de déclaration au sein d'une classe ou structure : *publique* (mot-clé `public`) , *protégé* (mot-clé `protected`), *privé* (mot-clé `private`).

public : Dans ce domaine, toute méthode ou attribut est accessible de partout ailleurs dans le code, et donc en particulier accessible du code défini dans les classes dérivées et du code défini dans la classe elle-même. C'est le domaine par défaut des structures.

protected : Dans ce domaine, toute méthode ou attribut est accessible du code défini dans les classes dérivées et du code défini dans la classe elle-même, mais de nulle part ailleurs sinon (sauf classes "amies").

private : Dans ce domaine, toute méthode ou attribut est accessible du code défini dans la classe elle-même, mais de nulle part ailleurs sinon (sauf classes "amies"). C'est le domaine par défaut des classes.

Dans la classe on change de domaine de déclaration en écrivant le mot-clé correspondant suivi de "`:`".

```
class A {
  A() {
    everywhere = -1; // ok
    restricted = -1; // ok
    forbidden  = -1; // ok
  }
public:
  int everywhere;
protected:
  int restricted;
private;
  int forbidden;
};
```

```
class B : public A { // classe dérivée
  B() {
    everywhere = 4; // ok
    restricted = 2; // ok
    forbidden  = 0; // NON
  }
};
int f() {
  A a;
  a.everywhere = 5; // ok
  a.restricted = 3; // NON
  a.forbidden  = 1; // NON
}
```

2.2 Méthodes/attributs de classe ; méthodes/attributs d'instance

En C++, les fonctions ou variables déclarées dans une classe ou une structure sont associées par défaut à chaque instance/objet : ce sont alors des *méthodes* ou des *attributs d'instance*. Il faut rajouter le mot-clé `static` pour préciser que la fonction ou la variable déclarée est associée à la classe toute entière : ce sont alors des *méthodes* ou des *attributs de classe*. On peut par exemple s'en servir pour compter le nombre d'objet d'un type donné qui existent au même moment.

```
// file A.hpp
struct A {
  A() { ++nb; }  // constructeur
  ~A() { --nb; } // destructeur
  static int nbInstances() { return
      nb; }
private:
  static int nb; // attribut de
      classe
};
```

```
// file A.cpp
#include "A.hpp"
int A::nb = 0; // initialisé avant d'
    entrer dans le main.
```

```
// file any.cpp
#include <iostream>
#include "A.hpp"
int main()
{
  A a1,a2,a3; // creates 3 objects.
  A a[8];     // creates 8 objects.
  std::cout << "Nb A = " << A::nbInstances() << std::endl;
  // Nb A = 11
}
```

2.3 Sous-typage ; héritage simple

Une classe ou une structure peut indiquer qu'elle est une *sous-classe* (ou *spécialisation* d'une classe de base en utilisant la notation " :" lors de sa définition. On peut ensuite préciser si l'héritage est `public` (lorsqu'il s'agit de modéliser la relation "est un"), ou `protected` ou `private` (lorsque la classe de base est juste un détail d'implémentation).

```
struct Animal {
  Animal( const string & nom ) : mNom( nom ) {}
  string nom() const { return mNom; }
protected:
  string mNom;
};
struct Mammifere : public Animal { ... };
struct Insecte : public Animal { ... };
struct Lion : public Mammifere { ... };
struct Homme : public Mammifere { ... };
struct Mouche : public Insecte { ... };
...
```

La classe de base est aussi appelée *classe mère* ou *super-classe*. La *sous-classe* est aussi appelée *classe dérivée* ou *classe fille*.

La classe dérivée a accès à toutes les méthodes et attributs publiques ou protégés de la super-classe. Inversement, une super-classe ne connaît pas ses sous-classes.

Le tableau ci-dessous précise les accès possibles aux méthodes et attributs de la super-classe en fonction du type d'héritage (publique, protégé ou privé).

Domaine dans la classe de base	private	protected	public
Type d'héritage	Méthode/attribut hérité comme		
private	inaccessible	private	private
protected	inaccessible	protected	protected
public	inaccessible	protected	public

Dans les sous-classes, on peut appeler le constructeur souhaité de la super-classe en donnant le nom de la classe.

```
struct Mammifere : public Animal {
  Mammifere( const string & nom )
```

```
  : Animal( nom ) // appelle le constructeur Animal( const string & )
  { ... }
};
```

Attention, toutes les instances ont une taille bien définie en C++. Un `Animal` n'a pas la même taille (mémoire) qu'un `Mammifere`. Un tableau d'`Animal` n'a *a priori* pas la même taille qu'un tableau de `Mammifere`, et ils ne sont pas compatibles. Pour manipuler des collections d'animaux différents, il faut passer par des pointeurs. Dans le cas général, la programmation générique via polymorphisme requiert en C++ l'utilisation de pointeurs et/ou de références.

```
void affiche( const Animal & a )
{
  std::cout << a.nom() << std::endl;
}
...
{
  Mammifere panda( "Panda" );
  Insecte   mouche( "Mouche" );
  affiche( panda );  // ok
  affiche( mouche ); // ok
}
```

Un tableau à n éléments crée d'un coup n variables en appelant leur constructeur par défaut. Il est donc impossible de faire un tableau de référence, puisque toutes les références doivent être initialisée dès leur création. Pour manipuler une collection d'animaux variés (i.e. polymorphes) il faut donc faire un tableau de pointeurs d'animaux.

```
Animal* animaux[ 5 ];
animaux[ 0 ] = new Lion( "Mufasa" );
animaux[ 1 ] = new Morpion( "Corback" );
animaux[ 2 ] = new Mouche( "Patouche" );
animaux[ 3 ] = new Chat( "Toulouse" );
animaux[ 4 ] = new Quiche( "Lorraine" ); // NON, une quiche ne dérive pas de
    Animal.
for ( int i = 0; i < 4; ++i )
  std::cout << "a[" << i << "]=" << animaux[i]->nom() << std::endl;
```

Le choix de faire ou non de l'allocation dynamique dépend de la durée de vie souhaitée des objets créés. La question est :

"Est-ce que l'objet a une durée de vie supérieure à la fonction/méthode où elle est créée ?"

— Si oui, alors si vous êtes dans une fonction, cet objet doit être alloué dynamiquement (cf. exemple ci-dessus) ; si vous êtes dans une méthode d'instance, cet objet peut être alloué dynamiquement ou être une donnée membre de cette instance.

— Si non, alors il suffit de créer cet objet sur la pile, ce qui donnerait avec l'exemple ci-dessus :

```
Lion mufasa( "Mufasa" );
Morpion corback( "Corback" );
Mouche patouche( "Patouche" );
Chat toulouse( "Toulouse" );
Quiche lorraine( "Lorraine" );
Animal* animaux[ 5 ] = { &mufasa, &corback, &patouche, &toulouse,
  &lorraine // NON, une quiche ne dérive pas de Animal.
  };
for ( int i = 0; i < 4; ++i )
  std::cout << "a[" << i << "]=" << animaux[i]->nom() << std::endl;
```

2.4 Polymorphisme dynamique ; redéfinition

En POO, l'héritage est souvent un moyen pour manipuler de façon homogène des objets de types concrets différents. On souhaite souvent que ces objets puissent répondre de manière différente même s'ils partagent une interface commune. On parle de *polymorphisme*. Lorsque le type réel de l'objet est déterminé à l'exécution, on parle de *polymorphisme dynamique* ou *ad hoc*.

Les interfaces graphiques constituent un des meilleurs exemples du polymorphisme : c'est d'ailleurs leur développement qui a promu la POO. Ainsi, tout élément d'une interface est une spécialisation d'un composant, qui est un type abstrait qui spécifie un ensemble d'opérations que doit savoir faire tout composant. Certaines classes dérivées d'un composant sont concrètes : un menu, une zone de

dessin, un bouton, une fenêtre, etc. D'autres sont abstraites comme les conteneurs. Tout composant sait par exemple s'afficher, gérer un événement souris ou clavier, donner sa taille et sa position, etc.

En C++, les classes abstraites sont des classes où certaines méthodes sont forcées à être redéfinies dans une sous-classe. Ces méthodes sont dites *virtuelles*. On utilise le mot-clé `virtual` devant la méthode pour indiquer que celle-ci peut être redéfinie dans une sous-classe et le mot-clé `= 0` derrière la méthode pour rendre la classe abstraite et forcer la redéfinition.

```
class Component { // classe abstraite.
  ...
  virtual void draw( GraphicContext& gr ) = 0; // doit être redéfinie
  virtual bool onEvent( Event& event ) = 0;    // doit être redéfinie
};
class Container { // classe abstraite.
  ...
  virtual void draw( GraphicContext& gr ) = 0; // doit être redéfinie
  virtual bool onEvent( Event& event )         // peut être redéfinie
  { // Par défaut, envoie l'événement aux composants internes.
    bool ok = true;
    for ( int i = 0; i < mComposants.size(); ++i )
      ok = mComposants[ i ]->onEvent( event ) && ok;
    return ok;
  }
  void ajoutComposant( Component* c )
  { mComposants.push_back( c ); }

  std::vector<Component*> mComposants;
};
class Label : public Component { // classe concrète
  ...
  virtual void draw( GraphicContext& gr ) // redéfinie
  {
    gr.drawText( mLabel, ... );
    ...
  }
  virtual bool onEvent( Event& event )    // redéfinie
  { // Quel que soit l'événement, un label ne fait rien.
    return true;
  }
  std::string mLabel;
};
class Frame : public Container {
... };
class Button : public Component {
... };

{
  Frame frame;
  frame.ajoutComposant( new Label( "Toto" ) );
  frame.ajoutComposant( new Label( "Titi" ) );
  frame.ajoutComposant( new Button( "Quit" ) );

  frame.main_loop(); // appelera draw, onEvent, etc
}
```

Redéfinir une méthode est le fait de réécrire le corps d'une méthode dans une classe fille. La classe mère doit préciser que la méthode est redéfinissable via le mot-clé `virtual`. La classe fille doit utiliser exactement la signature de la méthode (nom, type et nom des paramètres, const ou pas). Le type de la valeur de retour peut-être changée au besoin. On pense par exemple à un opérateur de clonage, où on peut spécialiser le type de retour. On note que les constructeurs ne peuvent être des méthodes virtuelles et que l'on ne peut pas redéfinir les constructeurs.

```
class A {
  virtual A* clone() const = 0;
};
class B : public A {
  B( const B & other ) { ... }
  virtual B* clone() const  // légal
  { return new B( *this ); }
};
class C : public A {
  C( const C & other ) { ... }
  virtual C* clone() const  // légal
  { return new C( *this ); }
};
```

```
B b1( ... );
C c1( ... );

B* pB = b1.clone();  // légal
C* pC = c1.clone();  // légal
A* pA = b1.clone();  // légal aussi.
```

Le programme détermine à l'*exécution* le type réel de l'objet pointé/référencé et en déduit quelle est la bonne méthode à appeler. En C++, cela se fait au moyen d'une table virtuelle associée à chaque classe contenant des méthodes virtuelles. Cela introduit donc un léger surcoût mémoire (4 ou 8 octets par instance) et un surcoût temps à l'exécution (pour déterminer la bonne méthode à appeler). Néanmoins, on retrouve toute la généricité offerte par le polymorphisme dynamique et notamment la gestion facile de collections hétéroclites d'objets. Le langage JAVA a par exemple fait le choix d'un polymorphisme dynamique quasi-systématique (seules les méthodes `final` ne le sont pas).

On note qu'une classe est dite *polymorphe* lorsqu'elle possède au moins une méthode virtuelle. Voici ci-dessous un exemple plus complet d'utilisation du polymorphisme dynamique.

```
// Le repère est celui utilisé en mathématiques.
struct Boite {
  Boite( double xg, double yb, double xd, double yh )
  : bg( xg, yb ), hd( xd, yh )
  {}
  bool estDans( Point p ) const
  {
    return p.x() >= bg.x() && p.y() >= bg.y()
        && p.x() <= hd.x() && p.y() <= hd.y();
  }
  Point bg; // point bas gauche
  Point hd; // point haut droit
};

class Forme { // classe abstraite
  // Toute forme peut retourner la boite qui l'englobe au plus près.
  // C'est par exemple utilisé pour accélérer les tests de collision.
  virtual Boite boiteEnglobante() const = 0;
  // virtual: indique que la méthode peut être redéfinie dans une classe dérivée.
  // = 0    : indique que la méthode doit être redéfinie dans une classe dérivée.
}

class Disque : public Forme {
  Disque( Point c, double r )
  : centre( c ), rayon( r )
  {}
  virtual Boite boiteEnglobante() const
  {
    return Boite( c.x() - r, c.y() - r,
                  c.x() + r, c.y() + r );
  }
  ...
  Point centre;
  double rayon;
};

class Triangle : public Forme {
  Triangle( Point a, Point b, Point c )
  : p1( a ), p2( a ), p3( a )
  {}
  virtual Boite boiteEnglobante() const
  {
    return Boite( std::min( p1.x(), std::min( p2.x(), p3.x() ) ),
                  std::min( p1.y(), std::min( p2.y(), p3.y() ) ),
                  std::max( p1.x(), std::max( p2.x(), p3.x() ) ),
```

```
                    std::max( p1.y(), std::max( p2.y(), p3.y() ) ) );
  }
  ...
  Point p1, p2, p3;
};
...
{
  Forme* tbl[ 4 ]; // on ne peut pas faire un tableau de Forme.
  tbl[ 0 ] = new Disque( Point( -2, 1 ), 3 );
  tbl[ 1 ] = new Triangle( Point( -4, -3 ), Point( 1, -7 ), Point( 2, -4 ) );
  tbl[ 2 ] = new Disque( Point( 4, 0 ), 2 );
  tbl[ 3 ] = new Disque( Point( 0, 5 ), 1 );
  Point clic( 2, -1 );
  for ( int i = 0; i < 4; ++i ) {
    Forme & f = *(tbl[ i ]); // référence la forme actuelle
    if ( f.boiteEnglobante().estDans( clic ) )
      std::cout << "clic est dans la boite englobante de " << i << std::endl;
  }
}
```

NB : Depuis C++11, le mot-clé `override` *existe pour spécifier qu'une méthode virtuelle redéfinit bien une méthode virtuelle définie dans une super-classe. Il n'est pas obligatoire d'utiliser le mot-clé, mais il permet d'améliorer la lisibilité du code ainsi que de laisser le compilateur vérifier qu'il existe bien une méthode correspondante dans la super-classe.*

```
struct A {
  virtual void f();
};
struct B : public A {
  virtual void f() override { ... } // ok
  virtual void g() override { ... } // invalide, pas de méthode g dans A
};
```

2.5 Surcharge

On parle de *surcharge* en C++ lorsqu'une fonction ou une méthode d'un nom donné est définie avec plusieurs signatures distinctes. Il y a surcharge lorsque le type des paramètres change et/ou leur nombre, ainsi que si le qualifieur `const` est mis ou non. Lorsque cette fonction ou méthode est appelée à un endroit du programme, le compilateur détermine dès la compilation quelle est la bonne fonction ou méthode surchargée à appeler. Il essaye à chaque fois de choisir la plus spécialisée (en un certain sens).

En conséquence, la surcharge permet donc la généricité puisque vous avez plusieurs variantes d'une même fonction. Elle permet surtout de simplifier (en apparence) les interfaces ou l'usage d'une classe. Ainsi, vous pouvez définir plusieurs constructeurs ou méthodes d'initialisation, qui portent tous le même nom. On n'aura pas à retenir qu'il existe une méthode `initFromX`, `initFromY`, et `initFromZ`, il suffira de savoir que `init` peut prendre indifféremment un `X`, un `Y` ou un `Z`.

