Le langage C++

Cássia Trojahn dos Santos

ctrojahn@univ-tlse2.fr

Organisation de l'UE

1. Éléments de base du langage
2. Abstraction de données et programmation orientée objet
3. Programmation générique
4. *Standard Template Library* (STL)

Partie I

Éléments de base du langage

Quelques langages de programmation

- Langages impératifs (= procéduraux)
  - COBOL, Fortran, Pascal, C, C++, Ruby
- Langages déclaratifs (dont les langages fonctionnels)
  - Lisp, Caml, Haskell, Scala
- Langages orientés objets
  - Smalltalk, C++, Java, Ruby

C++

- C++ a été développé par Bjarne Stroustrup (AT&T Labs, 1983) à partir du langage C et il retient C en tant que sous-ensemble
- C++ est un langage de programmation polyvalent supportant :
  - l'abstraction de données
  - la programmation orientée objet
  - la programmation générique

Abstraction de données

- Il permet de définir de nouveaux types de données en s'appuyant sur les notions d'encapsulation et de polymorphisme

```
class Stack {
      public :
          // virtual (definie plus tard), = 0 (classes derivees la definiront)
          virtual void push (char c) = 0;
          virtual char pop () = 0;
}

void f(Stack& s_ref) {
     s_ref.push('c')
}

Array_stack as(200);
f(as);
List_stack ls;
f(ls);
```

- Dans l'exemple, $f()$ utilise l'interface Stack en ignorant complètement les détails de l'implementation de ce type de donnée
- Une classe qui fournit l'interface à divers autres classes est souvent nommée *type polymorphique*

Programmation orientée objet

- Définition d'objets (ses attributs et ses comportements) et leurs interactions
- Trois piliers : l'encapsulation, l'héritage et le polymorphisme

```
// bad solution
enum

class


     public
         void


void
    switch
        case
            // dessine circle;
            break
        case
            // dessine triangle;
            break
        case
            // dessine square;
            break
```

```
// good solution
class Shape {
      Color color;
      public :
          virtual void draw () = 0;
          virtual void rotate () = 0;
}

class Circle : public Shape {
      int radius;
      public :
          void draw() {
              // dessiner circle
          };

      ...
}

...
```

Programmation générique

- Mécanisme de généralisation de types de données en les transformant en modèle, grâce à la notion de template

```
template<class T>class Stack {
      T* v;
      int max_size;
      public :
          Stack (int size); // constructor
          ~Stack(); // destructor
          void push(T);
}

Stack<char> sc(200);
Stack<list<int>> sli(45);
```

- Dans l'exemple, lorsqu'on a besoin d'une pile, il ne s'agit pas toujours d'une pile de caractères (*push*(*char c*))
- Le préfixe template<class T> signale que T est un paramètre de la déclaration dont ce préfixe fait partie

C++ vs. C

- C++ retient C en tant que sous-ensemble
- C suit le paradigme de programmation procédural alors que C++ est un langage multi-paradigme (procédural ainsi qu'orienté objet)
- En C, les données ne sont pas 'sécurisées' alors qu'en C++ ils le sont (les données sont 'cachées', grâce au paradigme orienté objet)
- C++ supporte la surcharge (*overloading*) de fonctions alors que C ne le support pas
- Espace de noms (pour éviter le conflit de noms) est présent en C++ et absent en C
- Les entrées et sorties sont simplifiées en C++
- L'allocation dynamique de mémoire se fait plutôt à l'aide de new en C++, malloc/alloc en C (delete et free, respectivement)

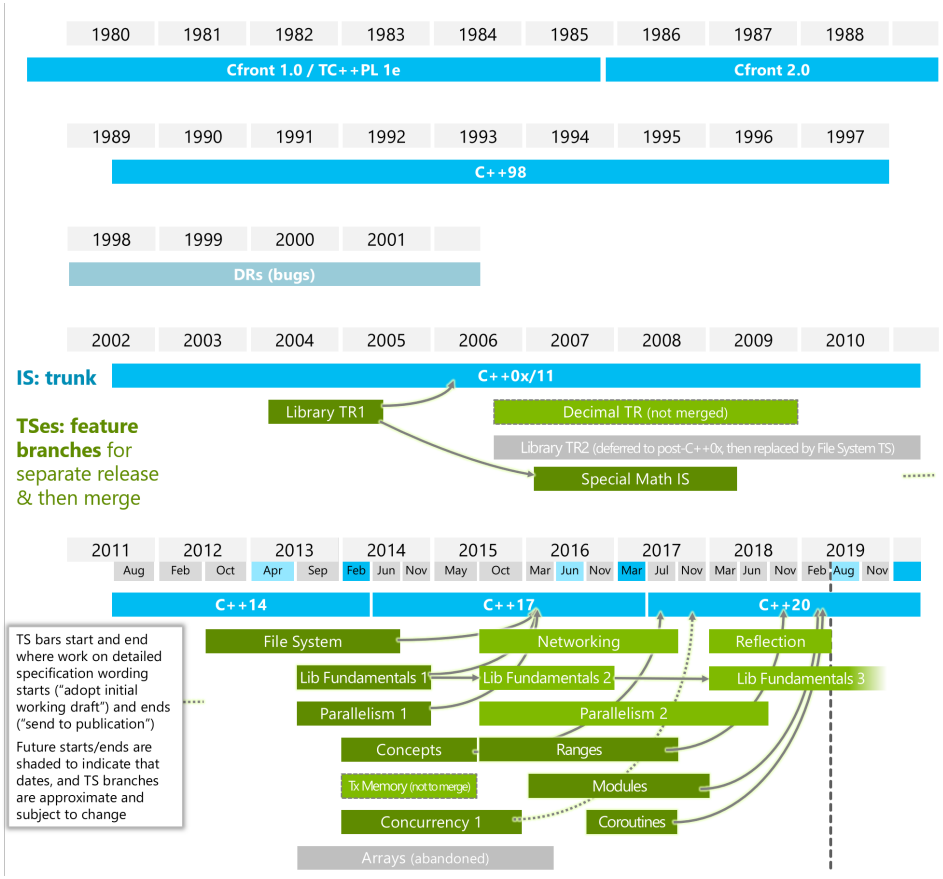
Évolution du langage

'Every professional developer should invest in learning the new language version and try introducing its benefits into projects.'

- C++ est normalisé par l'ISO (*International Organization for Standardization*)
  - C++11 : version supportée par la plupart de compilateurs
  - C++14 : livrée en 2014 (revision mineure)
  - C++17 : plusieurs compilateurs implémentent déjà certaines fonctionnalités (`http://en.cppreference.com/w/cpp/compiler_support` (consulté Jan/2019))

Évolution du langage

- *Recent milestones : C++17 published, C++20 underway*



Source : `https://isocpp.org/std/status` (consulté Mars/2019)

Bibliothèque standard

- Bibliothèque inclue dans le *C++ Standard* (*Standard Library* – STD) et qui fournit de différents supports et implémentations, tels que :
  - des possibilités du langage tel que la gestion de mémoire (*new*, *delete*, *unique_ptr*, etc.) et la gestion de *types à l'exécution* (*dynamic_cast*)
  - des types pour gérer les exceptions (*exception* et sa hiérarchie de classes)
  - des types pour supporter le traitement numérique de base (*log*, *pow*, etc.), le traitement des *nombre complexes* (*complex*), des opérations arithmétiques sur les vecteurs (*valarray*), etc.
  - un type pour manipuler efficacement les chaînes de caractères (*string*) – paramétré par le type de caractères qu'elle contient
  - des types pour la manipulation de flux d'entrées (*istream*) et sorties (*ostream*)
  - des conteneurs (et itérateurs) qui facilitent la manipulation d'ensemble d'objets (*vecteur*, *list*, *map*, etc.)
  - des algorithmes pour la manipulation de conteneurs (*for_each*, *find*, *sort*)
  - de support pour la programmation multithreading, pour les expressions régulières, pour la gestion du temps (time utilities), etc.

Hello L3 (v1) !

```cpp
#include <iostream>

int main() {
   std::string nom;
   std::cout << "Votre nom : ";
   std::cin >> nom;
   std::cout << "Bonjour " << nom;
   std::cout << std::endl;
   return 0;
}
```

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

int main() {
    string nom;
    cout << "Votre nom : ";
    cin >> nom;
    cout << "Bonjour " << nom;
    cout << endl;
    return 0;
}
```

- **#include** : cette directive permet d'inclure dans un programme la définition de certains objets, types ou fonctions
- **namespace std** : espace de nommage std (bibliothèque standard)
- **cout** : sortie console
- **cin** : entrée console

Espaces de noms

- Un espace de noms est un mécanisme permettant d'exprimer un regroupement logique
- Si certaines déclarations peuvent être associées selon certains critères, il est préferable de les placer dans un espace de noms commun pour exprimer ce fait
- Un espace de noms représente les séparations logiques des diverses parties d'un programme (il représente un module)

```
namespace Example {
        void f();
        void g();
}

void Example::f() { ... }
void Example::g() { ... }
void Example::h() { ... } // erreur : pas de fonction h dans Example
```

- Notez : : comme l'opérateur de portée

Espaces de noms

- Lorsque l'on utilise fréquemment un nom en dehors de son espace de noms, on peut utiliser la construction using [déclaration] (qui peut être placée dans la définition d'un autre espace de noms)

```
namespace Example {
          void f();
          void g();
}

void Example::f() { ... }
void Example::g() { ... }

void a() {
     using Example::f;
     using Example::g;
     g();
     f();
}
```

```
namespace Example {
          void f();
          void g();
}
namespace Example2 {
          void h();
          Example::f();
}
```

ou encore

```
namespace Example {
          void f();
          void g();
}
namespace Example2 {
          void h();
          using namespace Example;
}
```

- L'utilisation d'un espace de noms pour définir et déclarer une seule fonction est une solution exagérée !

Entrées et sorties

- iostream (Standard Input/Output Streams Library) : définit les objets de flots d'entrée et sortie standards
- Flot d'entrée : cin » expression1 .. » expression n
  - cin est un objet de la classe istream (durée statique)
  - » est l'opérateur binaire : le premier opérande est cin et le second opérande la variable à lire
  - Les caractères tapés sont stockés dans un tampon dans lequel cin fait la lecture des valeurs (les espaces, les tabulations et les fins de lignes sont de séparateurs)
- Flot de sortie : cout « expression1 .. « expression n
  - cout est un objet de la classe ostream (durée statique)
  - « est l'opérateur binaire : le premier opérande est cout et le second opérande l'expression à afficher
- « et » sont surchargés (voir plus loin) pour être utilisés pour l'affichage et la lecture des caractères, des entiers, des virgules flottantes, etc.

Fichiers sources

- C++ prend en charge la notion de compilation séparée du langage C
- Cela permet d'organiser un programme en un ensemble d'éléments partiellement indépendants
- Les déclarations liées à l'interface d'un module sont placées dans un fichier d'en-tête (.h/.hpp)
- Le code implémentant l'interface doit être placé dans un fichier de définition (.c/.cpp)
- Le mécanisme #include permet de rassembler les fragments de programmes sources dans une seule unité pour la compilation

```
#include <iostream> // a partir du repertoire d'inclusion standard
#include "myheader.h" // a partir du repertoire courant
```

Fichiers sources

- Déclaration de l'interface : fichier stack.hpp

```
namespace Stack {
     void push (char);
     char pop();
}
```

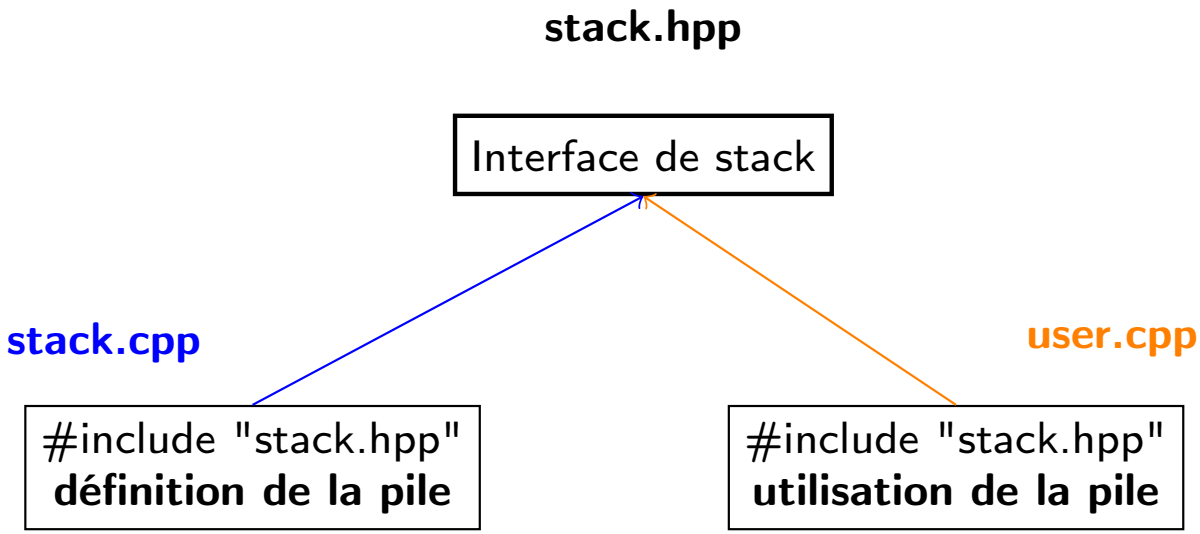
- Définition de l'implémentation : fichier stack.cpp

```
#include "stack.hpp" // definit l'interface
namespace Stack {
     const int max_size = 200;
     char v[max-size];
     int top = 0;
}

void Stack::push (char c) { .. }
char Stack::pop (char c) { .. }
```

- Code utilisateur : fichier user.cpp (les .cpp peuvent être compilés séparément)

```
#include "stack.hpp"

int main() {
    ...
    Stack::push('c');
    ...
}
```

Fichiers sources



- Une macro encore utile en C++ permet d'éviter l'inclusion double d'un même fichier .hpp dans la même unité de compilation

```
// fichier error.hpp
#ifndef CALC_ERROR_HPP
  #define CALC_ERROR_HPP
   ... // definitions du fichier error.hpp
#endif
```

  - dans l'exemple, le contenu du fichier entre *#ifndef* et *#endif* est ignoré par le compilateur si *CALC_ERROR* a été définie précédemment

using namespace dans les hpp ? NON !

- Une directive using dans un fichier d'en-tête injecte des noms dans chaque fichier qui inclut l'en-tête
- Vous ne devez absolument pas utiliser l'espace de noms dans les en-têtes :
  - il peut changer de façon inattendue la signification du code dans tous les autres fichiers qui incluent cet en-tête
- L'en-tête doit inclure uniquement les en-têtes dont il a besoin pour compiler.
- Bonnes pratique : `http://www.gotw.ca/publications/c++cs.htm` (consulté janvier 2018)

using namespace dans les hpp ? NON !

MISTAKE # 2: Incorporating "using namespace" statements at top level in a header file

Headers should define only the names that are part of the interface, not names used in its own implementation. However, a using directive at top level in a header file injects names into every file that includes the header.

This can cause multiple issues:

1. It is not possible for a consumer of your header file to undo the namespace include – thus they are forced to live with your namespace using decision, which is undesirable.
2. It dramatically increases the chance of naming collissions that namespaces were meant to solve in the first place.
3. It is possible that a working version of the program will fail to compile when a new version of the library is introduced. This happens if the new version introduces a name that conflicts with a name that the application is using from another library.
4. The "using namespace" part of the code takes effect from the point where it appears in the code that included your header, meaning that any code appearing before that might get treated differently from any code appearing after that point.

Recommendations:

1. Try to avoid putting any using namespace declarations in your header files. If you absolutely need some namespace objects to get your headers to compile, please use the fully qualified names (Eg. std::cout , std::string)in the header files.

```
//File:MyHeader.h:
class MyClass
{
private:
    Microsoft::WRL::ComPtr _parent;
    Microsoft::WRL::ComPtr _child;
}
```

2. If recommendation #1 above causes too much code clutter – restrict your "using namespace" usage to within the class or namespace defined in the header file. Another option is using scoped aliases in your header files as shown below.

http://www.acodersjourney.com/2016/05/top-10-c-header-file-mistakes-and-how-to-fix-them/ (consulté janvier 2018)

Mécanismes de base

- Types et déclarations
- Expressions et instructions
- Pointeurs et tableaux
- Fonctions
- Exceptions
- Structures

Types et déclarations

- Types fondamentaux
  - **bool** (1 octet) : true ou false, 1 ou 0, >0 ou <=0, respectivement
  - **char** (1 octet) : les 256 valeurs représentées peuvent être interprétées comme les valeurs de 0 à 255 (unsigned) ou de -127 à 127 (signed)
  - **int** (4 octets) : signed ou unsigned (signed par défaut), short (2 octets) ou long (8 octets)
  - **float** (4 octets), **double** (8 octets), **long double** (16 octets)
- Types énumérés
  - **enum** (1 octet) : pour représenter des ensembles de valeurs spécifiques (conversion implicite en int)
  - **enum class** : pour représenter des ensembles de valeurs (portée d'une classe) spécifiques (pas de conversion implicite en int)
- Absence d'information
  - **void** : retour de fonctions et pointeurs
  - **nullptr** : valeur de pointeur null (pas d'objet pointé)

Types et déclarations

- Types pointeurs
  - **type* var** : int* p (pointeur vers l'adresse mémoire d'une variable int)
- Types tableaux
  - **type var[]** : char tableau[20]
- Types références
  - **type& var** : int& p (autre nom pour l'objet p)
  - utilisé surtout pour la spécification des arguments et des valeurs renvoyées par les fonctions
- Constantes et constantes et expressions
  - **const**
  - **constexpr** : expression évaluée en temps de compilation (calculs simples)
  - C++14 : constantes modèles ont été introduites et certaines limitations de constexpr ont été levées (variables locales, boucles, etc.)
- Inférence de type
  - **auto**
  - C++14 : la syntaxe d'utilisation d'auto en tant que type de retour d'une fonction est simplifiée

Types et déclarations