Par exemple, on peut définir trois constructeurs pour Boite :

```
struct Boite {
  Boite( const Boite & other )
  : bg( other.bg ), hd( other.hd ) {}
  Boite( double xg, double yb, double xd, double yh )
  : bg( xg, yb ), hd( xd, yh ) {}
  Boite( Point any_bg, Point any_hd )
  : bg( any_bg ), hd( any_hd ) {}
...
};
```

On peut de plus surcharger tous les opérateurs usuels du C++, e.g. “+”, “-”, “[]“, “<<”, etc. Par exemple , l'opérateur << (en fait la fonction `operator<<`) est souvent surchargée pour qu'on puisse afficher simplement n'importe quel objet sur un flux de sortie. Sur la classe `Boite`, cela donnerait :

```
std::ostream& operator<<( std::ostream& out, const Point & p )
{
  out << "(" << p.x() << "," << p.y() << ")";
  return out;
}
std::ostream& operator<<( std::ostream& out, const Boite & b )
```

```
{
  out << "[Boite bg=" << bg << " hg=" << hg << "]";
  return out;
}
...
Boite b( -1, -2, 4, 2 );
std::cout << "b=" << b << std::endl; // appelle operator<<( std::ostream& , const
    Boite & )
```

La surcharge des opérateurs permet par exemple de définir une classe tableau de taille modifiable et d'avoir l'accès à une case via l'opérateur `[]`. Elle est aussi utile lorsqu'on fait du calcul scientifique. Ainsi on pourra, comme en mathématiques, surcharger les opérateurs d'addition, multiplication, division, pour les matrices et les vecteurs.

2.6 Transtypage

Dans une hiérarchie de classes, il est possible de transtyper directement un pointeur ou une référence d'une classe fille vers une classe mère.

```
Forme* f = new Disque( ... ); // ok
Triangle t( ... );
Forme& f2 = t;                // ok
```

A l'inverse, un changement de type d'une classe mère vers une classe fille doit être spécifié à l'aide du mot-clé `dynamic_cast`, et n'est possible que si la classe mère est polymorphe.

```
// dynamic_cast
#include <iostream>
#include <exception>
using namespace std;

class CBase { virtual void dummy() {} };
class CDerived: public CBase { int a; };

int main () {
    CBase * pba = new CDerived;
    CBase * pbb = new CBase;
    CDerived * pd;

    pd = dynamic_cast<CDerived*>(pba); // rien n'est affiché en-dessous
    if (pd==0) cout << "Null pointer on first type-cast" << endl;

    pd = dynamic_cast<CDerived*>(pbb); // le message ci-dessous est affiché
    if (pd==0) cout << "Null pointer on second type-cast" << endl;
    return 0;
}
```

On note que `dynamic_cast` retourne bien le pointeur sur l'objet dans son type dérivé ou 0 si l'objet n'était pas de ce type.

NB : Pour utiliser `dynamic_cast`, il faut en général préciser au compilateur que vous voulez avoir les informations sur les types à l'exécution (*run-time type information* ou RTTI). Avec g++, il faut donc ajouter l'option `-frtti` pour activer les RTTI. On peut aussi avoir des informations sur un type via l'opérateur `typeid`.

2.7 Héritage multiple

Il est possible en C++. Il suffit de rajouter d'autres classes séparées par une virgule lors de la définition de la classe dérivée.

```
struct A {};
struct B {};
struct C : public A, protected B {};
// C dérive publiquement de A, et cache pour ses seules filles l'héritage de B.
```

On peut aussi traiter le problème de l'héritage en diamant via l'héritage *virtuel*. Une lecture de `http://en.wikipedia.org/wiki/Virtual_inheritance` traite de ce point.

En résumé : Programmation orientée objet ; redéfinition contre surcharge

— L'encapsulation se fait en choisissant pour tout membre son domaine publique, protégé ou privé.
— Les méthodes et attributs sont liés par défaut à un objet, on utilise `static` pour les lier à une classe.
— On indique qu'une classe B hérite d'une classe A avec la notation " `:`".
— La relation d'héritage matérialise soit la relation "est un" (héritage publique) soit la relation "s'implémente sous la forme de" (héritage protégé ou privé).
— Manipuler une collection hétéroclite d'objets qui dérivent tous d'une super-classe `A` nécessite ensuite l'utilisation de pointeurs `A*` ou de références `a&`.
— Le polymorphisme dynamique permet de manipuler de façon homogène des objets de types différents.
— Cela se fait en redéfinissant les méthodes virtuelles dans les sous-classes. A l'exécution le programme détermine quel est le type réel pointé ou référencé pour appeler la bonne méthode.
— Une classe abstraite C++ est une classe dont au moins une méthode virtuelle est forcée à être redéfinie.
— La *surcharge* d'une fonction, d'une méthode ou d'un opérateur permet de traiter de façon unifiée des paramètres différents. L'appel à la bonne fonction, méthode ou opérateur, se résoud dès l'étape de compilation.
— On peut transtyper implicitement un pointeur ou une référence d'une classe fille vers un pointeur ou une référence d'une classe mère. Dans l'autre sens, on utilise `dynamic_cast` afin d'être sûr que le transtypage est valide (retourne 0 sinon).
— l'héritage multiple est tout à fait possible en C++, il suffit de mettre plusieurs classes après le " `:`".

Point de vue

Réfléchissez longuement avant de faire de l'héritage (multiple ou non). Très souvent, un mécanisme de délégation correspond mieux à ce que vous voulez faire, et permet aussi de changer dynamiquement les relations. De plus, le polymophisme induit un surcoût en temps non négligeable dans certains cas. La programmation générique a tendance à éliminer la nécessité de la relation d'héritage. Si vraiment vous pensez qu'un héritage est nécessaire ou utile, favorisez l'héritage d'interface(s) (au sens JAVA) plutôt que de classes avec des données.

3 Mécanismes d'allocation mémoire

Le C++ fournit comme le langage C deux mécanismes d'allocation mémoire pour les variables ou objets :

Allocation statique. Celle-ci est faite automatiquement par le compilateur à la déclaration de la variable. La zone mémoire allouée est sur la pile d'exécution du fils d'exécution. A la sortie du bloc où la variable a été déclarée, la zone mémoire de la variable est automatiquement désallouée. La durée de vie de la variable est donc limitée au bloc de définition.

Allocation dynamique. Celle-ci est nécessairement spécifiée par l'utilisateur par le biais d'un appel à l'opérateur `new`. Celui-ci retourne l'adresse du zone mémoire où la ou les variables ont été instanciés. La zone mémoire allouée est sur le tas du processus. A la sortie du bloc où l'opérateur `new` a été appelé, la zone mémoire allouée n'est pas désallouée et continue à exister. Ce n'est qu'à l'appel de l'opérateur `delete` que la zone est effectivement désallouée et rendue au système. La durée de vie de la variable ou des variables pointées n'est donc pas limitée au bloc de définition.

3.1 Allocation statique

C'est le mode par défaut d'allocation des variables. Lorsqu'il s'agit d'une classe ou une structure, on note qu'un constructeur est appelé au moment de la déclaration de la variable. A la fin du bloc de définition, les destructeurs des variables sont appelés puis leurs zones mémoires sont automatiquement désalloués.

```
struct A {
  ~A() { std::cout << "A::~A()" << std::endl; }
  A() { std::cout << "A::A()" << std::endl; }
  A( int ) { std::cout << "A::A( int )" << std::endl; }
  A( const A& ) { std::cout << "A::A( const A& )" << std::endl; }
};
{
  A a1;        // "A::A()"
  A a2( 5 );   // "A::A( int )"
  A a3( a1 );  // "A::A( const A& )"
  A a4 = a1;   // "A::A( const A& )" aussi
  A as[ 3 ];   // 3 fois "A::A()"
  A& ref = a1;// rien, ce n'est qu'une référence
  A* ptr =&a2;// rien, ce n'est qu'un pointeur
}              // 7 fois "A::~A()"
```

3.2 Allocation dynamique

Elle se fait avec les opérateurs `new` et `delete` pour une variable, et `new[]` et `delete[]` pour les tableaux.

Les opérateurs `new` alloue l'espace mémoire correspondant (qui dépend bien sûr de la taille mémoire du type et du nombre de cases du tableaux), appellent le(s) constructeur(s) spécifié(s) ou par défaut, puis retourne l'adresse en mémoire correspondante.

Les opérateurs `delete` appellent le(s) destructeur(s) puis désalloue l'espace mémoire correspondant et le rende au système.

```
A* alloueA()
{
  A* p = new A( 5 ); // "A::A( int )"
  return p;
} // variable p détruite, mais pas l'objet instancié.
A* alloueTblA( int t )
{
  A* p = new A[ t ]; // t fois "A::A()"
  return p;          // p est l'adresse de la première case du tableau
} // variable p détruite, mais pas les objets instanciés.
{
  A* p1 = new A;      // "A::A()"
  A* p2 = alloueA(); // "A::A( int )", p2 pointe vers un A valide
  A* pT = alloueTblA( 3 ); // 3 fois "A::A()"
  A& ref = pT[ 0 ];  // rien, ref référence la variable pT[ 0 ]
  delete p1;         // "A::~A()" la variable pointée par p1 est non valide
  delete[] pT;       // 3 fois "A::~A()": ref est aussi invalidée.
} // Fuite mémoire: il n'y a plus de pointeur qui pointe vers là où pointait p2.
```

3.3 Surcharge des opérateurs `new` et `delete`

Vous pouvez redéfinir ces opérateurs pour toute classe que vous écrivez. Si on suppose que vous avez prévu un allocateur mémoire spécifique plus rapide offrant les fonctions `allocate_from_pool` et `release`, on peut les appeler ainsi :

```
#include <new> // for size_t
class A {
public:
    A();
    ~A();
    static void* operator new (size_t size);
    static void operator delete (void *p);
};
void* A::operator new (size_t size)
```

```
{
 void *p=allocate_from_pool(size);
 return p;
} // le constructeur de A est appelé implicitement ici.
void A::operator delete (void *p)
{
 release(p); // return memory to pool
} // le destructeur de A est appelé implicitement ici.
```

En résumé : Mécanismes d'allocation mémoire

- Les variables sont allouées statiquement par le compilateur sur la pile d'exécution. Elles sont automatiquement détruites à la fin du bloc de définition. L'appel aux constructeurs et destructeurs est systématique.
- Si on veut avoir des objets dont la durée de vie dépasse celle du bloc de définition, on peut utiliser l'allocation dynamique. Cela se fait via les opérateurs `new` et `delete`, et `new[]` et `delete[]` pour les tableaux.
- Attention à ne pas oublier l'appel à `delete` lorsque vous n'avez plus besoin de l'objet.

Point de vue

L'allocation statique doit être la norme. Très souvent, on peut éviter l'allocation dynamique. Même si vous souhaitez avoir des conteneurs de taille variable, il est préférable de passer par les conteneurs standards de la STL comme `vector`, `set`, `map`, `list`, etc. En général, un de ces conteneurs répond déjà à votre problème.
Si vous devez faire de l'allocation dynamique, mon point de vue est qu'elle doit être systématiquement encapsulée dans une classe, avec allocation à la construction de l'objet et désallocation à la destruction de l'objet. De cette façon, vous éviterez systématiquement les fuites de mémoire.

4 Patrons de fonctions et patrons de classes

En programmation, la généricité d'une fonction repose sur son indépendance vis-à-vis du type, et éventuellement du nombre de ses arguments. C'est un concept important pour un langage de haut niveau car il permet d'augmenter le niveau d'abstraction du langage. Plusieurs mécanismes visant à permettre l'écriture de fonctions génériques ont donc été mis en œuvre par les différents langages de programmation. Nous avons vu avant comment le polymorphisme dynamique était une possible réponse au besoin de généricité. L'inconvénient de cette approche est que le programme passe beaucoup de temps à déterminer quel est le bon type. De plus, ce type de généricité impose une dispersion des données, ce qui nuit à l'efficacité des caches processeurs.

Les fonctions et surtout les classes patrons (classes *template*, ou classes *paramétrées*, ou modèles) constituent l'outil essentiel pour faire de la *programmation générique* efficace. La différence essentielle avec le polymorphisme usuel est que les types sont fixés dès la compilation, ce qui évite d'une part les tests de type et d'autre part favorise la contiguïté des données en mémoire.

4.1 Ecriture polymorphe contre écriture paramétrée

Nous proposons deux programmes pour évaluer les performances d'une écriture polymorphe (genre JAVA où tout objet qui dérive de Comparable peut être trié) par rapport à une écriture paramétrée (ou générique). Nous avons écrit une procédure de tri dit "à bulle" dans les deux cas. L'algorithme n'est pas optimal, mais ce n'est pas la question ici.

Tout d'abord, l'écriture polymorphe requiert la définition d'une super-classe abstraite `Comparable` qui force le type dérivé à avoir l'opérateur `<`. Ensuite, la classe `Point` redéfinit cet opérateur proprement. On note ensuite que, pour exploiter le polymrphisme, on est obligé de manipuler un tableau

de `Comparable*`, ce qui induit une écriture un peu lourde, mais possible, du tri à bulle. La fonction `bubbleSort` pourra trier n'importe quel tableau de `Comparable*`, ce qui la rend tout à fait générique.

```
// file polymorph-sort.cpp
// Tri à la sauce polymorphe dynamique.
#include <cstdlib>
#include <iostream>

struct Comparable {
  virtual bool operator<( const Comparable& other ) const = 0;
  virtual ~Comparable() {}
};

struct Point : public Comparable {
  double x;
  double y;
  Point()
  { // par défaut, point aléatoire dans [0,1]x[0,1]
    x = (double) random() / (double) RAND_MAX;
    y = (double) random() / (double) RAND_MAX;
  }
  virtual ~Point() {}
  virtual bool operator<( const Comparable& other ) const
  {
    const Point & otherp = dynamic_cast< const Point & >( other );
    return ( x < otherp.x ) || ( ( x == otherp.x )
                                 && ( y < otherp.y ) );
  }
};

// debut est l'adresse de la première case du tableau de Comparable*
// fin est l'adresse après la dernière case du tableau de Comparable*
void bubbleSort( Comparable* debut[], Comparable* fin[] )
{
  for ( Comparable** loop1 = debut; loop1 != fin-1; ++loop1 )
    for ( Comparable** loop2 = fin-1; loop2 != loop1; --loop2 )
      {
        Comparable & c1 = **(loop2-1);
        Comparable & c2 = **loop2;
        if ( c2 < c1 )
          std::swap( *loop2, *(loop2-1) ); // tri indirect en fait.
      }
}
int checkSort( Comparable* debut[], Comparable* fin[] )
{
  for ( Comparable** loop = debut; loop != fin-1; ++loop )
    {
      Comparable & c1 = **loop;
      Comparable & c2 = **(loop+1);
      if ( c2 < c1 ) return loop-debut;
    }
  return fin-debut;
}
int main( int argc, char* argv[] )
{
  int nb = argc > 1 ? atoi( argv[ 1 ] ) : 1000;
  // Création du tableau.
  Comparable** tbl = new Comparable*[ nb ];
  for ( int i = 0; i < nb; ++i )
    tbl[ i ] = new Point;
  // Tri du tableau
  std::cout << "Tableau trie jusqu'a " << checkSort( tbl, tbl + nb ) << std::endl;
  bubbleSort( tbl, tbl + nb );
  std::cout << "Tableau trie jusqu'a " << checkSort( tbl, tbl + nb ) << std::endl;

  // Destruction du tableau.
  for ( int i = 0; i < nb; ++i )
    delete tbl[ i ];
  delete[] tbl;
}
```

```
//-------------------------------------------------------------------------
// avec dynamic_cast
// $ time ./polymorph-sort 20000
// Tableau trie jusqu'a 0
// Tableau trie jusqu'a 20000
// real 0m4.048s
// user 0m4.032s
// sys  0m0.004s

//-------------------------------------------------------------------------
// en remplaçant dynamic_cast par static_cast
// $ time ./polymorph-sort 20000
// Tableau trie jusqu'a 0
// Tableau trie jusqu'a 20000

// real 0m1.672s
// user 0m1.660s
// sys  0m0.004s
```

Après compilation avec optimisation maximum (`gcc -O3 ...`), le temps d'exécution pour 20000 éléments est de 4s environ si on utilise `dynamic_cast` (ce serait la façon propre de faire les choses). On peut forcer un transtypage non dynamique (i.e. sans vérification) avec `static_cast`, ce qui est valide ici car tous nos `Comparable*` sont bien des `Point*`. Dans ce cas, le temps de calcul est ramené à 1.67s.