```
int cont;
const double pi = 3.14159265358979;
char c = 's'
int x1;

// => {} ! mecanisme uniforme pour l'initialisation d'objets de tout type (pas de conversion
       de 'retrecissement' ?)
int x2 = {}; // valeur par default 0
int x3 {7}; // x3 = 7

int x0 {7.3}; // error: narrowing
int x1 = 7.3;

X x{v};
X* p = new X{v};
auto x2 = X{x};

unsigned int ui = 3U; // long int li = 3L;
// declare un nouveau nom pour un type donne'
typedef unsigned char uchar;
uchar i;

enum Flags {good=0,fail=1,bad=2,eof=4}; // Flags f; f = good;
enum class formes {Shape,Rectangle}; // if (type == formes::Shape)
```

Types et déclarations

```
// C++11 : recursion au lieu d'iteration ------------------------
constexpr int factorial(int n) {
    return n <= 1 ? 1 : (n * factorial(n-1));
}
int f = factorial(4); // calcule' en temps de compilation

// C++14 : variables locales et boucles sont autorisees pour constexpr
constexpr int factorial(int n) {
         int fact = 1;
         for (int i=2;i<=n;i++)
              fact = fact*i;
         return fact;
}

// C++11 -------------------------------------------------------
int a = 42; // base decimal (10)
int const a {42};
auto const a = 42;
int b = 052; // base octal (8)
int c = 0x2A // base hexadecimal (16)

// C++14 -------------------------------------------------------
// deja supporte par plusieurs compilateurs (extensions non-officielles)
int d = 0b00101010 // base binaire (2)
cout << d; // affiche 42
```

Types et déclarations

```
// C++14 : constantes modeles ----------------------------------------
<template typename T>
constexpr T pi = T(3.141592)
auto const piD = pi<double>;
auto const piF = pi<float>;
auto const piI = pi<int>; // piI = 3

// C++11 : auto en tant que retour de fonction -------------------------
auto add(int i,int ii) -> int {
    return i+ii;
}

// C++14 : simplification et uniformisation des syntaxes (il est possible d'omettre -> dans
     les fonctions avec auto)
auto add(int i,int ii) {
    return i+ii;
}
```

Tableaux

- Pour un type T, $T[taille]$ est le type tableau de *taille* éléments de type T
- Les éléments sont indexés de 0 à $taille - 1$
- Le nombre d'éléments et les limites du tableau doivent être une expression constante (sinon, il faut utiliser un *array* ou un *vector*, présentés plus tard)
- Les tableaux à plusieurs dimensions sont représentés comme des tableaux de tableaux

```
int tableau[5] = {1,2,3,4,5};
int tableau[5] = {1,2,3}; // = tableau2[5] = {1,2,3,0,0}
int tableau[] = {1,2,3}; // = tableau3[3] = {1,2,3}
int tableau[2][5] = {{1,2,3,4,5},{1,2,3,4,5}};
char tableau[10] = "Hello L3";
```

Chaînes de caractères et la classe string

- Pour simplifier la manipulation de chaînes de caractères (au lieu d'utiliser char*), la bibliothèque standard fournit la classe **string** (*basic_string*, en effet), implémentant un ensemble d'opérations (affectation, concaténation, comparaison, ajout en fin, recherche de sous-chaîne, etc.)

```
class std::basic_string { ... }
typedef basic_string<char> string;
```

```
string s1 = "Hello" ;
string s2 = " monde L3" ;
string s3 = s1 + " " + s2 + "\n!"
cout << s3 << s3.size() << endl;
```

```
string s1 = "Hello" ;
s1 = s1 + " monde L3" ;
s1 += "\n!"
cout << s1 ;
for (int i=0; i<s1.length(); ++i) {
     cout << s1[i];
}
```

- Les fonctions destinées à la manipulation de chaînes de style C se trouvent dans les en-têtes <string.h> et <cstring.h> (strcpy, strcmp, etc.)

Opérateurs et expressions

- Opérateurs classiques
  - Opérateurs arithmétiques : *,+,-,/,% (modulo)
  - Opérateurs de comparaison : <,<=,==,>,>=, !=
  - Opérateurs booléens : && (ET), || (OU), ! (négation)
- Pré et post incrément et décrément ($++$ et $--$)
  - $++\,var$ incrémente *var* et retourne la nouvelle valeur
  - $var\,++$ incrémente *var* et retourne l'ancienne valeur
  - en dehors d'une expression $++\,var$ et $var\,++$ sont équivalentes

Structures de contrôle

- Structures conditionnelles

```
if (condition)
   instruction

if (condition)
   instruction1
else
   instruction
```

```
switch (variable) {
       case valeur1 : {
            instruction ;
            break ;
       }
       case valeurN : {
            instruction ;
            break ;
       }
       default {
            instruction ;
            break ;
       }
}
```

Structures de contrôle

- Structures de répétition

```
for (initialisation;condition;increment)
instruction(s)

for (type var : 'conteneurs ou range')
instruction(s)

while (condition)
instruction(s)

do
   instruction(s)
while (condition(s))
```

Références

- Une référence fournit un autre nom pour un objet : la notation $X\&$ signifie la référence à X
- La principale application des références concerne la spécification d'arguments et de valeurs renvoyées par les fonctions en générales et les opérateurs surchargés (voir plus loin)
- Une référence doit être initialisée ($\neq$ affectation) avec un *lvalue* (un objet qui a une adresse mémoire)

```
int i = 1;
int& r = i;
int x = r; // x=1
r = 2; // i=2
```

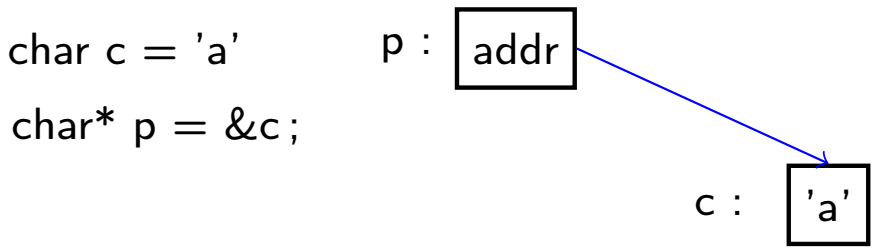
```
int ii = 0;
int& rr = ii;
rr++;

int* pp = &rr; // pp pointe sur ii
```

- Les références de constantes sont souvent utilisées en tant qu'arguments de fonctions : const T&

```
double& get_value(const string& s)
```

Pointeurs

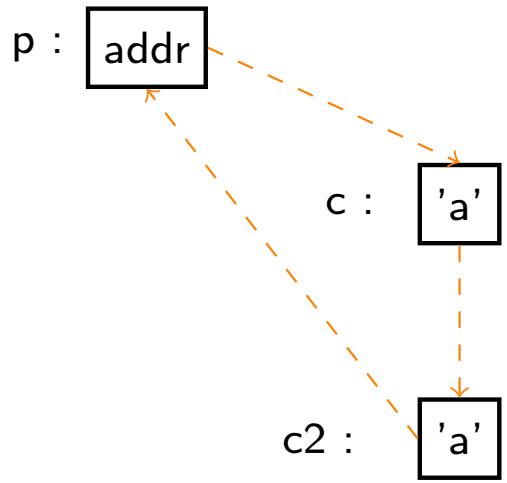
- Pour un type T, $T*$ est le type pointeur de T
- La variable $T*$ peut donc contenir l'adresse d'un objet de type T



Pointeurs

- L'opération fondamentale sur un pointeur est l'indirection
- Elle fait référence à l'objet vers lequel est dirigé le pointeur
- L'opérateur d'indirection unaire (préfixe) est * :

```
char c = 'a'
char* p = &c;
char c2 = *p;
```



- La variable pointée par p est c, et la valeur stockée dans c est 'a', donc la valeur de $*p$ est $'a'$

Pointeurs de type void

- Le pointeur de *tout type* d'objet peut être attribué à une variable de type *void*$*$
  - pour utiliser ce type d'objet, il faut explicitement le convertir en un pointeur d'un type spécifique
  - il est dangereux d'utiliser un pointeur converti en un type différent de celui de l'objet pointé
  - l'application principale de void* concerne la transmission de pointeurs à de fonctions que ne sont pas autorisées à émettre des hypothèses concernant le type d'objet et le renvoi (par des fonctions) d'objets non typés (conversion explicite au sein de la fonction)
  - ces fonctions généralement se trouvent à un niveau très bas du système et signalent une erreur de conception si utilisées à haut niveau

Pointeurs de type const

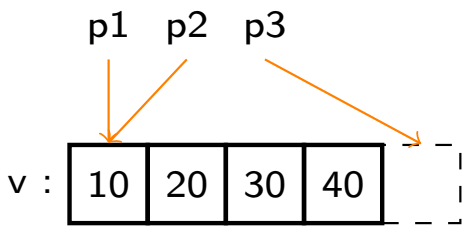
- L'utilisation d'un pointeur implique deux objets : le pointeur lui-même et l'objet pointé
- En préfixant la déclaration d'un pointeur avec const, l'objet pointé devient une constante (et non le pointeur lui-même)
- Pour déclarer le pointeur comme une constante, il faut utiliser la déclaration *const

```
char s[] = "L3";
const char* ccp = s; // pointeur de constante
ccp[1] = 'g'; // erreur : ccp pointe sur une constante
ccp = p; // ok

char *const cp = s; // pointeur constante
cp[1]='a'; // ok
cp = p; // erreur cp est une constante
```

Pointeurs et tableaux

- En C++, pointeurs et tableaux sont étroitement liés et il est possible d'utiliser le nom d'un tableau comme pointeur de son élément initial
- Il est donc possible de définir un pointeur en faisant référence à l'élément suivant le dernier élément du tableau (cet élément, cependant, ne fait pas partie du tableau et ne doit pas être utilisé pour la lecture ou l'écriture)

```
int v[] = {10,20,30,40} ;
int* p1 = v ; // pointeur de l'element initial (conversion implicite)
int* p2 = &v[0] ; // pointeur de l'element initial
int* p3 = &v[4] ; // pointeur de l'element suivant le dernier
```



Pointeurs et tableaux

- La conversion implicite d'un nom de tableau vers le pointeur de l'élément initial est largement utilisée dans les appels de fonctions C

```
char val[] = "UT2J";
// conversion implicite : char[] -> char* (!= pour char* -> char[])
char* p = val;

// size_t strlen (const char* str) : detecte la fin de la chaine par '\0'
strlen(p);
strlen(val);

// strcpy (char* destination, const char* source) de string.h
char* source = "Mirail";
char dest[10];
strcpy(dest,source);
```

- L'accès aux éléments d'un tableaux peut être obtenu, soit par l'intermédiaire d'un pointeur de tableau associé à un index, soit par l'intermédiaire d'un pointeur d'élément
- ++ incrémente le pointeur pour qu'il fasse référence à l'élément suivant du tableau :

```
char v[] = "UT2J";
for (char* p = v; *p!='\0'; p++) {
     cout << *p << '\n';
}
```

Pointeurs et tableaux

- Le résultat de l'emploi des opérateurs arithmétiques dépend du type de l'objet pointé
- Lorsqu'on applique un opérateur arithmétique à un pointeur p de type $T*$, on suppose que p pointe sur un élément d'un tableau d'objets de type T ($p+1$ pointe sur l'élément suivant et $p-1$ sur l'élément précédant)
- Cela implique que la valeur de $p+1$ sera $sizeof(T)$ plus grande que la valeur entière de p
- Lorsque l'on soustrait un pointeur à l'autre, le résultat correspond au nombre d'élément de tableau situé entre les deux pointeur (un entier)

```
int v1[10];
int* p1 = v1;
int* p2 = &v1[7];
int i1 = p2-p1; // i1 = 7
```

- Pour parcourir un tableau qui ne contient pas de caractère de fin (comme pour les chaînes de caractères), il faut fournir le nombre d'éléments d'une façon ou d'une autre !

Allocation dynamique

- La durée de vie d'un objet est déterminée par sa portée
- Il est souvent utile, cependant, de créer un objet dont l'existence est indépendante de la portée dans laquelle il a été conçu
- Il est donc utile (en termes de gestion de la mémoire) d'utiliser des variables dynamiques
- Ces variables n'ont pas d'objets indépendants (int a, par exemple) et peuvent seulement être désignées par de variables de pointeur déréférencés * ("*pa", par exemple)

Allocation dynamique

- Les variables dynamiques sont créées en utilisant new et sont détruites en utilisant delete
  - un objet crée par new existe jusqu'à ce qu'il soit explicitement détruit par delete
  - l'opérateur delete ne peut pas être appliqué qu'à un pointeur retourné par new (ou stockant nullptr ou zéro, opération sans effet !)
  - pour libérer la mémoire allouée il faut déterminer la taille de l'objet : un objet alloué via new occupe une place légèrement supérieure à celle d'un objet statique (un mot est, en effet, nécessaire pour stocker la taille de l'objet)

```
char* pc = new char;
*pc = 'a';
delete pc;
pc = nullptr;

char* s = new char[3];
delete[] s;
s = nullptr;
```

  - nullptr permet d'indiquer qu'un pointeur ne fait pas référence à un objet (l'utiliser plutôt que NULL ou 0)
  - ne pas supprimer un objet n'est pas considéré comme une erreur au niveau du langage mais plutôt comme une fuite de mémoire

Fonctions

- Une fonction ne peut pas être appelée que si elle a été préalablement déclarée

```
// Declarations :
double sqrt(double);
Elem* next_elem();
char* strcpy(char* to, const char* from);
void exit(int);

// Definitions
double sqrt(double d) {
      ...
      return d1;
}
```

- La sémantique de la transmission des arguments est identique à celle de l'initialisation de variables

Fonctions inline

- Il est possible de définir une fonction inline

```
inline int factoriel(int n) {
       return (n<2) ? 1 : n*factoriel(n-1);
}
```

- Le spécificateur inline est destiné au compilateur
- Au lieu de générer le code, puis d'appeler la fonction via le mécanisme habituel d'appel de fonction, il lui demande de remplacer directement l'appel de fonction par le code
- Par exemple, un compilateur peut générer la constante 720 pour un appel de *factoriel(6)*
- Cependant, cela dépend du niveau de sophistication du compilateur, où certains peuvent générer *6*factoriel(5)* ou encore un appel à *factoriel(6) non inline*

Fonctions et variables statiques

- Pour les variables locales déclarées comme static, un unique objet, alloué de façon statique, permet de représenter cette variable pour l'ensemble des appels à la fonction
- Cette variable sera initialisée uniquement à la première exécution de la définition
- Elle apporte une "mémoire" à la fonction sans introduire de variable globale susceptible d'être utilisée et altérée par d'autres fonctions

```
void f(int a) {
    while (a--) {
        static int n = 0;
        int x = 0;
        cout << "n == " << n++ << " x == " << x++ << "\n" ;
    }
}
```

Fonctions et arguments

```
void f(int val, int& ref) {
      val++;
      ref++;
}
void g() {
     int i = 1;
     int j = 1 ;
     f(i,j);
}
```

- Le premier argument *i* est transmis par valeur et *j* est transmis par référence (les tableaux sont toujours transmis par référence)
- Les fonctions modifiant les arguments par référence compliquent la lecture des programmes et devraient être évitées
- La transmission d'un objet de grande taille par référence peut toutefois se révéler plus efficace qu'une transmission par valeur : dans ce cas, l'argument pourrait être déclaré avec const
- Comme on la vue précédemment, la déclaration d'un argument pointeur avec const signale que la valeur de l'objet pointé par cet argument ne sera pas modifiée par la fonction

Fonctions et arguments

- A chaque appel d'une fonction, une nouvelle copie de ses arguments (passés par valeur) et de ses variables locales est créée
- Un pointeur ou une référence sur une variable locale ne devrait donc jamais être renvoyé (pour une autre voie, voir *unique_ptr* pour les pointeurs)

```
int& f() {
     int i = 0;
     ..
     return i; // muito ruim ! :'-(
}

int* f() {
     int i = 0;
     ..
     return &i; // so bad ! :-(
}

alternativement

int* f() {
     int* i = new int;
     (*i) = 0 ;
     return i; // less problematic but i has to be deleted somewhere ! ;-)
}
```

Fonctions, arguments par défaut et adresses

- Par soucis de simplicité, des arguments par défaut peuvent être définis (mais qu'en fin de liste) :

```
void print(int value, int base=10);
void f() {
     print(31);
     print(31,10);
     print(31,2);
}
```

- Sur une fonction, seules deux opérations sont envisageables : l'appeler ou récupérer son adresse
  - le pointeur obtenu en récupérant l'adresse d'une fonction peut ensuite être utilisé pour l'appeler

```
void error(string str) { ... }
void f() {
     void (*efct)(string); // sintaxe pour la definition d'un pointeur de fonction
     efct = &error; // efct pointe vers error
     efct("error"); // appelle error via efct
```

  - le compilateur 'découvrira' que *efct* est un pointeur et appellera la fonction pointée
  - l'intérêt des pointeurs de fonction est de permettre l'appel d'une fonction parmi un éventail de fonctions au choix
    - ex : il est possible de faire un tableau de pointeurs de fonctions et d'appeler la fonction dont on connaît l'indice de son pointeur dans le tableau

Fonctions, arguments par défaut et adresses