L'écriture paramétrée ou générique du tri à bulle repose simplement sur la définition d'une fonction patron paramétré par le type de chaque élément. Ce type est déduit à l'appel de la fonction par le compilateur, qui fait lui-même l'association `T=Point` lors de l'appel des fonctions paramétrées `checkSort` et `bubbleSort`. Presque toutes les écritures génériques sont simplifiées, car il n'y a plus besoin de manipuler les pointeurs. (On pourrait bien sûr aussi trier un tableau de pointeurs.) Cela donne le code suivant :

```
// file generic-sort.cpp
// Tri à la sauce générique/classe patron
#include <cstdlib>
#include <iostream>

struct Point {
  double x;
  double y;
  Point()
  { // par défaut, point aléatoire dans [0,1]x[0,1]
    x = (double) random() / (double) RAND_MAX;
    y = (double) random() / (double) RAND_MAX;
  }

  bool operator<( const Point& other ) const
  {
    return ( x < other.x ) || ( ( x == other.x )
                                && ( y < other.y ) );
  }
};

// debut est l'adresse de la première case du tableau de T
// fin est l'adresse après la dernière case du tableau de T
template <typename T>
void bubbleSort( T debut[], T fin[] )
{
  for ( T* loop1 = debut; loop1 != fin-1; ++loop1 )
    for ( T* loop2 = fin-1; loop2 != loop1; --loop2 )
      {
        T & c1 = *(loop2-1);
        T & c2 = *loop2;
        if ( c2 < c1 )
          std::swap( c1, c2 ); // Tri direct
      }
}

template <typename T>
int checkSort( T debut[], T fin[] )
```

```
{
  for ( T* loop = debut; loop != fin-1; ++loop )
    {
      T & c1 = *loop;
      T & c2 = *(loop+1);
      if ( c2 < c1 ) return loop-debut;
    }
  return fin-debut;
}

int main( int argc, char* argv[] )
{
  int nb = argc > 1 ? atoi( argv[ 1 ] ) : 1000;
  // Création du tableau.
  Point* tbl = new Point[ nb ];
  // Tri du tableau
  std::cout << "Tableau trie jusqu'a " << checkSort( tbl, tbl + nb ) << std::endl;
  bubbleSort( tbl, tbl + nb );
  std::cout << "Tableau trie jusqu'a " << checkSort( tbl, tbl + nb ) << std::endl;

  // Destruction du tableau.
  delete[] tbl;
}
// $ time ./generic-sort 20000
// Tableau trie jusqu'a 0
// Tableau trie jusqu'a 20000

// real 0m0.912s
// user 0m0.908s
// sys  0m0.000s
```

Après compilation avec optimisation maximum (`gcc -O3 ...`), le temps d'exécution pour 20000 éléments est de 0.912s environ.

Par souci de complétude, on donne ci-dessous un code "équivalent" au code polymorphe, qui passerait par un tableau de pointeurs et un tri indirect. On note qu'il s'exécute toujours en un temps moindre que le code polymorphe (1.2s) mais reste, du fait de la dispersion en mémoire, moins efficace que le code utilisant juste un tableau de `Point`.

```
// file generic-sort2.cpp
// Tri à la sauce générique/classe patron en gardant un tableau de pointeurs
// Tri indirect en somme.
#include <cstdlib>
#include <iostream>

struct Point {
  double x;
  double y;
  Point()
  { // par défaut, point aléatoire dans [0,1]x[0,1]
    x = (double) random() / (double) RAND_MAX;
    y = (double) random() / (double) RAND_MAX;
  }

  bool operator<( const Point& other ) const
  {
    return ( x < other.x ) || ( ( x == other.x )
                                && ( y < other.y ) );
  }
};

template <typename T>
void bubbleSort( T* debut[], T* fin[] )
{
  for ( T** loop1 = debut; loop1 != fin-1; ++loop1 )
    for ( T** loop2 = fin-1; loop2 != loop1; --loop2 )
      {
        T & c1 = **(loop2-1);
        T & c2 = **loop2;
        if ( c2 < c1 )
          std::swap( *loop2, *(loop2-1) ); // tri indirect en fait.
```

```
      }
}

template <typename T>
int checkSort( T* debut[], T* fin[] )
{
  for ( T** loop = debut; loop != fin-1; ++loop )
    {
      T & c1 = **loop;
      T & c2 = **(loop+1);
      if ( c2 < c1 ) return loop-debut;
    }
  return fin-debut;
}

int main( int argc, char* argv[] )
{
  int nb = argc > 1 ? atoi( argv[ 1 ] ) : 1000;
  // Création du tableau.
  Point** tbl = new Point*[ nb ];
  for ( int i = 0; i < nb; ++i )
    tbl[ i ] = new Point;
  // Tri du tableau
  std::cout << "Tableau trie jusqu'a " << checkSort( tbl, tbl + nb ) << std::endl;
  bubbleSort( tbl, tbl + nb );
  std::cout << "Tableau trie jusqu'a " << checkSort( tbl, tbl + nb ) << std::endl;

  // Destruction du tableau.
  for ( int i = 0; i < nb; ++i )
    delete tbl[ i ];
  delete[] tbl;
}
// $ time ./generic-sort2 20000
// Tableau trie jusqu'a 0
// Tableau trie jusqu'a 20000
// real 0m1.208s
// user 0m1.204s
// sys  0m0.000s
```

4.2 Patron de fonction ; fonction générique

Toute fonction peut être paramétrée par un ou plusieurs types, ou par une constante entière ou booléenne. On parle de *patron de fonction*, *modèle de fonction* ou *fonction générique*, et *template function* en anglais. Ces paramètres sont réutilisables dans toute la fonction. C'est lors de l'appel d'une telle fonction que les paramètres sont fixés. Le compilateur produit donc autant de codes différents qu'il y a d'appels de cette fonction avec des types différents. On prend l'exemple du maximum de deux nombres. Voilà comment définir une fonction générique :

```
template <typename T>                // T est un type (presque) quelconque
T                                    // la valeur de retour est du même type
max( const T & t1, const T & t2 ) // les paramètres de type T sont référencés
{
  return ( t1 < t2 ) ? t2 : t1;    // Le type T doit avoir l'opérateur< et
}                                    // le constructeur de copie (pour retourner
                                     // la valeur)
...
{
  int j = max( 4, 17 );              // appelle la fonction max<int>
  double x = max( 3.5, 14.2 );       // appelle la fonction max<double>
}
```

On voit que le compilateur trouve tout seul les types paramétrés si cela lui est possible (ici `int` ou `double`). Les lignes ci-dessous montrent que parfois il ne peut pas le faire, par exemple parceque les types ne coïncident pas directement et nécessite une conversion (même implicite). On voit aussi que l'on peut préciser soit même les paramètres de type lors de l'appel de la fonction générique (cas (3)).

```
double y = max( 2, 1.5 );          // (1) non: 2 de type int et 17.3 de type double
double z = max( (double) 2, 1.5 );// (2) oui: appelle la fonction max<double>
double t = max<double>( 2, 1.5 ); // (3) oui: appelle la fonction max<double>
```

Le type réel instancié étant connu dès la compilation de la fonction, le compilateur peut optimisé à loisir la fonction comme si on l'avait écrite directement avec ce type là. En revanche, pour le programmeur, il n'a écrit qu'un seul fois sa fonction et peut s'en servir avec d'autres types à divers endroits du programme.

4.3 Premiers concepts ; premiers modèles

Reprenons l'exemple précédent. Le programmeur peut se servir de la fonction générique `max` avec n'importe quel type, mais attention, du moment *qu'il satisfait la syntaxe et la sémantique d'utilisation de ce type dans la fonction générique.* Ici, *implicitement*, la fonction `max` utilise deux méthodes/opérateurs du type `T` :

— Un opérateur de comparaison sur `T` avec la sémantique de "inféreur strict". Il faut donc que `T` ait défini `bool T::operator<( const T&) const`.

— Le constructeur par copie avec la sémantique de clonage. Il faut donc que `T` ait défini '`T::T( const T&)`.

On appelle *concept* ces contraintes syntaxiques (opérateurs présents) et sémantiques (ils ont un sens donné) requises pour une fonction (ou classe) générique donnée. On donne un nom à chacun de ces concepts. Par exemple, on dirait ici que `T` doit être `CopyConstructible` and `LessThanComparable` (oui, on les dit presque toujours en anglais).

Ainsi le code suivant ne compile pas au moment de l'appel de la fonction `max` car `Chou` n'est pas `LessThanComparable`.

```
struct Chou {
  Chou( const char* nom ) : mNom( nom ) {}
  Chou( const Chou & autre ) // Chou est CopyConstructible
  : mNom( autre.mNom ) {}
  const char* mNom;
}; // Chou n'est pas LessThanComparable

{
  Chou rouge("Chou rouge");
  Chou brux("Chou de Bruxelles");
  Chou melange = max( rouge, brux );
}
```

Cela donne à la compilation :

```
$prompt$ g++ generic-max.cpp
generic-max.cpp: In function 'T max(const T&, const T&) [with T = Chou]':
generic-max.cpp:25:35:   instantiated from here
generic-max.cpp:5:29: erreur: no match for 'operator<' in 't1 < t2'
```

Ici, le compilateur indique clairement que l'opérateur `operator<` n'est pas défini pour le type `T=Chou`. Malheureusement, et c'est un gros défaut des patrons de fonction ou de classe en C++, les erreurs du compilateur sont parfois beaucoup plus obscures.

On dira donc que `int` et `double` sont des *modèles* des concepts `CopyConstructible` et `LessThanComparable`, tandis que `Chou` est un modèle de `CopyConstructible` mais pas de `LessThanComparable`.

La définition propre de concept associé à une fonction générique est un problème souvent délicat, lié à la capacité d'abstraction d'un algorithme. La vérification qu'un modèle satisfait un concept était un problème délicat en C++ jusqu'à C++20. On verra quelques moyens de le faire dans la section 6 et comment c++20 simplifie considérablement la définition et l'utilisation des concepts.

4.4 Déclaration et définition des patrons de fonction

Idéalement, en C++, on découple interface et implémentation dans fichers entête et fichiers source. Malheureusement, le compilateur a en général besoin du code complet d'une fonction générique pour vérifier qu'il peut bien la générer. Le mot-clé `export` était supposé permettre de déclarer une fonction générique dans un fichier entête, tandis que la définition se fait dans un fichier source ailleurs. En pratique, un seul compilateur l'avait intégré, et le standard de normalisation a décidé de rendre ce mot-clé obsolète au vu des difficultés rencontrées. La pratique en C++ est donc soit :

1. de tout mettre dans le fichier entête en mélangeant déclaration et définition,

```
// file max.hpp
template <typename T>
```

```
T max( const T & t1, const T & t2 ) // définition
{
  return ( t1 < t2 ) ? t2 : t1;
}
```

2. de faire la déclaration dans le fichier entête, puis de faire la définition soit plus bas dans le même fichier dans un fichier spécial (fichier `.tpp`, `.ih`, etc) qui est inclus dans le fichier entête `.hpp`.

```
// file max.hpp
template <typename T>
T max( const T & t1, const T & t2 ); // déclaration
...
#include "max.tpp"
```

```
// file max.tpp
template <typename T>
T max( const T & t1, const T & t2 ); // définition
{
  return ( t1 < t2 ) ? t2 : t1;
}
```

4.5 Fonctions génériques à plusieurs paramètres

Il est tout à fait possible de spécifier plusieurs paramètres génériques à un patron de fonction ou de classe. On les sépare avec une "`,`" et on leur donne des noms différents. Voilà l'exemple d'une fonction `pour_chacun_faire` qui applique un foncteur à chacun des éléments d'un intervalle de données (aussi `std::for_each`).

```
template <typename Iterateur, typename Fonction>
void pour_chacun_faire( Iterateur it, Iterateur itEnd, Fonction & f )
{
  for ( ; it != itEnd; ++it )
    f( *it );
}
struct Accumulateur {
  Accumulateur() : cumul( 0.0 ) {}
  void operator()( double v )
  { cumul += v; }

  double cumul;
}
{
  double values[] = { 4.5, 12.3, -2.5, 8.4, 9.8 };
  Accumulateur acc;
  pour_chacun_faire( values, values + 5, acc );
  std::cout << "Somme=" << acc.cumul << std::endl;
}
```

On note que la fonction `pour_chacun_faire` nécessite deux types, chacun devant être le modèle d'un concept différent :

Itérateur. De façon générale, un itérateur est un objet qui permet d'accéder à une valeur (opérateur d'indirection), qui permet de passer à la valeur suivante (opérateur de pré ou post-incrémentation) et qui peut être comparé à un autre itérateur (opérateur d'égalité ou d'inégalité). En anglais, on dit *Iterator*.

Fonction. Une fonction est un objet qui possède l'opérateur parenthèses "()" (i.e. appel de fonction avec paramètre). Ici il faut que le type de paramètre soit le même que le type de la valeur de l'itérateur.

On vérifie aisément qu'un pointeur `double*` est un modèle d'`Iterateur` tandis que la structure `Accumulateur` est un modèle de `Fonction` avec paramètre `double`.

NB : Il serait facile de faire un `Compteur` sur le même principe que l'`Accumulateur`, par exemple un compteur des valeurs positives.

4.6 Patrons de classe ou classes génériques

Le C++ permet aussi de définir des *patrons de classe*, *classes paramétrées* ou *classes génériques*. Cela se fait aussi via le mot-clé `template`. De façon générale, la syntaxe est :
`template <` *liste-de-typename-typeid-ou-integral-id* `>`
(`class` or `struct`) *Nom-de-la-classe* `{`
...On utilise les typeid et id comme on veut.
`};`

Par exemple, vous voulez définir un type "paire de n'importe quoi" (déjà fait dans la `STL` sous le nom `std::pair`). On écrirait :

```
#include <iostream>
template <typename First, typename Second>
struct Paire {
  Paire( const First & f, const Second & s )
  : first( f ), second( s ) {}

  First first;
  Second second;
};

int main()
{
  typedef Paire<int,const char*> MaPaire;
  MaPaire recette[ 4 ];
  recette[ 0 ] =  MaPaire( 3, "oeufs" );
  recette[ 1 ] =  MaPaire( 250, "g de farine" );
  recette[ 2 ] =  MaPaire( 50, "ml de lait" );
  recette[ 3 ] =  MaPaire( 1, "pincée de sel" );
  std::cout << "Pour faire des crêpes, mélangez:" << std::endl;
  for ( unsigned int i = 0; i < 4; ++i )
    std::cout << "- " << recette[ i ].first
              << " " << recette[ i ].second << std::endl;
}
```

On note là encore que les types `First` et `Second` sont au choix de l'utilisateur mais doivent satisfaire des concepts (certes assez légers) :

`DefaultConstructible` pour que le compilateur puisse écrire le constructeur par défaut.

`Assignable` pour que le compilateur puisse écrire l'affectation par défaut.

`CopyConstructible` pour que le compilateur puisse écrire le constructeur par copie.