```
void SelectionSort(int *anArray, int nSize, bool (*pComparison)(int, int)) {
    for (int nStartIndex= 0; nStartIndex < nSize; nStartIndex++) {
        int nBestIndex = nStartIndex;
        for (int nCurrentIndex = nStartIndex + 1; nCurrentIndex < nSize; nCurrentIndex++) {
            if (pComparison(anArray[nCurrentIndex], anArray[nBestIndex])) // <====
                nBestIndex = nCurrentIndex;
        }
        swap(anArray[nStartIndex], anArray[nBestIndex]);
    }
}

bool Ascending(int nX, int nY) { return nY > nX; }
bool Descending(int nX, int nY) { return nY < nX; }

int main() {
    int anArray[9] = { 3, 7, 9, 5, 6, 1, 8, 2, 4 };
    bool (*pComparison)(int, int);
    pComparison = &Descending;
    SelectionSort(anArray, 9, Descending);
    PrintArray(anArray, 9);
    SelectionSort(anArray, 9, Ascending);
    PrintArray(anArray, 9);
    return 0;
}
```

Fonctions surchargées

- L'utilisation d'un même nom pour des opérations s'appliquant à de types différents est nommée surcharge
- La technique est largement utilisée dans la définition du langage C++ :
  - par exemple, l'opérateur + (surcharge d'opérateur) est utilisé pour l'addition des entiers ou des virgule flottantes
- Une erreur de compilation est générée si la correspondance n'est pas trouvée (ambiguïté, spécialement avec les entier) :

```
void print(double)
void print(long)

void f() {
     print(1L); // print(long)
     print(1.0); // print(double)
     print(1); // ?
}
```

- Les types retournés ne sont pas pris en compte dans la résolution de la surcharge

Fonctions lambda

- Depuis C++11, il est possible de définir une fonction lambda (anonyme)
- Ces fonctions sont très utiles, par exemple, en tant que prédicat pour les algorithmes de la bibliothèque standard (voir plus loin)

```
// = [capture-list] (params) { body }

int i = 10;
function<int (int)> pair = [](int p) { return p%2 == 0; };
cout << "Pair " << i << " ? " << pair(i) << '\n';

function<int (int)> f = [i](int j) {return i * j; };
cout << f(15);

find_if (vec.begin(), vec.end(), [](int i){ return i%2 == 1; });
```

- C++14 : la déduction de types est possible pour ce type de fonction, ce qui permet de définir des fonctions lambda polymorphiques

```
auto add = [](auto x, auto y) { return a+b; }
```

Exceptions

- Une exception lance l'interruption de l'exécution du programme à la suite d'un événement particulier
- L'objectif est de traiter ces événements qui en sont la cause, pour pouvoir rétablir l'exécution du programme dans son mode de fonctionnement normal
- La gestion d'exceptions est basée sur deux notions principales : la notion de déclencher (*throw*) l'exception et la notion de la capturer (*try/catch*)
- Lorsqu'une exception est déclenchée (lancée), elle traverse toute la pile de fonctions appelantes pour trouver une fonction capable de la capturer
- Les instructions de la fonction qui a lancée l'exception sont immédiatement terminées
- Si l'exception n'est pas capturée, l'exécution du programme est donc interrompue

Exceptions

- Déclencher une exception consiste à retourner une erreur sous la forme d'une valeur quelconque (int, char, string, DivisionZero, etc.), qui sera capturée dans un bloc *catch* correspondant

```
int division(int a, int b) {
    if (b==0)
        throw 0; // pas une bonne
             pratique !
    else
        return a/b;
}

try {
    cout << "1/0=" << division(1,0);
}
catch (int code) {
     cerr << "Exception " << code;
}
```

```
int division(int a, int b) {
    if (b==0)
        throw "Division par zero!";
             // ou string("Division
             par zero!");
    else
        return a/b;
}

try {
    cout << "1/0=" << division(1,0);
}
catch (const char* error) { // ou (
     string error)
     cerr << "Exception " << error;
}
```

- L'expression catch(...) peut être utilisée pour capturer toute exception pas capturée par un bloc *catch* précédent – ou *catch(exception e)* pour toute exception de la bibliothèque standard

Exceptions

- La bibliothèque standard définit la classe exception pour gérer les exceptions
- La méthode *what()* de cette classe permet de récupérer une description de l'exception déclenchée
- Quelques classes qui héritent de la classe exception :
  - *bad_exception* (type d'exception non déclarée)
  - *bad_cast* (conversion de types non valide)
  - *bad_alloc* (échec d'allocation de mémoire)
  - *range_error* (out of range)
  - *runtime_error* (erreur détectée lors de l'exécution)

```
try {
    int* myarray= new int[10000];
}
catch (bad_alloc& ba) {
      cerr << "bad_alloc caught: " << ba.what() << '\n';
}
```

Exceptions

- La classe exception peut être dérivée (voir plus loin) permettant la définition de nouveaux types pour la gestion d'exceptions
- La méthode what doit être surchargée : *virtual const char * what()*

```
class DivisionZero : public exception {
    public :
        const char* what() const throw() {
             return "Division par zero";
        }
} divisionZero;

int division(int a, int b) {
    if (b==0) {
        throw divisionZero;
    } else {
        return a/b;
    }
}

int main() {
     try {
         int d = division(1,0);
     } catch(DivisionZero e) {
       cerr << "Erreur : " << e.what() << '\n';
     }
     return 0;
}
```

Structures

- Une structure (**struct**) est un agrégat d'éléments de types arbitraires

```
struct adresse {
        string prenom;
        string nom;
        long int code_postal;
};


adresse ad;
ad.prenom = "Cassia";
ad.nom = "Trojahn dos Santos";
ad.code_postal = 31300;


// new pour allouer dynamiquement en memoire (tas)
adresse* ad1 = new adresse; // in C = malloc(sizeof(struct adresse));
ad1->prenom = "Cassia"; // ou (*ad1).nom

ad1->nom = "Trojahn dos Santos";
ad1->code_postal = 31300;
delete ad1;
ad1=nullptr;
```

Partie II

Programmation orientée objet et abstraction de données

Struct vs. Classe

- Une struct est un type définit par l'utilisateur
- Une classe est un type définit par l'utilisateur
- Une struct est une sorte de classe
- Dans une struct, tous les membres sont publics

```
struct Date {
      int d,m,y;
};
void init_date(Date& d,int,int,int);
void add_year(Date& d,int n);
void add_month(Date& d,int n);
void add_day(Date& d,int n);
```

```
struct Date {
      int d,m,y;
      void init(int,int,int);
      void add_year(int n);
      void add_month(int n);
      void add_day(int n);
};
```

Classes

- Pour imposer que les fonctions membres doivent être les seules à accéder directement aux attributs de **Date**, il faut déclarer une classe plutôt qu'une struct
- Une struct est une classe dont les membres sont publics par défaut – au contraire, toutes les membres d'une classe sont privées par défaut
- Il faut explicitement définir le niveau d'accès aux membres, en utilisant les mot clés private, protected, et public

```
class Date {
      int d,m,y;
      public :
        void init(int,int,int);
        void add_year(int n);
        void add_month(int n);
        void add_day(int n);
};
```

Contrôle d'accès

- Un membre d'une classe peut être private, protected, ou public
  - s'il est private, son nom ne peut pas être utilisé que par les fonctions membres et les amies (plus loin dans le cours) de la classe dans laquelle il est déclaré
  - s'il est protected, son nom ne peut pas être utilisé que par les fonctions membres et les amies de la classe dans laquelle il est déclaré ainsi que par les classes dérivées (plus loin dans le cours) de cette dernière
  - s'il est public, son nom peut être utilisé par toute fonction

Membres privés

- Il est possible de coder le descripteur d'accès private pour signaler que les membres suivants sont privés
- Les déclarations suivantes sont équivalentes :

```
class Date {
      int d,m,y;
      public :
        Date(int,int,int);
        Date& add_year(int n);
        Date& add_month(int n);
        Date& add_day(int n);
};
```

```
class Date {
      private :
        int d,m,y;
      public :
        Date(int,int,int);
        Date& add_year(int n);
        Date& add_month(int n);
        Date& add_day(int n);
};
```

Fonctions membres

- Les fonctions déclarées dans une définition de classe sont nommées fonctions membres
  - ces fonctions ne peuvent pas être appelées que pour une variable de type approprié (objet de la classe)
  - elles peuvent accéder aux variables membres sans référence explicite à un objet

```
class Date {
       int d,m,y;
       public :
         void init(int,int,int);
         void add_year(int n);
         void add_month(int n);
         void add_day(int n);
};
void Date::init(int dd,int mm, int yy) {
          d = dd;
          m = mm;
          y = yy;
}

Date today;
today.init(16,03,2017);
```

Fonctions membres

- Une fonction membre définie (et non simplement déclarée) dans la définition de classe est considérée comme inline
- Ce type de définition doit plutôt concerner les fonctions courtes et fréquemment utilisées

```
class Date {
      int d,m,y;
      public :
        void init(int,int,int);
        void add_year(int n);
        void add_month(int n);
        void add_day(int n);
        void print_date() { cout << d << "/" << m << "/" << y << endl; }
};
```

- On peut également définir la fonction en dehors de la définition de classe, en utilisant le mot clé inline