Les classes génériques permettent notamment d'écrire des conteneurs génériques comme les tableaux de taille variable, listes, ensembles, etc. On donne ici l'exemple du vecteur de valeur de dimension fixée.

```
// file FArray.hpp
#ifndef _FARRAY_H_
#define _FARRAY_H_
#include <cassert>

// Modélise un simple tableau à N éléments de Valeur.
// Teste en plus les bornes.
template <typename Valeur, int N>
struct FArray {
  // On garde tous les constructeurs, affectation et destructeur par
  // défaut.
  FArray() {}

  // Opérateur [] en lecture
  inline
  const Valeur & operator[]( int index ) const
  {
    assert( ( 0 <= index ) && ( index < N ) );
    return mData[ index ];
  }
  // Opérateur [] en écriture
  inline
```

```
  Valeur & operator[]( int index )
  {
    assert( ( 0 <= index ) && ( index < N ) );
    return mData[ index ];
  }

private:
  Valeur mData[ N ]; // taille fixée à la compilation
};

#endif
```

```
// file testFArray.cpp
#include <iostream>
#include <cmath>
#include "FArray.hpp"

int main()
{
  typedef FArray<double,3> Ligne;
  typedef FArray<Ligne,3> Matrice;
  Matrice m;
  m[ 0 ][ 0 ] =  cos( M_PI/4.0 ); m[ 0 ][ 1 ] = sin( M_PI/4.0 );  m[ 0 ][ 2 ] =
      0.0;
  m[ 1 ][ 0 ] = -sin( M_PI/4.0 ); m[ 1 ][ 1 ] = cos( M_PI/4.0 );  m[ 1 ][ 2 ] =
      0.0;
  m[ 2 ][ 0 ] = 0;                m[ 2 ][ 1 ] = 0;                 m[ 2 ][ 2 ] =
      1.0;

  Matrice m2 = m;
  for ( int i = 0; i < 3; ++i )
    {
      for ( int j = 0; j < 3; ++j )
        std::cout << m2[ i ][ j ] << " ";
      std::cout << std::endl;
    }
}
```

On note une certaine lourdeur à l'initialisation. Ce problème est soluble via les *initializer-list* à partir de c++0x (cf. Annexe).

4.7 Spécialisation des patrons de classes

Pourquoi voudrait-on spécialiser alors qu'on vient de voir comment écrire générique ? Simplement pour conserver une interface commune et, dans des cas bien précis, spécialiser le code par exemple pour faire les choses plus efficacement en temps, en mémoire, ou tout simplement parce que le code doit vraiment être différent dans un cas précis.

`template <` *liste-de-typename-typeid-ou-integral-id* `>`
(`class` or `struct`) *Nom-de-la-classe* `<` *liste-de-types-ou-constantes* `>{`
...On utilise les typeid et id comme on veut.
`};`

Si on veut spécialiser `FArray` pour une taille 1, on écrirait :

```
template <typename Valeur>
struct FArray<Valeur,1> {
... };
```

On réécrit alors toute la classe pour la spécialisation souhaitée. Cela donne le code ci-dessous. Notez que les fonctions ajoutées permettent de créer directement un FArray à partir d'une seule valeur ou de le convertir en valeur.

```
// Spécialise le tableau à 1 élément de Valeur.
template <typename Valeur>
struct FArray<Valeur,1> {
  // On garde tous les constructeurs, affectation et destructeur par
  // défaut.
  inline FArray() {}
  // Constructeur à partir de val
```

```
  inline FArray( const Valeur & v )
    : mData( v ) {}
  // Opérateur cast vers Valeur
  inline
  operator Valeur() const
  { return mData; }

  // Opérateur [] en lecture
  inline
  const Valeur & operator[]( int /*index*/ ) const
  {
    return mData;
  }
  // Opérateur [] en écriture
  inline
  Valeur & operator[]( int /*index*/ )
  {
    return mData;
  }

private:
  Valeur mData; // pas besoin de tableau
};
```

On aurait pu spécialiser la classe pour les seules valeurs qui sont des pointeurs :

```
template <typename Valeur, int N>
struct FArray<Valeur*,N> {
... };
```

Une spécialisation utile est celle dans le cas du tableau de `bool`, car on peut économiser considérablement en mémoire en stockant juste un bit par valeur. Cela donne :

```
// Modélise un simple tableau à N éléments de bool.
// Teste en plus les bornes.
template <int N>
struct FArray<bool, N> {
  // On garde tous les constructeurs, affectation et destructeur par
  // défaut.
  FArray() {}

  // Opérateur [] en lecture
  inline
  bool operator[]( int index ) const
  {
    assert( ( 0 <= index ) && ( index < N ) );
    return ( mData[ index >> 3 ] >> ( index % 8 ) ) & 0x1;
  }
  // Opérateur [] en écriture... plus compliqué... il faut passer par
  // une autre classe... omis ici
  inline
  Proxy_bool operator[]( int index )
  { ... }

private:
  unsigned char mData[ (N+7) / 8 ]; // taille fixée à la compilation
};
```

4.8 Spécialisation des patrons de fonctions

C'est possible, mais il faut tout spécialiser (ce n'est pas possible partiellement, donc dans l'exemple ci-dessus il faut choisir la `Valeur` et le `N`). Cela fonctionne similairement à la spécialisation des patrons de classes du point de vue syntaxique. Néanmoins, pour que le compilateur ne se mêle pas les pinceaux en cherchant la fonction la plus spécialisée pour votre problème par rapport aux différentes surcharges de votre fonction, il est préférable soit :

1. de faire de la simple surcharge de fonctions (possible en C++), et jamais de la spécialisation de patrons de fonction. Par exemple on peut définir la version suivante pour `max<bool>` :

   ```
   bool max( bool a, bool b )
   { return a || b; }
   ```

2. de passer par une classe “annexe” pour faire les spécialisations. Vous utiliserez alors les mécanismes de spécialisation des classes pour spécialiser votre code. Il n'y a pas d'interférence entre la spécialisation et la surcharge dans le cas des classes. Cela pourrait donner dans le cas du `max` :

```
// Déclaration d'un patron de classe (et non définition).
template <typename T> struct MaxImpl;

// Notre fonction générique que l'on souhaite spécialiser.
template <typename T>
T
max( const T & t1, const T & t2 )
{ return MaxImpl<T>::f( t1, t2 ); } // utilisateurs, n'y touchez pas !

// utilisateurs, spécialisez cette méthode de classe.
template <class T>
struct MaxImpl
{
  static
  T f( const T & t1, const T & t2 )
  {
    return ( t1 < t2 ) ? t2 : t1;
  }
};

// Par exemple, spécialisation avec bool (true > false).
template <>
struct MaxImpl<bool>
{
  static
  bool f( bool t1, bool t2 )
  {
    return t1 || t2;
  }
};
```

En résumé : Patrons de fonctions et patrons de classes

— Les fonctions et classes patron forment l'outil de base de la programmation générique.

— Le polymorphisme induit est décidé dès la compilation. Il est systématiquement plus efficace que le polymorphisme dynamique. Il est généralement aussi efficace que l'écriture spécialisée dans un type donné.

— Les patrons de fonctions ou fonctions génériques ont aussi des paramètres de type ou valeur entière qui sont déterminés à la compilation. Une même fonction peut donc avoir plusieurs écritures spécialisées, chacune étant adaptée à un ensemble de types précis. Une seule écriture = plusieurs fonctions.

— Pour que la fonction générique garde un sens, il faut évidemment que les types donnés en paramètres satisfassent quelques contraintes, rassemblées sous le nom de concept. On dit que le type est un modèle du concept lorsqu'il les satisfait.

— Les patrons de fonctions et de classes utilisent la syntaxe `template <` *liste* `>`. La *liste* peut comporter des noms de types (précédés du mot-clé `typename`) ou des constantes entières ou booléennes.

— On choisit les types paramétrés à l'instanciation de la fonction (le compilateur peut souvent trouver les types en regardant vos arguments) ou de la classe (on utilise les < et > pour préciser les types).

— On peut aussi spécialiser les patrons de classes, de manière partielle ou complète. Cela permet d'avoir du code spécifique dans certains cas, afin de gagner du temps de calcul, de la place mémoire ou de prendre en compte des cas particuliers.

Point de vue

Comprendre les concepts sous-jacents à un algorithme et les formaliser clairement est peut-être le travail essentiel de l'informaticien.
La syntaxe choisie pour les classes génériques en C++ induit *a posteriori* une écriture de code un peu lourde et verbeuse, à laquelle il faut malheureusement s'habituer vite.

5 Généricité et Standard Template Library (STL)

5.1 Conteneurs

Un *conteneur* est un objet qui stocke des données du même type (ou collection). Dans tout conteneur, on peut ajouter, modifier ou supprimer des données, ou rechercher des données. On peut aussi parcourir toutes les données ou une partie. Suivant les opérations que l'on cherche à effectuer (notamment le nombre de recherche, le nombre d'insertion ou de suppression ainsi que leur localisation), certains conteneurs seront plus efficaces que d'autres. Tous les conteneurs prennent une place en mémoire linéaire en le nombre de données stockées. Voici les conteneurs principaux en C++98 :

— `std::vector` un tableau de taille modifiable,
— `std::list` une séquence de valeurs,
— `std::stack` une pile de valeurs,
— `std::queue` une file de valeurs,
— `std::deque` une file à double-entrée de valeurs (en gros, une pile + une file en même temps),
— `std::priority_queue` une file à priorité de valeurs,
— `std::set` un ensemble de valeurs (sans duplicata),
— `std::multiset` un ensemble de valeurs (avec duplicata),
— `std::map` un tableau associatif clé/valeur (sans duplicata),
— `std::multimap` un tableau associatif clé/valeur (avec duplicata),

Le C++11 ajoute quelques conteneurs standards : `std::array`, `std::forward_list`, plus les variantes d'ensemble et de tableau associatif basé sur le principe des tables de hachage : `unordered_set`, `unordered_multiset`, `unordered_map`, `unordered_multimap`.

Tous les conteneurs servent à stocker des données de type homogène. Ils fournissent aussi des *itérateurs* pour parcourir les données, y accéder en lecture ou en écriture. Les *itérateurs* permettent ensuite l'écriture d'algorithmes génériques, car ils abstraient les méthodes pour les conteneurs en les réduisant juste à des opérations : avancer or réculer (incrémenter, décrémenter, ou aller directement à la n-ème position), obtenir ou modifier la valeur (déréférencement).

5.2 Conteneurs séquentiels

Ce sont `std::vector` et `std::list` (C++11 introduit `std::array`, `std::forward_list`). Les classes `std::stack`, `std::queue`, `std::deque` et `std::priority_queue` sont des adapteurs (de `std::vector`) pour un usage particulier.

Le conteneur le plus utilisé est sans conteste `std::vector`, qui correspond à un tableau de taille ajustable à l'exécution. Asymptotiquement, pour n éléments dont chaque instance coût b octets, la taille mémoire est nb. Le conteneur `std::list` prend plus de place car chaque cellule de la liste est alloué dynamiquement (par défaut). Ainsi, il faut plutôt compter sur une taille mémoire de $n(b+12)$ sur une architecture 32 bits, voire $n(b+20)$ sur une architecture 64 bits.

On suppose que la structure contient n éléments. Voilà les complexités des différentes opérations possibles.

opération	`vector`	`list`	`queue`	`stack`	`deque`
Insertion (début)	$O(n)$	$O(1)$		$O(1)$	$O(1)$
Insertion (fin)	$O(1)$	$O(1)$	$O(1)$		$O(1)$
Insertion (k-ème donné)	$O(n-k)$	$O(1)$			
Suppression (début)	$O(n)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$
Suppression (fin)	$O(1)$	$O(1)$			$O(1)$
Suppression (k-ème donné)	$O(n-k)$	$O(1)$			
Accès (début)	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$
Accès (fin)	$O(1)$	$O(n)$			$O(1)$
Accès (k-ème élément)	$O(1)$	$O(k)$			
Parcours (temps total)	$O(n)$	$O(n)$			

Attention, en pratique, les processeurs ont maintenant des unités de calcul vectoriel. Il est plus facile pour le compilateur d'optimiser le conteneur `vector` pour le calcul vectoriel, donc `list` est rarement plus rapide (il faut vraiment beaucoup d'éléments et d'insertion/suppression au milieu).

Tous les conteneurs séquentiels fournissent :

— des itérateurs bidirectionnels en lecture (type `const_iterator`) et en lecture/écriture (type `iterator`).

— des méthodes `begin` et des méthodes `end` pour obtenir un itérateur au début et après la fin.

— des itérateurs inversés avec les méthodes adéquates `rbegin` et `rend`

— les itérateurs de `vector` sont en plus des `RandomAccessIterator`, c'est-à-dire qu'ils autorisent les déplacements de n cases d'un coup.

```cpp
// vector::begin/end
#include <iostream>
#include <vector>

int main ()
{
  std::vector<int> myvector;
  for (int i=1; i<=5; i++) myvector.push_back(i);

  std::cout << "myvector contains:";
  for (std::vector<int>::iterator it = myvector.begin() ; it != myvector.end(); ++
      it)
    std::cout << ' ' << *it;
  std::cout << '\n';

  return 0;
}
```

Ce programme s'exécute ainsi :

```
myvector contains: 1 2 3 4 5
```

5.3 Conteneurs associatifs

Les conteneurs associatifs permettent de modéliser des ensembles d'éléments (appelés clés) ou des tableaux associatifs associant une valeur à tout élément (appélé clé). Il faut un opérateur de comparaison entre clés, soit donné par l'opérateur inférieur, soit donné par un objet comparateur.

opération	`set`	`map`
Insertion	$O(\log(n))$	$O(\log(n))$
Suppression (clé)	$O(\log(n))$	$O(\log(n))$
Suppression (itérateur)	$O(1)$	$O(1)$
Recherche	$O(\log(n))$	$O(\log(n))$
Parcours (temps total)	$O(n)$	$O(n)$

Les `multiset` et `multimap` permettent de gérer des ensembles ou tableaux associatifs avec plusieurs fois la même clé. Ils sont beaucoup moins utilisés.

Le code suivant transforme une séquence d'élements (avec répétitions) en un ensemble d'éléments (sans répétition). Ces conteneurs ordonne les éléments. Un parcours du conteneur donne donc les éléments dans l'ordre.

```cpp
#include <iostream>
#include <set>
```

```cpp
using namespace std;

int main()
{
  int t[] = { 5, 2, 7, 12, 9, 7, 4, 3, 2, 15, 4, 6 };
  set<int> s; // prend le < des int
  for ( int i = 0; i < 12; ++i ) s.insert( t[ i ] );
  for ( set<int>::const_iterator it = s.begin(), itE = s.end();
        it != itE; ++it )
    cout << " " << *it;
  cout << endl;
  return 0;
}
```

Ce programme affichera les éléments distincts réordonnés.

```
 2 3 4 5 6 7 9 12 15
```

Dans le tableau associatif `map`, l'opérateur [] est surchargé. Cela permet de se servir d'un tableau associatif comme d'un tableau, sauf que l'indice n'est pas forcément un entier. Le code suivant crée un annuaire téléphonique

```cpp
#include <cassert>
#include <iostream>
#include <string>
#include <map>

using namespace std;

int main()
{
  typedef map< string, string > Annuaire;
  Annuaire annuaire;
  annuaire[ "Lebaudu" ] =    "0479624545";
  annuaire[ "Grospalais" ] = "0479633678";
  annuaire[ "Encornet" ] =   "0479676543";
  annuaire[ "Estraminot" ] = "0475678765";
  annuaire[ "Youplein" ] =   "0479898312";
  annuaire[ "Amovibel" ] =   "0479812112";
  annuaire[ "Marmiton" ] =   "0479767842";
  string num = annuaire[ "Youplein" ]; // num = "0479898312"
  Annuaire::iterator it = annuaire.find( "Grospalais" );
  assert( it != annuaire.end() ); // vrai
  // Affiche tous les noms et numeros à partir de Grospalais.
  for ( Annuaire::iterator itE = annuaire.end();
        it != itE; ++it )
    std::cout << it->first << " : " << it->second << std::endl;
  return 0;
}
```

Ce programme affiche tous les noms et numéros à partir de "Grospalais".

```
Grospalais : 0479633678
Lebaudu : 0479624545
Marmiton : 0479767842
Youplein : 0479898312
```

5.4 Itérateurs

Tous les conteneurs précités et même un certain nombre de quasi-conteneurs (`string`, `valarray`, `bitset`, etc) fournissent des itérateurs pour parcourir et modifier les données. Ils n'ont en revanche pas tous les mêmes propriétés et n'offrent pas tous les mêmes services. En un sens, les itérateurs forment une hiérarchie (au sens sémantique), mais ne sont pas reliés physiquement par des relations d'héritage. Voilà les principales catégories d'itérateurs :

InputIterator Cet itérateur est `CopyConstructible`, `Assignable`, `Destructible`. En plus, il peut être incrémenté (`operator++`), et peut être comparé (`operator==` et `operator!=`). Enfin, il peut être déréférencé (`operator*`) comme une *rvalue*.

OutputIterator Pareil qu'un `InputIterator` sauf qu'il peut être déréférencé (`operator*`) comme une *lvalue*.

ForwardIterator Fusionne `InputIterator` et `OutputIterator` et rajoute que l'on peut accéder plusieurs fois au même élément par un déréférencement. Imaginez le cas d'un itérateur en lecture sur un flux audio/vidéo : une fois lu, la valeur est perdu si l'itérateur est passé à la suite. Dans ce cas, l'itérateur est un `InputIterator` et non un `ForwardIterator`.

BidirectionalIterator Ajoute au `ForwardIterator` les opérations de décrémentation (`operator-`).

RandomAccessIterator Ajoute au `BidirectionalIterator` toutes les opérations arithmétiques pour avancer ou reculer de plusieurs éléments d'un seul coup, les comparateurs inférieurs, supérieurs, et l'opérateur de déréférencement à la n-ème case (`operator[]`).

On note que :

list, set, map fournissent des `BidirectionalIterator`s. Il y a une version lecture/écriture appelée `iterator` et une version lecture seulement appelée `const_iterator`.

vector fournit des `RandomAccessIterator`s. Il y a une version lecture/écriture appelée `iterator` et une version lecture seulement appelée `const_iterator`.

Si l'on n'a pas besoin de modifier le conteneur, il est préférable d'utiliser les itérateurs `const_iterator`, qui empêche la modification des données. D'une part, c'est une sécurité. D'autre part, pour certaines structures de données, les itérateurs non-modifiables sont plus rapides.

NB : Le découpage `InputIterator`, `ForwardIterator`, etc, est en fait un peu plus fin que ça, car celui-ci est insuffisant dans certains cas (voir ce découpage classique `http://www.cplusplus.com/reference/iterator` et le nouveau découpage proposé dans BOOST `http://www.boost.org/doc/libs/1_55_0/libs/iterator/doc/index.html`).

5.5 Algorithmes

On donne ci-dessous tous les algorithmes définis dans le header `<algorithm>` ainsi que leur sens général. On se reportera à la documentation de référence pour obtenir les paramètres précis. On note que tous ces algorithmes sont des *patrons de fonction*, et que les intervalles (*range* en anglais) sont spécifiés par des *itérateurs*. Cette liste a été générée à partir du site `www.cplusplus.com`. On note qu'il existe un algorithme déjà écrit (testé, validé, optimal) pour quasiment toutes les requêtes usuelles sur les collections d'éléments. En un sens, il faudrait les connaître par cœur, car rien ne justifie de ne pas s'en servir.

Non-modifying sequence operations :

all_of	Test condition on all elements in range
any_of	Test if any element in range fulfills condition
none_of	Test if no elements fulfill condition
for_each	Apply function to range
find	Find value in range
find_if	Find element in range
find_if_not	Find element in range (negative condition)
find_end	Find last subsequence in range
find_first_of	Find element from set in range
adjacent_find	Find equal adjacent elements in range
count	Count appearances of value in range
count_if	Return number of elements in range satisfying condition
mismatch	Return first position where two ranges differ
equal	Test whether the elements in two ranges are equal
is_permutation	Test whether range is permutation of another
search	Search range for subsequence
search_n	Search range for elements

Modifying sequence operations :

copy	Copy range of elements
copy_n	Copy elements
copy_if	Copy certain elements of range
copy_backward	Copy range of elements backward
move	Move range of elements
move_backward	Move range of elements backward
swap	Exchange values of two objects
swap_ranges	Exchange values of two ranges
iter_swap	Exchange values of objects pointed by two iterators
transform	Transform range
replace	Replace value in range
replace_if	Replace values in range
replace_copy	Copy range replacing value
replace_copy_if	Copy range replacing value
fill	Fill range with value
fill_n	Fill sequence with value
generate	Generate values for range with function
generate_n	Generate values for sequence with function
remove	Remove value from range
remove_if	Remove elements from range
remove_copy	Copy range removing value
remove_copy_if	Copy range removing values
unique	Remove consecutive duplicates in range
unique_copy	Copy range removing duplicates
reverse	Reverse range
reverse_copy	Copy range reversed
rotate	Rotate left the elements in range
rotate_copy	Copy range rotated left
random_shuffle	Randomly rearrange elements in range
shuffle	Randomly rearrange elements in range using generator

Partitions :

is_partitioned	Test whether range is partitioned
partition	Partition range in two
stable_partition	Partition range in two - stable ordering
partition_copy	Partition range into two
partition_point	Get partition point

Sorting :

sort	Sort elements in range
stable_sort	Sort elements preserving order of equivalents
partial_sort	Partially sort elements in range
partial_sort_copy	Copy and partially sort range
is_sorted	Check whether range is sorted
is_sorted_until	Find first unsorted element in range
nth_element	Sort element in range

Binary search (operating on partitioned/sorted ranges) :

lower_bound	Return iterator to lower bound
upper_bound	Return iterator to upper bound
equal_range	Get subrange of equal elements
binary_search	Test if value exists in sorted sequence

Merge (operating on sorted ranges) :

merge	Merge sorted ranges
inplace_merge	Merge consecutive sorted ranges
includes	Test whether sorted range includes another sorted range
set_union	Union of two sorted ranges
set_intersection	Intersection of two sorted ranges
set_difference	Difference of two sorted ranges
set_symmetric_difference	Symmetric difference of two sorted ranges

Heap :

push_heap	Push element into heap range
pop_heap	Pop element from heap range
make_heap	Make heap from range
sort_heap	Sort elements of heap
is_heap	Test if range is heap
is_heap_until	Find first element not in heap order

Min/max :

min	Return the smallest
max	Return the largest
minmax	Return smallest and largest elements
min_element	Return smallest element in range
max_element	Return largest element in range
minmax_element	Return smallest and largest elements in range

Other :

lexicographical_compare	Lexicographical less-than comparison
next_permutation	Transform range to next permutation
prev_permutation	Transform range to previous permutation

Le code ci-dessous montre comment éliminer les duplicata dans un conteneur `vector`. Les éléments restants sont triés mais tous distincts.

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main()
{
  int t[] = { 5, 2, 7, 12, 9, 7, 4, 3, 2, 15, 4, 6 };
  vector<int> s;
  for ( int i = 0; i < 12; ++i ) s.push_back( t[ i ] );
  std::sort( s.begin(), s.end() );                                    // sort
  vector<int>::const_iterator itLast = std::unique( s.begin(), s.end() ); // unique
  for ( vector<int>::const_iterator it = s.begin(); it != itLast; ++it )
    cout << " " << *it;
  cout << endl;
  return 0;
}
```

```
 2 3 4 5 6 7 9 12 15
```

Le code suivant ne garde que les noms entre les lettres "D" (inclus) et "L" de l'annuaire.

```cpp
// annuaire-2.cpp. Attention C++11 ! Compilez avec g++ -std=c++0x
#include <iostream>
#include <string>
#include <map>
#include <vector>
#include <algorithm>

using namespace std;

// Predicat: retourne vrai seulement si la clé str est entre deux mots.
struct AnnuairePredicate {
  AnnuairePredicate( const std::string & min, const std::string & max )
```

```
    : m_min( min ), m_max( max ) {}
  bool operator()( const std::pair< std::string, std::string > & val ) const
  {
    return ( m_min <= val.first ) && ( val.first < m_max );
  }
  std::string m_min;
  std::string m_max;
};


int main()
{
  // Tableau associatif string -> string
  typedef map< string, string > Annuaire;
  // chaque élément d'un annuaire est une paire (clé,valeur), ici (string,string)
  typedef std::pair< std::string, std::string > Valeur;
  // Autre conteneur pour montrer qu'on passe sans problème d'un
  // conteneur associatif à un conteneur séquentiel
  typedef std::vector< Valeur > Sequence;

  // Crée l'annuaire
  Annuaire annuaire;
  annuaire[ "Lebaudu" ] =    "0479624545";
  annuaire[ "Grospalais" ] = "0479633678";
  annuaire[ "Encornet" ] =   "0479676543";
  annuaire[ "Estraminot" ] = "0475678765";
  annuaire[ "Youplein" ] =   "0479898312";
  annuaire[ "Amovibel" ] =   "0479812112";
  annuaire[ "Marmiton" ] =   "0479767842";

  // Extrait une partie.
  Sequence names_d_l;
  AnnuairePredicate pred( "D", "L" ); // entre D inclus et L exclus.
  std::copy_if( annuaire.begin(), annuaire.end(),
                // insert automatiquement à la fin de names_d_l
                std::back_inserter< Sequence >( names_d_l ),
                pred );

  // Affiche tous les noms et numeros sélectionnés.
  for ( Sequence::const_iterator it = names_d_l.begin(), itE = names_d_l.end();
        it != itE; ++it )
    std::cout << it->first << " : " << it->second << std::endl;
  return 0;
}
```

```
Encornet : 0479676543
Estraminot : 0475678765
Grospalais : 0479633678
```

5.6 Les mots-clé `auto` et `decltype`

Une des grandes difficultés de C++ est de forcer le programmeur à connaître le type exacte des expressions s'il veut les stocker (dans des variables par exemple). Or, c'est parfois très verbeux, cf l'exemple ci-dessous :

```
int main() {
  std::map< string, string > annuaire;
  annuaire[ "Jean Jobard" ]      = "0632176565";
  annuaire[ "Christiano Space" ] = "0698436513";
  annuaire[ "Helmut Weird" ]     = "0695987223";
  for ( std::map< string, string >::const_iterator it = annuaire.begin(),
          itE = annuaire.end(); it != itE; ++it )
    std::cout << it->first << " " << it->second << std::endl;
}
```

Une façon est de faire des typedefs :

```
int main() {
  typedef std::map< string, string > Annuaire;
  typedef Annuaire::const_iterator   AnnuaireCIterator;
  Annuaire annuaire;
```

```
  ...
  for ( AnnuaireCIterator it = annuaire.begin(), itE = annuaire.end();
        it != itE; ++it )
    std::cout << it->first << " " << it->second << std::endl;
}
```

Sinon, depuis C++11, le mot-clé `auto` permet de laisser le compilateur déduire automatique le type. Notez que s'il y a ambiguïté, il bloquera et vous le dira.

```
int main() {
  std::map< string, string > annuaire;
  ...
  for ( auto it = annuaire.begin(), itE = annuaire.end(); it != itE; ++it )
    std::cout << it->first << " " << it->second << std::endl;
}
```

Cela simplifie considérablement l'écriture de code en C++.

Le mot-clé **decltype** est utilisé pour récupérer le type d'une expression, comme par exemple la valeur de retour d'une fonction. C'est un outil pratique dans certaines situations. L'exemple ci-dessous montre comment récupérer le type pointé par un itérateur. std::declval<T>() permet quant à lui de déclarer une référence à une variable de type T, sans l'instancier vraiment. Voilà ce que ça donne (pour le fun) :

```
#include <iostream>
#include <vector>
#include <type_traits>

// nécessite c++20
// Calcul le maximum d'un tableau
// On doit combiner declval et decltype pour fabriquer le type de retour.
template <typename Iterator>
typename std::remove_reference< decltype( *std::declval<Iterator>() ) >::type
maximum( Iterator b, Iterator e )
{
  typedef typename std::remove_reference< decltype(*b) >::type Value;
  Value v;
  if ( b == e ) return v;
  v = *b++;
  for ( ; b != e ; ++b )
    if ( v < *b ) v = *b;
  return v;
}

int main()
{
  std::vector<int> V = { 4, 3, 18, 12, 5, 2, 13 };
  std::cout << maximum( V.begin(), V.end() ) << std::endl;
  return 0;
}
```

On voit que c'est peu lisible, mais ça marche ! Ici il aurait mieux valu utiliser les types traits des itérateurs :

```
#include <iostream>
#include <vector>
#include <type_traits>

// nécessite c++11
// Calcul le maximum d'un tableau
// Dans ce cas, c'est plus facile  de passer par les traits des itérateurs, qui
    sont capables de déterminer le type des valeurs pointées.
template <typename Iterator>
typename std::iterator_traits<Iterator>::value_type
maximum( Iterator b, Iterator e )
{
  typename std::iterator_traits<Iterator>::value_type v;
  if ( b == e ) return v;
  v = *b++;
  for ( ; b != e ; ++b )
    if ( v < *b ) v = *b;
  return v;
}
```

```cpp
int main()
{
  std::vector<int> V = { 4, 3, 18, 12, 5, 2, 13 };
  std::cout << maximum( V.begin(), V.end() ) << std::endl;
  return 0;
}
```

6 Ecrire générique : concepts, instantiation, spécialisation

6.1 Programmation générique

Une grande partie de la programmation générique se base sur l'écriture :
— d'*algorithmes génériques* grâce à l'utilisation de patrons de fonctions
— de *conteneurs génériques* grâce à l'utilisation de patrons de classes
— les types paramétrés des patrons définissent des *concepts* que doivent satisfaire les types donnés à l'instanciation de la fonction ou de la classe patron.
— un type concret qui satisfait un *concept* s'appelle un *modèle du concept*.

Tout algorithme ou classe générique doit donc identifier correctement les concepts qu'il impose. Depuis C++20, les concepts sont intégrés dans le langage, ce qui évite la lourdeur des concepts "a la mano" écrits en BOOST ou via SFINAE. Nous nous focaliserons ici sur l'usage des concepts depuis C++20.

L'écriture générique se base sur l'identification correcte des types manipulés dès l'étape de compilation. On verra donc comment déterminer les types, et se servir de ces catégorisations pour développer des codes qui prennent en compte ces caractéristiques.

NB : Vous devez vérifier que votre compilateur supporte bien C++20, avec l'option `-std=c++20`.

6.2 Identification des types

Le C++ fournit une bibliothèque de méta-programmation (#include <type_traits>) qui fournit plein d'interfaces génériques pour récupérer des informations sur les types. En général ils définissent une constante value qui vaut **true** ou **false** à la compilation. Voilà un exemple d'utilisation :

```cpp
bool a = std::is_integral<int>::value; // true
bool b = std::is_integral<char>::value; // true
bool c = std::is_integral<float>::value; // false
bool d = std::is_floating_point<double>::value; // true
struct A {};
bool e = std::is_integral<float>::value; // false
bool f = std::is_class<A>::value; // true
```

Le C++ définit une cinquantaine de tests différents des types :
— des tests en lien avec les types de base : is_void, is_null_pointer, is_integral, is_floating_point, is_array, is_enum, is_union, is_class, is_function, is_pointer, is_lvalue_reference, is_rvalue_reference, is_member_object_pointer, is_member_function_pointer.
— des tests pour identifier les types composites : is_fundamental, is_arithmetic, is_scalar, is_object, is_compound, is_reference, is_member_pointer
— des tests de propriétés des types : is_const, is_volatile, is_trivial, is_trivially_copyable, is_standard_layout, is_empty, is_polymorphic, is_abstract, is_final, is_aggregate, is_implicit_lifetime, is_signed, is_unsigned. On note ici que l'on peut tester si une classe est abstraite ou polymorphe ou si un type est non modifiable.
— des tests pour savoir si des opérations usuelles (construction, destruction, copie, assignation, etc) sont possibles, e.g. is_constructible, is_default_constructible, is_copy_constructible, is_assignable, is_trivially_assignable, is_copy_assignable, is_move_assignable, is_destructible, has_virtual_destructor, is_swappable_with, is_swappable
— des relations entre types : is_same, is_base_of, is_convertible, is_invocable. On note ce dernier qui permet de tester si c'est une fonction appelable avec certains paramètres.

Cette bibliothèque fournit aussi des "fonctions" de transformation des types (à la compilation).

— changements de const et volatile : remove_cv, remove_const, remove_volatile, add_cv, add_const, add_volatile
— changements de références : remove_reference, add_lvalue_reference, add_rvalue_reference
— changements de signes : make_signed, make_unsigned
— changements de tableaux : remove_extent, remove_all_extents
— changements de pointeurs : remove_pointer, add_pointer
— transformation de types : decay, remove_cvref, enable_if, conditional, common_type, common_reference, underlying_type, invoke_result, void_t, type_identity. Notamment enable_if is extrêmement utile pour définir des fonctions, classes ou méthodes qui n'existe que si une propriété est respectée.
— combinaisons logiques : conjunction, disjunction, negation.

The following code shows how to use some of these type analyzers to write compile-time conditional functions.