```
class Date {
      ...
      public :
          int day() const;
};

inline int Date::day() const { return d;}
```

Fonctions non-membres (assistantes)

- Il arrive souvent qu'un certain nombre de fonctions associées à une classe n'aient pas à être définies dans la classe elle-même, car elles n'ont pas besoin d'accéder directement à sa représentation

```
int diff(Date a, Date b);
Date next_weekday(Date d);
```

- La définition de telles fonctions dans la classe elle-même compliquerait l'interface de cette dernière
- Comment telles fonctions sont-elles 'associées' à la classe ? Leurs déclarations sont simplement placées dans le même fichier que la déclaration de la classe
- En plus, il est possible de rendre l'association explicite en incluant la classe et ses fonctions assistantes dans le même espace de noms

```
namespace Chrono {
  class Date { ... };
  int diff(Date a, Date b);
  Date next_weekday(Date d);
}
```

Constructeurs

- L'utilisation de fonctions telles que **init()** dans le but de fournir l'initialisation des objets d'une classe n'est pas très élégante et constitue une source d'erreurs
- Il n'est mentionné nulle part qu'un objet doit être initialisé et le programmeur peut facilement oublier de le faire ou le faire deux fois
- Une meilleure approche consiste à utiliser un constructeur qui est une fonction ayant le même nom de la classe et sans type de retour et qui est appelée implicitement lors de la création d'un objet

```
class Date {
       int d,m,y;
       public :
         Date(int,int,int);
         void add_year(int n);
         void add_month(int n);
         void add_day(int n);
};
Date::Date(int dd, int mm, int yy) {
     d = dd; m = mm; y = yy;
}
Date today(16,3,2017);
// ou Date today = Date(16,3,2017) ou Date today {16,3,2017} ou Date today = Date
     {16,3,2017}
```

- Un constructeur peut être surchargé

Constructeurs par défaut

- Un constructeur par défaut est un constructeur qu'il est possible d'appeler sans fournir d'argument

```
class Date {
       int d,m,y;
       public :
         Date(int d=17,int m=3,int y=2017); // ou Date()
         void add_year(int n);
         void add_month(int n);
         void add_day(int n);
};

Date today(16,3,2017);
Date tomorrow;
Date yesterday = Date(16,3,2017);
```

Initialisateurs de membres

- Les arguments pour les constructeurs peuvent être spécifiés dans une liste d'initialisation située dans la définition du constructeur (notez le : avant la liste d'initialisateurs)

```
class Date {
      int d,m,y;
      public :
        Date(int dd, int mm, int yy);
        void add_year(int n);
        void add_month(int n);
        void add_day(int n);
};
Date::Date(int dd,int mm,int yy) : d(dd), m(mm) { // ou d{dd}
     y = yy;
}
```

- Ces initialisateurs sont essentiels pour les types dont l'initialisation diffère de l'affectation (objets membres de classes sans constructeurs par défaut, membres const, et membres de type référence)
- En utilisant ces initialisateurs, on obtient une meilleure efficacité : *d* et *m* sont initialisés avec une copie de *dd* et *mm*, respectivement, alors que *y* est d'abord initialisé avec 0, puis une copie de *yy* lui est affectée

Destructeurs

- Un destructeur est une fonction appelée lors de la destruction de l'objet
- Il a le même nom de la classe, avec le préfixe '$\sim$'
- Les destructeurs doivent contenir le code pour nettoyer et libérer les ressources précédemment allouées (mémoire, fichiers, etc.)
- Ils sont appelés implicitement lorsqu'une variable automatiquement sort de la portée courante, lorsqu'un objet en mémoire dynamique est supprimé, etc.

```
class Table {
      Date* p_d;
      int taille;
      public :
        Table(int s=15) {
              p_d = new Date[taille=s];
        }
        ~Table() {
              delete[] p_d;
        }
}
```

Allocation dynamique

- Le constructeur d'un objet crée en mémoire dynamique est appelé par l'opérateur new
- Le destructeur d'un objet crée en mémoire dynamique est appelé par l'opérateur delete

```
Date* d1 = new Date;
Date* d2 = new Date;

...

delete d1;
delete d2;
```

Fonctions membres constantes

- Il est possible d'indiquer qu'une fonction ne modifie pas l'état de l'objet en utilisant const

```
class Date {
      int d,m,y;

      public :
        Date(int,int,int);
        int day() const { return d;}
        int month() const { return m;}
        int year() const { return y;}
};
```

- Lorsqu'une fonction membre const est définie en dehors de la classe, le suffixe const est toujours requis

```
int Date::year() const { return y; }
```

- Une fonction const peut être appelée par des fonctions const ou non-const, en revanche, une fonction membre non-const ne peut pas être appelée que par les fonctions non-const

Membres statiques

- Un attribut qui fait partie d'une classe, mais non d'un objet de cette classe est appelé membre static (il existe une copie et une seule d'un membre static)
- De façon analogue, une fonction ayant besoin d'accéder aux membres d'une classe, mais ne nécessitant pas d'être appelée par un objet particulier est nommée fonction membre static
- Un membre static est référencé sans faire référence à un objet : il faut le qualifier en utilisant le nom de la classe (: : au lieu de .)

```
class Date {
      int d,m,y;
      static Date default_date;
      public :
        Date(int dd=0,int mm=0,int yy=0);
        static void set_default(int dd, int mm, int yy);
};

...

Date::set_default(16,03,2017);
```

Membres statiques

- Les fonctions membres static peuvent être définies en dehors de la définition de la classe (pas besoin de répéter le mot clé static)

```
class Date {
       int d,m,y;
       static Date default_date;
       public :
         Date(int dd=0,int mm=0,int yy=0);
         static void set_default(int dd, int mm, int yy);
};
Date::Date(int dd,int mm,int yy) {
      d = dd ? dd : default_date.d;
      ...
}
// definition du membre default_date
Date Date::default_date(); // == Date Date::default_date(0,0,0)

// definition du membre set_default
void Date::set_default(int dd, int mm, int yy) {
     default_date = Date(dd,mm,yy);
}
```

Classes et fonctions amies

- Une déclaration de fonction membre spécifie trois éléments principaux :
  1. la fonction peut accéder à la partie privée de la classe
  2. la fonction se trouve dans la portée de la classe
  3. la fonction doit être appelée sur un objet (possède un pointeur *this*)

  - Une fonction static a les propriétés 1 et 2
  - Une fonction friend a la propriété 1

- Les fonctions amies ne sont pas des membres de la classe !

Classes et fonctions amies

```
class Rectangle {
      int width, height;
      public:
        Rectangle() {}
        Rectangle (int x, int y) : width(x), height(y) {}
        int area() {
            return (width*height);
        }
        friend Rectangle duplicate (const Rectangle&); // <==== amie !
};

// Pas dans la portee de la classe
Rectangle duplicate (const Rectangle& param) {
       Rectangle res;
       res.width = param.width*2; // pas de fonction get = amie peut acceder partie privee
       res.height = param.height*2; // pas de fonction get = amie peut acceder partie privee
       return res;
}

int main () {
    Rectangle foo;
    Rectangle bar (2,3);
    foo = duplicate (bar);
    cout << foo.area() << '\n';
    return 0;
}
```

Classes et fonctions amies

- Pour que toutes les fonctions d'une classe soient amies d'une autre, il faut ajouter le nom de la classe amie dans la définition de la classe

```
class Square;

class Rectangle {
    int width, height;
    public:
    int area () {
          return (width * height);
    }
    void convert (Square a);
};

void Rectangle::convert (Square a) {
     width = a.side;
     height = a.side;
}
```

```
class Square {
    friend class Rectangle;
    private:
      int side;
    public:
      Square (int a) : side(a) {}
};
```

Classes et fonctions amies

- La situation la plus courante d'utilisation de fonctions amies
  - certains opérateurs (tels que +), qui produisent une nouvelle valeur en fonction de celles de leurs arguments sont alors définis en dehors de la classe
  - c'est le cas de la surcharge d'opérateurs

- Réflexion : fonction(objet) ou objet.fonction() ?
  - inverse(matrice) ou matrice.inverse() ?
  - si inverse() inverse effectivement la matrice elle même au lieu de renvoyer une nouvelle Matrice, cette fonction devrait être une fonction membre

Autoréférence

- Pour pouvoir enchaîner les opérations sur un objet, une référence de l'objet mis à jour doit être réenvoyée

```
void f(Date& d) {
     d.add_day(1).add_month(1).add_year(1);
}
```

- Pour cela, chaque fonction devrait être déclarée de façon à renvoyer une référence

```
class Date {
       int d,m,y;
       public :
         Date(int,int,int);
         Date& add_year(int n);
         Date& add_month(int n);
         Date& add_day(int n);
};
```

Autoréférence

- Dans une fonction membre non-static, le mot clé this est un pointeur sur l'objet pour lequel la fonction a été appelée
- L'expression *this fait référence à l'objet pour lequel une fonction membre a été appelée (≡ opérateur d'indirection ->)