```cpp
#include <type_traits>
#include <iostream>

// c++17 minimum
template <typename T>
void affiche( const T&, std::string name )
{
  if constexpr( std::is_integral< T >::value ) {
    std::cout << "Type of " << name << " is integral." << std::endl;
    T b = 0x65;
    std::cout << "Value is " << b << std::endl;
  }
  if constexpr ( std::is_floating_point< T >::value ) {
    std::cout << "Type of " << name << " is floating point." << std::endl;
    T b = 3.14;
    std::cout << "Value is " << b << std::endl;
  }
  if ( std::is_class< T >::value )
    std::cout << "Type of " << name << " is a class." << std::endl;
  if ( std::is_union< T >::value )
    std::cout << "Type of " << name << " is a union." << std::endl;
  if ( std::is_function< T >::value )
    std::cout << "Type of " << name << " is a function." << std::endl;
  if ( std::is_pointer< T >::value )
    std::cout << "Type of " << name << " is a pointer." << std::endl;
}

#define SHOW( VAL ) affiche( VAL, #VAL )

struct A {};
union  B {};
void f() {}

int main( int argc, char* argv[] )
{
  SHOW( char(0) );
  SHOW( int(0) );
  SHOW( double(0.0) );
  SHOW( (void*)(0) );
  SHOW( f );
  SHOW( affiche<int> );
  SHOW( A() );
  SHOW( B() );
  return 0;
}
```

The result (compiled with `-std=c++17`) gives :

```
Type of char(0) is integral.
Value is e
Type of int(0) is integral.
Value is 101
Type of double(0.0) is floating point.
Value is 3.14
```

```
Type of (void*)(0) is a pointer.
Type of f is a function.
Type of affiche<int> is a function.
Type of A() is a class.
Type of B() is a union.
```

Notez le **if constexpr**(...) qui construit du code conditionnel à la compilation ! Si la condition, qui doit être évaluable à la compilation, n'est pas vérifiée, alors le code n'est pas générée. Cela permet donc de faire des sections de code conditionnées par le type de façon très facile. Le code précédent ne compile d'ailleurs pas si on remplace les **if constexpr**(...) par des **if** (...) , du fait que les affectations `T b =` ne fonctionnent pas pour tous les types.

6.3 Compilation conditionnelle ; SFINAE

On a vu la directive **if constexpr**(...) au paragraphe précédent, qui permet la compilation conditionnelle, mais qui n'est disponible que depuis C++17. Le principe général est le mécanisme SFINAE : "Substitution Failure Is Not An Error". Il faut se rappeler que parfois, le compilateur a plusieurs possibilités pour choisir la fonction, classe ou méthode, selon si elles sont génériques, spécialisées ou surchargées. L'objectif pour le compilateur est de prendre la version la plus spécifique (en un certain sens) mais si des variantes ne conviennent pas, ce n'est pas pour autant une erreur de compilation. C'est cela le mécanisme SFINAE, qui permet de choisir entre des implémentations de fonctions, classes ou méthodes grâce à ces tentatives de trouver la version la plus spécifique.

Voilà ci-dessous un exemple de SFINAE "à l'ancienne", pour détecter si une classe a une méthode `swap`. En général, il est beaucoup plus rapide de passer par cette méthode (pensez à l'échange de 2 vecteurs d'1 million d'éléments, il suffit en gros d'échanger les pointeurs), qu'en passant par un objet temporaire et en réaffectant les objets.

```cpp
// Before C++11, how to detect has_swap method

// SFINAE test. Member value is true only the first 'test' method can
// be called. It is defined solely when swap method exists.
template <typename T>
class has_swap
{
    typedef char one;
    typedef long two;

    template <typename C> static one test( decltype(&C::swap) ) ;
    template <typename C> static two test(...);

public:
    enum { value = sizeof(test<T>(0)) == sizeof(char) };
};

// Generic class for making a swap with copy and assignments. The
// second parameter is used for template class specialization (when
// 'true', there is a specialized class).
template <typename T, bool B = has_swap<T>::value >
struct SwapImpl {
  static void make( T& t1, T& t2 ) {
    std::cout << "SwapImpl<T>::do: generic version" << std::endl;
    T tmp = t1;
    t1 = t2;
    t2 = tmp;
  }
};

// This specialization is called only if has_swap<T> is true.
template <typename T>
struct SwapImpl<T,true> {
  static void make( T& t1, T& t2 ) {
    std::cout << "SwapImpl<T>::do: specialized swap version" << std::endl;
    t1.swap( t2 );
  }
};
```

```cpp
// This template function just call the correct swap method within
// SwapImpl<T,bool> or its specialization SwapImpl<T,true>.
template <typename T>
void myswap( T& t1, T& t2 ) {
  SwapImpl<T>::make( t1, t2 );
}
```

Voilà une deuxième version qui utilise std::enable_if et **decltype** (retourne le type d'une expression). std::enable_if permet d'activer ou non un type dans la définition d'une fonction, d'une classe ou d'une méthode. Or si vous utilisez un type non activé dans la définition de cette entité, elle n'est pas compilée.

```cpp
// Shorter SFINAE approach C++11 but not so readable readable according to me.

// Generic swap template function
template <typename T>
void
makeSwap( T& t1, T& t2, long long ) {
  std::cout << "SwapImpl<T>::do: generic version" << std::endl;
  T tmp = t1;
  t1 = t2;
  t2 = tmp;
}

// Specialized swap template function enabled only when SFINAE test
// succeeds.
template <typename T>
typename std::enable_if< true, decltype(&T::swap) >::type
makeSwap( T& t1, T& t2, int ) {
  std::cout << "SwapImpl<T>::do: specialized swap version" << std::endl;
  t1.swap( t2 );
  return 0; // unused, avoid warning
}

// Swap methods. Tries to call specialized swap (0 is an integer so
// makeSwap( T&,T&,int ) is preferred over makeSwap( T&,T&,long long
// ), but is only available if T::swap exists.
template <typename T>
void myswap( T& t1, T& t2 ) {
  makeSwap( t1, t2, 0 );
}
```

Voilà enfin un exemple d'utilisation, qui permet de choisir à la compilation la fonction la plus rapide si le type fournit bien une méthode swap.

```cpp
// Displays a vector
template <typename T, typename Alloc = std::allocator<T> >
std::ostream& operator<<( std::ostream& out, const std::vector<T,Alloc>& V )
{
  for ( auto e : V ) out << e << " ";
  return out;
}

int main( int argc, char* argv[] )
{
  int i = 5, j = 10;
  std::cout << "i=" << i << " j=" << j << std::endl;
  myswap( i, j );
  std::cout << "i=" << i << " j=" << j << std::endl;
  std::vector<int> v1 = { 17, 4, 2 };
  std::vector<int> v2 = { 12, 25, 14, 4, 6 };
  std::cout << "v1=" << v1 << " v2=" << v2 << std::endl;
  myswap( v1, v2 );
  std::cout << "v1=" << v1 << " v2=" << v2 << std::endl;
  return 0;
}
```

Le programme affiche :

```
i=5 j=10
SwapImpl<T>::do: generic version
i=10 j=5
```

```
v1=17 4 2  v2=12 25 14 4 6
SwapImpl<T>::do: specialized swap version
v1=12 25 14 4 6  v2=17 4 2
```

Nous montrons ci-dessous un autre exemple de compilation conditionnelle d'abord sans **if constexpr** , puis avec. On utilise std::is_invocable<T, P1, P2, ...>::value qui est vrai si un objet t de type T peut être invoqué avec les paramètres de type P1, P2, ... , i.e. l'expression t(p1,p2 ,...) est valide si les paramètres sont du bon type. Ce code tente d'invoquer une fonction/objet fonction qui prend un **double**& en paramètre à tous éléments d'une collection spécifiée par un *range* d'itérateurs. Si ce n'est pas possible, elle se plaint !

```
#include <cstdlib>
#include <type_traits>
#include <iostream>

// c++17 minimum for std::is_invocable

// This function is called if f(double&) exists
template < typename Iterator,
           typename T,
           typename std::enable_if< std::is_invocable< T, double& >::value, bool
               >::type
           = true >
void apply( Iterator b, Iterator e, T& f )
{
  for ( ; b != e; ++b ) f( *b );
}

// This function is called if f(double&) does not exist
template < typename Iterator,
           typename T,
           typename std::enable_if<!std::is_invocable< T, double& >::value, bool
               >::type
           = true >
void apply( Iterator b, Iterator e, T& f )
{
  std::cout << "Not invocable as f(double&)" << std::endl;
}

double rand01() { return double( rand() ) / double( RAND_MAX ); }
void   display( double& x ) { std::cout << " " << x; }
void   reset  ( double& x ) { x = 0.0; }
void   random1( double& x ) { x = rand01(); }
void   incr   ( double& x ) { x += 1; }
void   dummy  () {}

int main()
{
  double t[ 5 ];
  apply( t, t+5, display ); std::cout << std::endl;
  apply( t, t+5, reset );
  apply( t+1, t+3, random1 );
  apply( t, t+5, incr );
  apply( t, t+5, display ); std::cout << std::endl;
  apply( t, t+5, dummy );   std::cout << std::endl;
  return 0;
}
```

Ce programme affiche :

```
 3.04553e-314 3.28013e-314 3.04553e-314 2.12583e-314 2.12582e-314
 1 1.00001 1.13154 1 1
Not invocable as f(double&)
```

On voit comment std::enable_if teste une propriété sur un type et active ou non son type interne type en fonction. La fonction générique n'est activée que si ce 3ème paramètre template existe.

On voit ci-dessous comment **if constexpr** simplifie le code dans ce cas, et le rend bien plus lisible.

```
// c++17 minimum for std::is_invocable and if constexpr
```

```cpp
template < typename Iterator,
           typename T >
void apply( Iterator b, Iterator e, T& f )
{
  // Test if f( double& ) exists
  if constexpr( std::is_invocable< T, double& >::value )
    for ( ; b != e; ++b ) f( *b );
  else
    std::cout << "Not invocable as f(double&)" << std::endl;
}
```

On verra comment les concepts de C++20 simplifient souvent le développement de codes spécifiques à des types (du point de vue du développeur, l'utilisateur de la bibliothèque ne voyant jamais les détails de ces mécanismes SFINAE).

6.4 Définition et utilisation des concepts

C++20 définit maintenant le mot-clé **concept** pour définir des concepts s'appliquant à un ou plusieurs types paramètrés. Attention ce ne sont pas des classes et il ne sont pas instanciables, mais on peut les voir (en un sens) comme des super-classes (abstraites) des modèles qui les satisfont.

Voilà deux exemples d'écriture de concepts.

```cpp
// Vérifie que l'on peut instancier par copie le type T
template <typename T>
concept CopyConstructible = requires(T a) {
  { T( a ) } -> std::same_as<T>;
};
// Vérifie la présence de operator= et du type de retour.
template <typename T>
concept Assignable = requires(T a, T b) {
  { a = b } -> std::same_as<T&>;
};
```

On voit que l'on définit les contraintes syntaxiques (*requirements*) que doivent respecter le(s) type(s) en écrivant des petits bouts de code. Une autre solution est d'utiliser la bibliothèque de méta-programmation de C++. Par exemple, on pourrait définir et utiliser le concept de nombre ainsi :

```cpp
#include <type_traits>
#include <iostream>

template <typename T>
concept Number
= std::is_integral<T>::value
  || std::is_floating_point<T>::value;

template <Number N>
N add( N a, N b ) { return a+b; }

int main()
{
  int a = 5, b = 8;
  std::cout << a << " + " << b << " = " << add( a, b ) << std::endl;
  double x = -1.5, y = 3.7;
  std::cout << x << " + " << y << " = " << add( x, y ) << std::endl;
  std::string s = "Yop", t = "Jup";
  std::cout << s << " + " << t << " = " << add( s, t ) //< ne compile pas
            << std::endl;
  return 0;
}
```

On voit qu'on peut utiliser un nom de concept en lieu et place de **typename** dans la déclaration d'une fonction ou classe générique. En fait, on peut imposer aux types paramétrés de satisfaire des concepts de plusieurs autres façons (presque) équivalentes :

```cpp
template <typename T>
requires CopyConstructible<T> && Assignable<T>
void myswap( T & t1, T & t2 )
{
  T tmp( t1 ); // copy construction
  t1 = t2;     // assignment
```

```
  t2 = tmp;     // assignment
}
```

Le mot-clé **requires** peut être mis après.

```
template <typename T>
void myswap( T & t1, T & t2 )
  requires CopyConstructible<T> && Assignable<T>
{
  T tmp( t1 ); // copy construction
  t1 = t2;     // assignment
  t2 = tmp;    // assignment
}
```

On peut aussi grouper des concepts ensembles (conjonction avec &&, disjonction avec ||), et utiliser le nom du concept comme un type. A ce moment, il faut rajouter le mot clé **auto** derrière.

```
template <typename T>
concept Duplicable = CopyConstructible<T> && Assignable<T>;

void myswap( Duplicable auto & t1, Duplicable auto & t2 )
{
  Duplicable auto tmp( t1 ); // copy construction
  t1 = t2;     // assignment
  t2 = tmp;    // assignment
}
```

Le code suivant permet de vérifier que les concepts doivent être validés pour que le programme compile.

```
struct A {};
struct B { B() = default; B(const B&) = delete; };
struct C { C& operator=(const C&) = delete; };

int main()
{
  std::string s1 = "Bonjour";
  std::string s2 = "Toto";
  myswap( s1, s2 );     //< ok une string peut être dupliquée
  std::cout << s1 << " " << s2 << std::endl;
  A a1, a2;
  myswap( a1, a2 );     //< ok A peut être dupliqué par défaut
  // B b1, b2;
  // myswap( b1, b2 );  //< ne compile pas, pas de copie
  // C c1, c2;
  // myswap( c1, c2 );  //< ne compile pas, pas d'affectation
}
```

Il s'exécute ainsi :

```
Toto Bonjour
```

On note par exemple que la compilation de myswap(b1, b2); échoue et renvoie les erreurs suivantes, assez lisibles :

```
concepts-1.cpp:38:3: error: no matching function for call to 'myswap'
  myswap( b1, b2 );  //< ne compile pas
  ^~~~~~
concepts-1.cpp:18:6: note: candidate template ignored: constraints not satisfied
[with T = B]
void myswap( T & t1, T & t2 )
     ^
concepts-1.cpp:17:10: note: because 'B' does not satisfy 'CopyConstructible'
requires CopyConstructible<T> && Assignable<T>
         ^
concepts-1.cpp:6:5: note: because 'T(a)' would be invalid: functional-style cast from
'B' to 'B' uses deleted function
  { T( a ) } -> std::same_as<T>;
    ^
1 error generated.
```

On reprend maintenant l'exemple de la détection de la méthode swap dans une classe, que l'on avait modélisé à l'aide du paradigme SFINAE puis de la compilation conditionnelle. Voilà maintenant l'écriture C++20 avec les concepts de ce problème.

```
// C++20 has_swap method

// Declaration of the concept "has_swap", which is satisfied by any type 'T'
// such that for values 'a' of type 'T', the expression 'a.swap( a )' compiles.
template<typename T>
concept has_swap = requires(T a)
{
  a.swap( a );
};

template <typename T>
void myswap( T& t1, T& t2 ) {
  std::cout << "myswap: copy t1 into t2 with tmp" << std::endl;
  T t = t1;
  t1  = t2;
  t2  = t;
}

// This template function expect a type T satisfying concept has_swap,
// just call the correct swap method.
template <has_swap T>
void myswap( T& t1, T& t2 ) {
  std::cout << "myswap: t1.swap( t2 )" << std::endl;
  t1.swap( t2 );
}
```

On voit qu'on peut utiliser un nom de concept en lieu et place de **typename** dans la déclaration d'une fonction ou classe générique.

6.5 Concepts classiques

Le C++ fournit dans sa bibliothèque standard tout un ensemble de concepts pré-écrits. Pour les concepts usuels, la plupart sont dans le header <concepts> :

Concepts essentiels du langage

same_as	specifies that a type is the same as another type
derived_from	specifies that a type is derived from another type
convertible_to	specifies that a type is implicitly convertible to another type
common_reference_with	specifies that two types share a common reference type
common_with	specifies that two types share a common type
integral	specifies that a type is an integral type
signed_integral	specifies that a type is an integral type that is signed
unsigned_integral	specifies that a type is an integral type that is unsigned
floating_point	specifies that a type is a floating-point type
assignable_from	specifies that a type is assignable from another type
swappable, swappable_with	specifies that a type can be swapped or that two types can be swapped with each other
destructible	specifies that an object of the type can be destroyed
constructible_from	specifies that a variable of the type can be constructed from or bound to a set of argument types
default_initializable	specifies that an object of a type can be default constructed
move_constructible	specifies that an object of a type can be move constructed
copy_constructible	specifies that an object of a type can be copy constructed and move constructed

Concepts liés aux comparaisons

boolean−testable	specifies that a type can be used in Boolean contexts (exposition-only concept*)
equality_comparable, equality_comparable_with	specifies that operator == is an equivalence relation
totally_ordered, totally_ordered_with	specifies that the comparison operators on the type yield a total order
three_way_comparable , three_way_comparable_with	specifies that operator <=> produces consistent result on given types, in header <compare>

Concepts liés aux objets

movable	specifies that an object of a type can be moved and swapped
copyable	specifies that an object of a type can be copied, moved, and swapped
semiregular	specifies that an object of a type can be copied, moved, swapped, and default constructed
regular	specifies that a type is regular, that is, it is both semiregular and equality_comparable

Concepts liés aux appels de fonctions (**operator**()(...))

invocable, regular_invocable	specifies that a callable type can be invoked with a given set of argument types
predicate	specifies that a callable type is a Boolean predicate
relation	specifies that a callable type is a binary relation
equivalence_relation	specifies that a relation imposes an equivalence relation
strict_weak_order	specifies that a relation imposes a strict weak ordering

De nombreux autres concepts sont définis dans la bibliothèque sur les itérateurs (header <iterator>), celle sur les algorithmes (header <algorithm>, celle sur les intervalles (header <ranges>). Nous n'allons pas les décrire ici, juste montrer quelques exemples.

6.6 Identification des abstractions

Il y a différents type d'abstractions dans un programme : des abstractions de l'ordre des données, des abstractions de l'ordre des algorithmes. Il faut savoir identifier les abstractions qui sont indépendantes les unes des autres, des abstractions dépendantes. Par exemple, lorsqu'on vous donnez un objet Comparateur à un algorithme de tri, la valeur associée au Comparateur doit être la même que la valeur pointée par les itérateurs.

6.7 Les types associés

On appelle types associés les types internes définis dans les concepts. Tout modèle d'un concept C doit définir les types associés au concept C, souvent avec une certaine sémantique derrière.

Par exemple, le concept d'itérateur (quelle que soit la variante) implique l'exemple de valeurs sur lequel point un itérateur. Il est alors assez naturel de demander à tout itérateur d'avoir un type interne value_type qui définit le type des données pointées.

On note néanmoins que le c++ tend à éliminer de plus en plus les types associés aux concepts, pour les remplacer par des classes traits (voir ci-dessous), ou alors laisser l'utilisateur utiliser **decltype**(*it) pour déduire le type.

Si It est un type d'itérateur, on utiliserait plutôt std :: iterator_traits <It>::value_type pour obtenir ce type, ou encore std :: itervalue_t<It> depuis C++20.

6.8 Les classes traits

Les classes traits permettent d'associer des propriétés, des valeurs, ou des types à d'autres types. On définit souvent des classes traits pour spécialiser certains comportement dans des algorithmes assez

complexes. Grâce à la spécialisation des patrons de classe, il est facile de définir un comportement par défaut puis de spécialiser les choses pour certains types. Par exemple, on peut spécifier comment passer les instances en paramètres par valeur ou par référence constante à l'aide de traits. L'idée est de définir une classe traits générique (ici `ParameterTraits`) et de la spécialiser dans les cas usuels.

```
#ifndef _PARAMETER_TRAITS_HPP_
#define _PARAMETER_TRAITS_HPP_

/// Définition générique du passage de paramètre non modifiable
template <typename T>
struct ParameterTraits {
  typedef const T & ConstParamType; // Par defaut, passage en référence constante.
};

// Spécialisation pour int
template <>
struct ParameterTraits<int> {
  typedef const int ConstParamType; // pour int, il vaut mieux passer par valeur
};

// Spécialisation pour double
template <>
struct ParameterTraits<double> {
  typedef const double ConstParamType; // pour double, il vaut mieux passer par
      valeur
};

#endif // _PARAMETER_TRAITS_HPP_
```

On peut ensuite s'en servir ainsi :

```
#include "ParameterTraits.hpp"

template <typename T>
struct Accumulateur {
  typedef typename ParameterTraits<T>::ConstParamType ConstT;
  T value;
  Accumulateur() : value( 0 ) {}
  void operator()( ConstT v ) // choisit le bon à l'instanciation.
  { value += v; }
};

struct B {
  typedef ParameterTraits<B>::ConstParamType ConstT;
  int tab[ 10 ];

  B( int v ) { tab[ 0 ] = v; }
  B& operator+=( ConstT other )
  {
    for ( int i = 0; i < 10; ++i )
      tab[ i ] += other.tab[ i ];
    return *this;
  }
};

int main()
{
  Accumulateur<int> acci;
  acci( 10 );  // utilise 'const int'
  Accumulateur<B> accb;
  accb( B( 4 ) ); // utilise 'const B &'
  return 0;
}
```

Un dernier exemple est la spécialisation du passage de paramètres en fonction de la taille du type de donnée. On utilise la spécialisation partielle pour choisir dès la compilation le bon type.

```
template <bool, typename T, typename F>
struct conditional {
  typedef T type; // par défaut retourne le premier type T
};
```

```cpp
template <typename T, typename F>  // partial specialization on first argument
struct conditional<false, T, F> {
  typedef F type; // spécialisation si false, retourne le deuxième type F
};

template <typename T>
struct ParamTraits {
  typedef typename std::conditional< sizeof( T ) <= 8, const T, const T & >::type
      ConstType;
};

template <typename T >
void f( typename ParamTraits< T >::ConstType value )
{}

struct A {
  double v;
  A() = default;
  A( const A& ) { std::cout << "A::A(const A&)" << std::endl; }
};
struct B {
  double v[4];
  B() = default;
  B( const B& ) { std::cout << "B::B(const B&)" << std::endl; }
};
struct C {
  int v;
  C() = default;
  C( const C& ) { std::cout << "C::C(const C&)" << std::endl; }
};

int main()
{
  A a;
  B b;
  C c;
  f<A>( a );
  f<B>( b );
  f<C>( c );
  return 0;
}
```

6.9 Hiérarchie de marques (tags), traits et spécialisation

Les modèles n'héritent pas de super-classe(s) même s'ils partagent des caractéristiques communes qui ressemblent fort à des hiérarchies de classes. En programmation générique, on simule les hiérarchies via des héritages entre marques (*tags*) associés aux modèles.

Ainsi un itérateur de vector est un random_access_iterator. En un sens, il est aussi un forward_iterator, etc. Mais il n'hérite pas d'une autre classe n'implémentant que la partie forward_iterator. En revanche, sa marque hérite bien de la marque d'un forward_iterator.

```cpp
namespace std {
  struct trivial_iterator_tag {}
  struct input_iterator_tag : trivial_iterator_tag {}
  struct output_iterator_tag : trivial_iterator_tag {}
  struct forward_iterator_tag : input_iterator_tag, output_iterator_tag {}
  struct bidirectional_iterator_tag : forward_iterator_tag {}
  struct random_access_iterator_tag : bidirectional_iterator_tag {}
}
```

On note qu'on retrouve la hiérarchie de concepts sous forme d'une hiérarchie d'héritage sur les marques. Le code suivant montre l'utilisation de la marque `random_access_iterator_tag` pour appeler le bon programme de tri en fonction du type d'itérateurs.

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

```cpp
template <typename Iterator, typename tag>
struct SortImpl {
  static void mySort( Iterator first, Iterator last )
  {
    std::cerr << "Unspecialized iterator: make a vector first." << std::endl;
    typedef typename std::iterator_traits<Iterator>::value_type Value;
    std::vector< Value > v;
    for ( Iterator it = first; it != last; ++it )
      v.push_back( *it );
    std::sort( v.begin(), v.end() );
    int i = 0;
    for ( Iterator it = first; it != last; ++it, ++i )
      *it = v[ i ];
  }

};

template <typename Iterator>
struct SortImpl<Iterator, std::random_access_iterator_tag> {
  static void mySort( Iterator first, Iterator last )
  {
    std::cerr << "Specialized iterator: use std::sort immediately." << std::endl;
    std::sort( first, last );
  }
};

template <typename Iterator>
void
mysort( Iterator first, Iterator last )
{
  typedef typename std::iterator_traits<Iterator>::iterator_category Tag;
  SortImpl< Iterator, Tag >::mySort( first, last );
}

int main()
{
  using namespace std;
  int t[] = { 5, 2, 7, 12, 9, 7, 4, 3, 2, 15, 4, 6 };
  list<int> l;
  vector<int> v;
  for ( int i = 0; i < 12; ++i ) {
    l.push_back( t[ i ] );
    v.push_back( t[ i ] );
  }

  //sort( v.begin(), v.end() ); // ok
  //sort( l.begin(), l.end() ); // fail
  mysort( v.begin(), v.end() ); // ok
  mysort( l.begin(), l.end() ); // ok
  cout << "vector: ";
  for ( int i = 0; i < 12; ++i ) cout << " " << v[ i ];
  cout << endl;
  cout << "list:   ";
  for ( list<int>::const_iterator it = l.begin(), itE = l.end();
        it != itE; ++it ) cout << " " << *it;
  cout << endl;
  return 0;
}
```

6.10 Spécialisation

Lorsque c'est pertinent : (1) surcharger les patrons de fonction, (2) spécialiser les patrons de classe. Attention, cela a tendance à dupliquer le code. On retombe sur le copier/coller usuel.

7 Bonus : méta-programmation, lambda, etc.

7.1 Méta-programmation

Le compilateur C++ peut compiler des expressions très complexes, et en réalité, effectuer des calculs comme un programme en utilisant les mécanismes liés aux classes patrons et à leur spécialisation. On parle de *méta-programmation*. On peut par exemple précalculer certaines expressions pour qu'à l'exécution le processeur n'est pas à les recalculer. Un exemple simple est donné ci-dessous, où on se débrouille pour faire calculer par le compilateur la factorielle de n'importe quel entier (pas trop grand).

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

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

int main()
{
  cout << "1! = " << Factorielle<1>::value << endl;
  cout << "2! = " << Factorielle<2>::value << endl;
  cout << "3! = " << Factorielle<3>::value << endl;
  cout << "4! = " << Factorielle<4>::value << endl;
  cout << "5! = " << Factorielle<5>::value << endl;
}
```

7.2 Expressions constantes `constexpr`

Le C++ plus récent (C++11 mais surtout C++14 et C++17) offre maintenant le mot-clé `constexpr`, qui désigne une expression pouvant être évaluée au moment de la compilation.

```
#include <iostream>

// C++11 constexpr functions use recursion rather than iteration
// (C++14 constexpr functions may use local variables and loops)
constexpr int factorial( int n)
{
    return n <= 1 ? 1 : (n * factorial(n - 1));
}

template <int N> struct Display {
  Display() { std::cout << N; }
};

int main()
{
  // compile-time
  std::cout << "10! = ";
  Display< factorial( 10 ) > d; // ok, compile time
  std::cout << std::endl;
  int n;
  std::cin >> n;
  // run-time
  std::cout << n << "! = ";
  // Display< factorial( n ) > dd; // ko
  // std::cout << std::endl;
  // std::cout << factorial( n ) << std::endl; // ok
  return 0;
}
```

Cela remplace avantageusement la méta-programmation "à la mano" précédente, qui devait passer par des classes annexes de calcul.

7.3 Programmation fonctionnelle par lambda expressions

Depuis C++11, le C++ s'est doté d'une syntaxe plus pratique pour faire de la programmation fonctionnelle. En C, on pouvait utiliser des pointeurs de fonctions pour simuler cette façon de programmer, mais c'était peu pratique lorsqu'il fallait des données externes (par exemple, faire un simple accumulateur nécessite de stocker une valeur). En C++ avant C++11, on crée des objets fonctions, par exemple :

```
struct Inferieur {
  int value;
  Inferieur( int v ) : value( v ) {}
  bool operator()( int a ) const { return a < v; }
};

int main() {
    std::vector<int> c { 12, 6, 9, 14, 15, 19, 5 };
    Inferieur inf10( 10 );
    // Repère les éléments inférieurs à 10 (et lesmet à la fin), puis les efface.
    c.erase( std::remove_if(c.begin(), c.end(), inf10 ),
             c.end());
}
```

Ca marche, mais c'est un peu long. Il faut bien spécifier constructeurs (et parfois destructeur), et le type de l'objet fonction n'est pas explicite. Il est caché dans l'opérateur.

Depuis C++11, on peut créer des fonctions (anonymes) en utilisant une syntaxe spéciale :

```
[ capture ] ( params ) mutable exception attribute -> ret { body } (1)
[ capture ] ( params ) -> ret { body }                              (2)
[ capture ] ( params ) { body }                                      (3)
[ capture ] { body }                                               (4)
```

1. Déclaration complète
2. Déclaration d'un lambda const : les objets capturés par copie ne peuvent pas être modifiés .
3. Type de retour omis : le type de retour de la fermeture operator() est déduite en regardant le type de retour de `return`.
4. Liste des paramètres omis : la fonction ne prend aucun argument, comme si la liste des paramètres est ()

Ainsi on peut écrire le code précédent ainsi :

```
int main() {
    std::vector<int> c { 12, 6, 9, 14, 15, 19, 5 };
    const int v = 10;
    // Repère les éléments inférieurs à 10 (et les met à la fin), puis les efface.
    auto last = std::remove_if( c.begin(), c.end(),
                                [v] (int n) { return n < v; } );
    c.erase( last, c.end() ); // les efface effectivement
}
```

Notez comment on capture une variable externe (avec []) pour pouvoir y accéder dans le corps du lambda.

Quelques exemples de captures :

`[a,&b]` où a est capturé par valeur et b est capturé par référence.
`[this]` capte le pointeur this
`[&]` capture tous les symboles par référence
`[=]` capture tous les symboles par valeur
`[]` ne capture rien.

On peut par exemple faire très facilement un tri indirect avec les lambda-expressions.