```
class Date {
       int d,m,y;
       public :
         Date(int,int,int);
         Date& add_year(int n);
         Date& add_month(int n);
         Date& add_day(int n);
};

Date& Date::add_year(int n) {
      this->y+=n; // ou (*this).y+=n;
      return *this;
}
```

Copie d'objets

- Par défaut, les objets d'une classe peuvent être copiés (membre à membre) – par initialisation ou par affectation (un objet est également copié dans trois autres cas : en tant qu'argument de fonction, en tant que valeur renvoyée par une fonction, et en tant qu'exception)
- Cependant, cela peut créer un effet surprenant lorsqu'elle s'applique aux objets d'une classe possédant de membres pointeurs
- La copie membre à membre ne représente pas la bonne sémantique pour la copie d'objets contenant de ressources dynamiquement gérées par une paire constructeur/destructeur :

```
class Table {
     int* p;
     int size;
     public :
        Table(int s=10) {
             p = new int[size=s];
        }
        ~Table() {
             delete[] p;
        }
};
```

```
Table t1;
Table t2 = t1; // initialisation de
     la copie
Table t3;
t3 = t2; // affectation de la copie
```

Copie d'objets

```
class Table {
      int* p;
      int size;
      public :
         Table(int s=10) {
               p = new int[size=s];
         }
         ~Table() {
               delete[] p;
         }
};
```

```
Table t1;
Table t2 = t1; // initialisation de
     la copie
Table t3;
t3 = t2; // affectation de la copie
```

- Dans l'exemple, le constructeur par défaut de Table est appelé deux fois : t1 et t3 (pas appelé pour t2 car cette variable a été initialisée par la copie) ; le destructeur est néanmoins appelé trois fois : t1, t2 et t3
- t1, t2 et t3 vont contenir chacun un pointeur vers le tableau alloué lors de la création de t1 (le pointeur du tableau alloué au moment de la création de t3 ayant été écrasé par l'affectation t3=t2 => fuite de mémoire)
- Le tableau crée par t1 se retrouve en t1, t2 et t3 et sera supprimé 3 fois (*double free or corruption*)

Copie d'objets

- De telles anomalies peuvent être évitées si l'on définit précisément ce qu'est la copie d'un objet, en définissant un constructeur de copie ainsi qu'un opérateur (surchargé) d'affectation = (voir plus loin pour la surcharge)
- Le développeur a la liberté de définir ces opérations de façon appropriée

```
class Table {
      ...
      public :
         Table(const Table&); // constructeur de copie
         Table& operator=(const Table&); // affectation par copie
};
Table::Table(const Table& t) {
     p = new int[sz=t.sz];
     for (int i=0;i<sz;i++) p[i] = t.p[i];
}
Table& Table::operator=(const Table& t) {
     if (this != &t) { // protection contre l'auto-affectation (t1=t1)
         delete[] p; // suppression des anciens elements
         p = new int[sz=t.sz]; // initialisation
         for (int i=0;i<sz;i++) p[i] = t.p[i]; //copie
     }
     return *this;
}
```

- Si le constructeur de copie et l'opérateur d'affectation ne sont pas explicitement définis, le compilateur les génère (par défaut)

move vs. *copy*

- L'opération de copier un objet peut s'avérer coûteuse pour les objets de grande taille
- Pour éviter cette opération, le constructeur et l'opérateur move ont été définis
- L'opération de 'déplacement' se fait uniquement lorsque la source de la valeur est un objet sans adresse en mémoire (ne existe que pendant l'évaluation d'une expression), c'est à dire un rvalue
  - le constructeur *move* est appelé lorsqu'un objet est initialisé avec un *rvalue*
  - de même, l'affectation *move* est appelée lorsqu'un *rvalue* est attribuée à un objet

```
Table t1 (100); // constructor Table(int)
Table t2; // constructor Table(int)
t2 = t1; // copy assignment
Table t3 = move(genTable()); // move constructor : <rvalue std::move(expression)>
t2 = Table(10); // move assign
```

move vs. *copy*

```
class Table {
      int* p;
      int size;
      public :
         Table(int s=10);
         ~Table();
         Table(const Table&); // constructeur de copie
         Table(Table&&); // constructeur par 'move'
         Table& operator=(const Table&); // affectation par copie
         Table& operator=(Table&&); // affectation par 'move'

};
...
Table::Table(Table&& t) {
       p = t.p;
       t.p = nullptr;
}

Table& Table::operator=(Table&& t) {
       delete[] p;
       p = t.p;
       t.p = nullptr;
       return (*this);
}
```

Opérateurs surchargés

- Il est souvent pratique de prévoir des fonctions qui permettent l'emploi d'une notation conventionnelle
- Par exemple, operator== ci-dessous définit un opérateur d'égalité permettant de comparer deux Dates :

```
bool operator==(Date a,Date b) {
     return a.day()==b.day() &&
            a.month()==b.month() &&
            a.year()==b.year();
}

bool operator!=(Date a,Date b);
bool operator>(Date a,Date b);
bool operator<(Date a,Date b);
Date& operator++(Date& d); // semantique ?
Date& operator--(Date& d); // semantique ?
```

Opérateurs surchargés

- Les principaux opérateurs qui peuvent être surchargés
  - +,-,*,/,-=,+=, », «, ==, !=, >, <, *=, /=, %, %=,++,–,etc.
- Un opérateur peut être soit le membre d'une classe, soit défini dans un espace de noms
- Opérateurs membres vs. non-membres
  - les opérateurs (tels que =+) modifiant la valeur de leur premier argument sont définis dans la classe elle-même
  - les opérateurs (tels que +), qui produisent une nouvelle valeur en fonction de celles de leurs arguments sont alors définis en dehors de la classe – intérêt des fonctions amies
- Les opérateurs définis dans des espaces de noms peuvent être identifiés à partir de leur types d'opérandes, exactement comme il est possible d'identifier des fonctions à partir de leurs types d'arguments

Opérateurs surchargés

- Opérateurs binaires vs. opérateurs unaires
  - un opérateur binaire peut être défini soit par l'intermédiaire d'une fonction membre non statique recevant un argument soit par intermédiaire d'une fonction non membre recevant deux arguments
    - pour tout opérateur binaire @, aa @ bb peut être interprété en aa.operator@(bb) ou en operator@(aa,bb)
  - un opérateur unaire (préfixe ou postfixe) peut être défini soit par intermédiaire d'une fonction membre non statique ne recevant pas d'argument soit par l'intermédiaire d'une fonction non membre recevant un argument
    - pour tout opérateur unaire préfixe @, @aa peut être interprété en aa.operator@() ou en operator@(aa)
    - pour tout opérateur unaire postfixe @, aa@ peut être interpreté en aa.operator@(int) ou en operator@(aa,int)

```
operator++(); // prefixe
operator++(int) // postfixe
```

    - l'argument *int* permet d'indiquer au compilateur que la fonction doit être appelée pour une application en version postfixe de ++
    - le type *int* n'est jamais utilisé (l'argument permet de différencier les applications préfixe et postfixe)

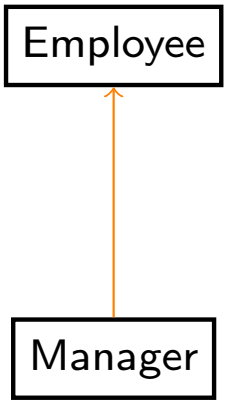
Opérateurs surchargés

```
class Date {
      private :
        int day,month,year;
      public :
        Date(int d,int m,int y) {day = d; month=m; year=y;}
        Date& add_year(int n);
        Date& add_month(int n);
        Date& add_day(int n);
        friend bool operator==(Date a,Date b);
        friend bool operator>(Date a,Date b);
        friend bool operator<(Date a,Date b);
        friend ostream& operator<<(ostream& out,Date& date);
};

ostream& operator<<(ostream& out, Date& date) {
  return out << date.day << "-" << date.month << "-" << date.year << '\n';
}

int main() {
    Date a = Date(22,1,2015);
    Date b = Date(22,1,2015);
    cout << a << " " << b;
    if (a==b) { // raccourci de 'operator==(a,b)'
        ...
    }
}
```

Classes dérivées

- Une classe dérivée (sous-classe) hérite des propriétés de sa classe de base (super-classe ou classe mère)
- Pour une classe ayant une seule classe de base directe, la relation est dite héritage simple

```
class Employee {
      private :
          string first_name;
          Date hiring_date;
          short department;
      public :
           void print() const;
};

class Manager : public Employee {
      set<Employee*> group;
      short level;
};
```



  - un *Manager* est (aussi) un *Employee*, pas nécessairement un *Employee* est (aussi) un *Manager*

Classes dérivées

- Un membre d'une classe dérivée peut utiliser les membres publics et protégés de sa classe de base, mais pas les membres privés

```
class Employee {
      private :
          string first_name;
          string family_name;
          Date hiring_date;
          short department;
      public :
           string full_name() const;
           void print() const;
};

class Manager : public Employee {
      set<Employee*> group;
      short level;
};

void Manager::print() const {
     cout << full_name(); // ok !
     cout << family_name; // erreur !
}
```

Classes dérivées

- Un membre d'une classe dérivée peut utiliser les membres publics et protégés de sa classe de base, mais pas les membres privés