7.4 Type des lambdas et `std::function`

Parfois on veut déclarer des fonctions et s'en reservir ailleurs. Pour les lambda-termes, on peut utiliser le mot-clé `auto`, mais du coup on perd l'information de type. Une autre façon est d'utiliser `std::function`, qui a l'extrême bon goût de pouvoir stocker des objets fonction, des lambda-expressions, des fonctions C, du moment qu'elles sont du bon type !

```cpp
#include <cmath>
#include <functional>

typedef std::function< double ( double ) > Fonction; // fonction double -> double

struct Bar01 {
  double operator()( double x )
  { return ( x >= 0.0 && x <= 1.0 ) ? 1.0 : 0.0; }
};

Fonction f1 = sin; // marche, car sinus prend un double et retourne un double.

Fonction f2 = Bar01(); // marche, car objet fonction avec double operator()( double
    )

Fonction f3 = [] (double x) { return ( x >= 0.0 && x <= 1.0 ) ? 1.0 : 0.0; }; //
    marche car lambda
```

7.5 Le mécanisme SFINAE plus en détails

SFINAE est l'acronyme de "Substitution Failure Is Not An Error". Lors du mécanisme qui déduit les types des arguments des fonctions génériques (template function), le C++ essaie de générer toutes les signatures possibles des fonctions génériques surchargées. Si un argument de type invalide ou une valeur de retour de type invalide est généré, cette signature est écartée de la liste des signatures possibles mais ne génère pas d'erreur. Il n'y a erreur que si, à la fin de ce processus, plusieurs signatures possibles sont équivalentes ou aucune ne peut s'appliquer.

On peut utiliser ce mécanisme pour connaître les propriétés de certains types et spécialiser le comportement d'un programme dès l'étape de compilation. L'exemple ci-dessous détermine si un type est un pointeur :

```cpp
template <class T>
struct is_pointer
{
  template <class U>
  static char is_ptr(U *); // select simple pointers

  template <class U>
  static char is_ptr(U (*)()); // select pointers to functions

  template <class X, class Y>
  static char is_ptr(Y X::*); // select pointers to function members

  static double is_ptr(...); // select everyone else

  static T t;
  enum { value = sizeof(is_ptr(t)) == sizeof(char) };
};

struct Foo { int bar; };

int main(void)
{
  typedef int * IntPtr;
  typedef int Foo::* FooMemberPtr;
  typedef int (*FuncPtr)();

  printf("%d\n",is_pointer<IntPtr>::value);          // prints 1
  printf("%d\n",is_pointer<FooMemberPtr>::value);    // prints 1
  printf("%d\n",is_pointer<FuncPtr>::value);         // prints 1
  printf("%d\n",is_pointer<Foo>::value);             // prints 0
}
```

Remarquez que `is_ptr` est surchargé de plusieurs façons. Si jamais c'est un pointeur classique, la première surcharge est choisie. Si c'est un pointeur vers un membre, la deuxième surcharge est choisie. Si c'est un pointeur de fonction, la troisième surcharge est choisie. Dans ces 3 cas, le type de retour est un `char`. Sinon la quatrième surcharge est choisie et le type de retour est `double`, qui n'a pas

la même taille que `char`. Ensuite, vous pouvez utilisé dès l'étape de compilation la valeur booléenne `is_pointer<T>::value`. Cela permet généralement de faire des spécialisations de classe.

Montrons comment utiliser ce mécanisme pour détecter si un type `T` contient une méthode `swap` d'échange. On fera une fonction générique d'échange, que l'on spécialisera si le type a bien cette méthode `swap`. Notez par exemple qu'utiliser la méthode `swap` des conteneurs standard prend un temps constant, alors que faire l'échange à la main en passant par des affectations prend un temps proportionnel à la taille des données. Ce mécanisme SFINAE permet ainsi des gains très importants de temps de calcul.

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

// SFINAE test. Member value is true only the first 'test' method can
// be called. It is defined solely when swap method exists.
template <typename T>
class has_swap
{
    typedef char one;
    typedef long two;

    template <typename C> static one test( decltype(&C::swap) ) ;
    template <typename C> static two test(...);

public:
    enum { value = sizeof(test<T>(0)) == sizeof(char) };
};

// Generic class for making a swap with copy and assignments. The
// second parameter is used for template class specialization (when
// 'true', there is a specialized class).
template <typename T, bool B = has_swap<T>::value >
struct SwapImpl {
  static void make( T& t1, T& t2 ) {
    std::cout << "SwapImpl<T>::do: generic version" << std::endl;
    T tmp = t1;
    t1 = t2;
    t2 = tmp;
  }
};

// This specialization is called only if has_swap<T> is true.
template <typename T>
struct SwapImpl<T,true> {
  static void make( T& t1, T& t2 ) {
    std::cout << "SwapImpl<T>::do: specialized swap version" << std::endl;
    t1.swap( t2 );
  }
};

// This template function just call the correct swap method within
// SwapImpl<T,bool> or its specialization SwapImpl<T,true>.
template <typename T>
void myswap( T& t1, T& t2 ) {
  SwapImpl<T>::make( t1, t2 );
}

// Displays a vector
template <typename T, typename Alloc = std::allocator<T> >
std::ostream& operator<<( std::ostream& out, const std::vector<T,Alloc>& V )
{
  for ( auto e : V ) out << e << " ";
  return out;
}

struct InvalidSwapMember {
  // bad signature
  void swap();
};
```

```
int main( int argc, char* argv[] )
{
  std::cout << "has_swap<int>::value="
            << has_swap<int>::value << std::endl;                    // 0: ok
  std::cout << "has_swap<std::vector<int>>::value="
            << has_swap< std::vector<int> >::value << std::endl; // 1: ok
  int i = 5, j = 10;
  std::cout << "i=" << i << " j=" << j << std::endl;
  myswap( i, j );
  std::cout << "i=" << i << " j=" << j << std::endl;
  std::vector<int> v1 = { 17, 4, 2 };
  std::vector<int> v2 = { 12, 25, 14, 4, 6 };
  std::cout << "v1=" << v1 << " v2=" << v2 << std::endl;
  myswap( v1, v2 );
  std::cout << "v1=" << v1 << " v2=" << v2 << std::endl;

  std::cout << "has_swap<InvalidSwapMember>::value="
            << has_swap< InvalidSwapMember >::value << std::endl;// 1: bad !
  return 0;
}
```

Le dernier exemple `has_swap< InvalidSwapMember >::value` montre les limites de cette première approche. En effet, un utilisateur pourrait mettre une classe avec une méthode `swap` qui n'a pas la bonne signature `void swap(T&)`. Cela est dû au fait qu'on teste juste si la méthode existe avec un `decltype`, qui retourne un type si la méthode existe et qui sinon n'est pas compilable.

On peut en fait être beaucoup plus précis sur la définition de la méthode souhaitée. Il suffit de demander l'existence d'un pointeur de méthode avec les bons paramètres et valeur de retour.

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

// Checks the existence of a method 'void swap(T&)'
template<typename T>
class has_standard_swap
{
  // This line specifies the exact signature (second template argument)
  template <typename U, void (U::*)(T&)> struct Check;
  template <typename U> static char func(Check<U, &U::swap> *);
  template <typename U> static int  func(...);
public:
  typedef has_standard_swap type;
  enum { value = sizeof(func<T>( nullptr )) == sizeof(char) };
};

// Generic class for making a swap with copy and assignments. The
// second parameter is used for template class specialization (when
// 'true', there is a specialized class).
template <typename T, bool B = has_standard_swap<T>::value >
struct SwapImpl {
  static void make( T& t1, T& t2 ) {
    std::cout << "SwapImpl<T>::do: generic version" << std::endl;
    T tmp = t1;
    t1 = t2;
    t2 = tmp;
  }
};

// This specialization is called only if has_standard_swap<T> is true.
template <typename T>
struct SwapImpl<T,true> {
  static void make( T& t1, T& t2 ) {
    std::cout << "SwapImpl<T>::do: specialized swap version" << std::endl;
    t1.swap( t2 );
  }
};

// This template function just call the correct swap method within
// SwapImpl<T,bool> or its specialization SwapImpl<T,true>.
template <typename T>
void myswap( T& t1, T& t2 ) {
```

```
  SwapImpl<T>::make( t1, t2 );
}

// Displays a vector
template <typename T, typename Alloc = std::allocator<T> >
std::ostream& operator<<( std::ostream& out, const std::vector<T,Alloc>& V )
{
  for ( auto e : V ) out << e << " ";
  return out;
}

// Types for checking precisely swap method.
struct CheckSwap1 { int  swap( CheckSwap1& )              { return 0; } };
struct CheckSwap2 { void swap( const CheckSwap2& )        {} };
struct CheckSwap3 { void swap( CheckSwap3& ) const        {} };
struct CheckSwap4 { void swap( CheckSwap4&, CheckSwap4& ) {} };
struct CheckSwap5 { void swap( CheckSwap5& )              {} };

// Just for display
template <typename T>
std::string hasStandardSwap( const T& )
{
  return has_standard_swap<T>::value ? "Specialized" : "Generic    ";
}

int main( int argc, char* argv[] )
{
  std::cout << "has_standard_swap<int>::value="
            << has_standard_swap<int>::value << std::endl;
  std::cout << "has_standard_swap<std::vector<int>>::value="
            << has_standard_swap< std::vector<int> >::value << std::endl;

  int i = 5, j = 10;
  std::cout << "---- int --- " << hasStandardSwap(i)  << "-----" << std::endl;
  std::cout << "i=" << i << " j=" << j << std::endl;
  myswap( i, j );
  std::cout << "i=" << i << " j=" << j << std::endl;

  std::vector<int> v1 = { 17, 4, 2 };
  std::vector<int> v2 = { 12, 25, 14, 4, 6 };
  std::cout << "---- vector<int> - " << hasStandardSwap(v1) << "-----" << std::endl
      ;
  std::cout << "v1=" << v1 << " v2=" << v2 << std::endl;
  myswap( v1, v2 );
  std::cout << "v1=" << v1 << " v2=" << v2 << std::endl;

  // Only CheckSwap5 will induce the call to the specialized swap
  std::cout << "- CheckSwap1 -- " << hasStandardSwap( CheckSwap1() )  << std::endl;
  std::cout << "- CheckSwap2 -- " << hasStandardSwap( CheckSwap2() )  << std::endl;
  std::cout << "- CheckSwap3 -- " << hasStandardSwap( CheckSwap3() )  << std::endl;
  std::cout << "- CheckSwap4 -- " << hasStandardSwap( CheckSwap4() )  << std::endl;
  std::cout << "- CheckSwap5 -- " << hasStandardSwap( CheckSwap5() )  << std::endl;

  return 0;
}
```

Comme on peut le voir, `has_standard_swap<T>` vérifie plus précisément que `has_swap<T>` la signature de la méthode `T::swap`.

En résumé : Bonus ...

— Le mot-clé `auto` simplifie considérablement l'écriture de code en évitant d'avoir à écrire exactement le type de retour. Attention, cela ne doit pas vous empêcher d'avoir une bonne idée de ce que doit être le type de retour. D'ailleurs, s'il y a la moindre ambiguïté, une erreur sera levée par le compilateur.
— Les lambda-expressions sont très pratiques aussi pour simplifier le code, et évite la définition lourde d'objets-fonction (que l'on devait faire avant C++11).
— Le mécanisme SFINAE a permis le développement de concepts très subtils en C++ (dont les concept checks, mais aussi des mécanismes de spécialisation dès l'étape de compilation). En revanche, force est de reconnaître que le code devient extrêmement complexe et lourd. Vous comprenez pourquoi la courbe d'apprentissage du C++ est beaucoup plus lente que beaucoup d'autres langages de programmation.
— C++20 (ou C++2a maintenant) apporte des solutions pour simplifier tout ça. C'est une bonne nouvelle mais ce n'est encore guère supporté par les compilateurs.

A Bonnes pratiques ; trucs et astuces

A.1 Questions liées à l'efficacité

1. *Lorsque la valeur de retour est un objet un peu gros, comment éviter que l'objet soit créé dans la fonction appelée, puis réinstancié dans la fonction appelante ?*

   Exemple : Concaténation de deux chaînes C.

```
std::string concat( const char* prenom, const char* nom )
{ // cree la variable locale s
  std::string s = std::string( prenom ) + std::string( " " ) + std::
      string( nom );
  return s; // elle sera recopiée dans la fonction appelante
}
{
  std::string s2 = concat( "Jean", "Bon" ); // copie
}
```

   Discussion : Evidemment, on pourrait arguer qu'il suffit de ne pas faire une fonction, mais par exemple une procédure avec une référence modifiable en paramètre. C'est évidemment une solution. Néanmoins, on souhaite parfois réellement avoir le résultat comme valeur de retour (par exemple dans un foncteur), afin de l'utiliser simplement dans des expressions.

   Solution : Il faut créer l'objet `string` dans le `return`. A ce moment-là, il est directement créé dans le bon contexte (i.e., dans la variable de la fonction appelante).

```
std::string concat( const char* prenom, const char* nom )
{ // cree la variable locale s
  return std::string( std::string( prenom ) + std::string( " " ) +
      std::string( nom ) );
  // elle sera créée directement dans la fonction appelante
}
{
  std::string s2 = concat( "Jean", "Bon" ); // créé un seule fois
}
```

2. *Lorsqu'on évalue une expression ou lorsqu'on récupère le résultat d'une fonction (voir aussi plus haut), il y a beaucoup de variables temporaires qui sont créées, copiées intégralement dans un nouveau contexte, puis détruite. Comment éviter cela au maximum ?*

   Exemple :

   Discussion :

   Solution : C++11 apporte la notion de référence à une *rvalue* via la syntaxe `&&`. C'est un mécanisme assez subtil, mais assez puissant pour résoudre ces problèmes. Je conseille la lecture de `http://thbecker.net/articles/rvalue_references/section_01.html`, accessible si on parle anglais.

A.2 Questions liées à la lisibilité du code

1. *On aimerait pouvoir initialiser nos nouvelles classes comme les tableaux et structures en C, afin d'avoir un code bien compact et lisible. Comment est-ce possible lorsque le nombre d'arguments est inconnu, par un exemple pour un tableau ?*

   Exemple : On veut initialiser un "fixed array" `FArray` quelconque.

   ```
   typedef FArray<double,3> Ligne;
   typedef FArray<Ligne,3> Matrice;
   Matrice m = { { cos( M_PI/4.0 ),  sin( M_PI/4.0 ), 0 },
                 { -sin( M_PI/4.0 ), cos( M_PI/4.0 ), 0 },
                 { 0, 0, 1 } };
   ```

   Discussion : Le code précédent ne compile pas, et d'ailleurs on ne peut écrire de constructeur générique qui prendrait en compte tous les cas. Un hack possible est de d'abord initialisé un tableau à deux dimensions, puis de forcer une conversion et d'avoir un constructeur prenant en paramètre un pointeur.

   ```
   struct FArray { ...
     // Constructeur à partir du pointeur vers la première case d'un tableau.
     FArray( const Valeur* v )
     {
       for ( int i = 0; i < N; ++i )
         mData[ i ] = *v++;
     } ...
   };

   double tmp[ 3 ][ 3 ] = { { ... } };
   Matrice m( (const Ligne*) tmp[ 0 ] ); // ugly, beurk
   ```

   Solution : On utilise les *initializer-list* du nouveau standard c++0x ou c++11. Il faut les voir comme une conversion de tout bloc de valeurs `{ 3.5, 17.2, ... }` vers une liste quel'on peut parcourir. Cela donne :

   ```
   struct FArray {
    ...
     FArray(std::initializer_list<Valeur> l) // C++0x et C++11 seulement
     {
       Valeur* mPtr = mData;
       for ( typename std::initializer_list<Valeur>::const_iterator
               it = l.begin(), itEnd = l.end(); it != itEnd; ++it )
         *mPtr++ = *it;
     }
   };

   // Fonctionne maintenant sans problème
   Matrice m = { { cos( M_PI/4.0 ),  sin( M_PI/4.0 ), 0 },
                 { -sin( M_PI/4.0 ), cos( M_PI/4.0 ), 0 },
                 { 0, 0, 1 } };
   ```

ScopeGuard trick
Exercice :
Spécialiser une fonction recherche pour des conteneurs à accès randomisé triés.