```
class Employee {
      private :
          Date hiring_date;
          short department;
      protected :
          string first_name;
          string family_name;
      public :
           void print() const;
           string full_name() const;
};

class Manager : public Employee {
      set<Employee*> group;
      short level;
      public :
           void print() const;
};
```

```
void Manager::print() const {
     cout << full_name(); // ok !
     cout << family_name; // ok !
}

alternativement :

void Manager::print() const {
     Employee::print();
     cout << level;
}
```

Classes dérivées et contrôle d'accès

- Comme c'est le cas pour un membre, il est possible de déclarer une classe de base private, protected, ou public

```
class Temporary { };
class Secretary : public Employee { };
class Consultant : private Temporary { }; // = class Consultant : Temporary { }
```

- Le spécificateur d'accès contrôle l'accès aux membres de la classe de base, ainsi que la conversion des pointeurs et références depuis le type de la classe dérivée vers le type de la classe de base

Classes dérivées et contrôle d'accès

- Le spécificateur d'accès contrôle l'accès aux membres de la classe de base, ainsi que la conversion des pointeurs et références depuis le type de la classe dérivée vers le type de la classe de base :
  - Si **B** est une base private, ses membres publics et protégés peuvent être utilisés que par les amies et fonctions membres de la dérivée **D**. Les amies et fonctions membres de **D** sont les seules à pouvoir convertir un **D*** en un **B***
  - Si **B** est une base protected, ses membres publics et protégés peuvent être utilisés que par les amies, les fonctions membres et des classes dérivées de **D**. Les amies, les fonctions membres et les classes dérivées de **D** sont les seules à pouvoir convertir un **D*** en un **B***
  - Si **B** est une base public, ses membres publics peuvent être utilisés par toute fonction. De plus, ses membres protégés peuvent être utilisés par les membres, les amies et les classes dérivées de **D**. Toute fonction peut convertir un **D*** en un **B***

Classes dérivées et constructeurs

- Si la classe de base et la classe dérivée possèdent de constructeurs, l'un d'entre eux doit être appelé
- L'appel au constructeur de la classe de base se fait dans la liste d'initialisation de la classe dérivée
  - cette stratégie est cohérente : on commence par initialiser les attributs en provenance de la classe de base avant d'initialiser les derniers introduits
- Le constructeur par défaut de la classe de base est automatiquement appelé

```
class Employee {
      string first_name, family_name;
      short department;
      public :
          Employee(const string& n,short d);
      };
Employee::Employee(const string& n,short d): family_name(n), department(d) {}

class Manager : public Employee {
      set<Employee*> group;
      short level;
      public :
          Manager(const string& n,short d,short lvl);
      };
Manager::Manager(const string& n,short d,short lvl): Employee(n,d), level(lvl) {}
```

Classes dérivées et constructeurs

- Pour hériter un constructeur de la classe de base, le mot clé using peut être utilisé

```
class Employee {
      string first_name, family_name;
      short department;
      public :
          Employee(const string& n,short d);

      };

class Manager : public Employee {
      set<Employee*> group;
      short level;
      public :
          using Employee::Employee; // <==================
          Manager(const string& n,short d,short lvl);
      };

Manager::Manager(const string& n,short d,short lvl): Employee(n,d), level{lvl} {
}

// sans using
Manager m ("Arnaud",1,4); // ou Manager m {"Arnaud",1,4};
// avec using
Manager m ("Arnaud",1); // ou Manager m {"Arnaud",1};
```

Classes dérivées

- Lorsqu'une classe dérivée définit une fonction membre ayant le même nom qu'une fonction de la classe base, la fonction de la classe base est masquée
- Pour appeler explicitement la fonction de la classe de base, il faut utiliser l'opérateur de portée ' : :'

```
class Employee {
      ...
      void print() {
      }
};
class Manager : public Employee {
      ...
      void print() {
           Employee::print(); // Appel de print() de la classe Employee
      }
};
Manager m;
m.print(); // Appel de print() de la classe Manager
```

Classes dérivées et copie d'objets

- La copie d'objets est définie par le constructeur de copie (initialisation) et par l'affectation
- Par défaut, seule la partie "commune" est copiée – opération de découpage (*slicing*)

```
void afficher(Employee e){
     e.print();
}
...
Employee e = Employee("Cassia",1);
Manager m = Manager ("Jaco",1,2);
afficher(e);
afficher(m);
```

```
Employee e = Employee("Cassia",1);
Manager m = Manager ("Jaco",1,2);
e = m ;
```

- L'une des raisons pour lesquelles on transmet des pointeurs et références d'objets dans une hiérarchie est justement d'éviter cette opération, préserver le comportement polymorphique (plusieurs types de *Employee* peuvent exister !) et améliorer l'efficacité du code

Vers le polymorphisme

- À partir d'un pointeur de type base **B***, comment identifier le type dérivé de l'objet effectivement pointé par **B*** ?

```
class Employee {
      ...
      public :
           void print() const;
};
class Manager : public Employee {
      ...
      public :
          void print() const;
};
...
Employee *e = new Manager();
e->print();
```

- s->print() : quelle fonction sera appelée ? *Employee* :: *print*() ou *Manager* :: *print*()
  - par défaut, la fonction du type déclaré (type statique == Employee) sera appelée
  - cette opération est nommée résolution statique de liens

Polymorphisme

- Le polymorphisme permet à un même code d'être utilisé avec différents types, en impliquant des implémentations plus abstraites et extensibles
- En C++, le comportement polymorphique est garantit à l'aide des fonctions virtuelles (et objets manipulés à l'aide de pointeurs et références !)

```
class Employee {
      ...
      public :
          virtual void print() const; // <== virtual
};
class Manager : public Employee {
      ...
      public :
         void print() const;
}

Employee *e = new Manager();
e->print();
```

- Quelle fonction sera appelée ? *Employee* :: *print*() ou *Manager* :: *print*()
  - un appel de fonction virtuelle va d'abord appeler (si elle existe et est accessible) la fonction du type dynamique (== Manager)
  - cette opération est nommée résolution dynamique de liens

Polymorphisme

- Une classe qui définit au moins une fonction virtuelle est dite polymorphe
- Une fonction virtuelle est autorisée d'être redéfinie dans une classe dérivée
- La signature de la redéfinition dans la classe dérivée doit être identique à la définition dans la classe de base (sinon, il s'agit d'une surcharge ! – très peu de modifications sont permises – que pour le type retourné)

```
class Employee {
      private :
         ...
      public :
          virtual void print() const;
};

class Manager : public Employee {
      private :
          ...
      public :
        void print();
};
```

Fonctions virtuelles

- Une fonction virtuelle doit être définie pour la classe dans laquelle elle est initialement déclarée (sauf si elle est déclarée comme étant une fonction virtuelle pure – plus loin)
- Elle peut être utilisée même lorsque aucune classe n'est dérivée de sa classe
- Une classe dérivée ne nécessitant pas sa propre version de la fonction virtuelle n'as pas besoin d'en fournir une

```
class Employee {
      ...
      public :
          virtual void print();
};

void Employee::print(){
     cout << "Employee ... " << family_name << " " << departement;
}

class Manager : public Employee {
      ...
      public :
        void print();
};
```

override

- Une fonction ayant la même signature qu'une fonction virtuelle de la classe de base implicitement redéfinit la fonction base
- Cependant, si la signature n'est pas respectée il s'agit d'une surcharge et non d'une redéfinition
- Le mot clé override explicitement exprime une redéfinition
- Si le programmeur utilise override pour une signature qui ne correspond pas à la signature de la fonction qui devrait être redéfinie, le compilateur le signalera

```
class Employee {
      ...
      public :
          virtual void print();
};

void Employee::print(){
     cout << "Employee ... ";
}

class Manager : public Employee {
      ...
      public :
        void print() override;
};
```

... donc

- ... pour obtenir un comportement polymorphique en C++
  - les fonctions membres concernées doivent être virtual
  - les fonctions membres concernées doivent être redéfinies dans les classes dérivées
  - les objets doivent être manipulés via des pointeurs ou des références
- Lorsqu'on manipule un objet directement (plutôt que par l'intermédiaire d'un pointeur ou d'une référence), son type exact est connu par le compilateur et le polymorphisme n'est plus nécessaire à l'exécution

Classes abstraites

- Certaines classes représentent des concepts abstraits pour lesquels il ne peut pas exister d'objet
- Une classe contenant une ou plusieurs fonctions virtuelles pures est une classe abstraite
- Une fonction virtuelle pure est indiquée par les mots-clés : virtual .. = 0 ;
- Une classe abstraite est une interface et ne peut pas être instanciée

```
class Shape {
      Color color;
      public :
          virtual void draw () = 0;
          virtual void rotate (int) = 0;
};
Shape s; // erreur : classe abstraite !
```

Classes abstraites

- La classe dérivée dans laquelle n'apparaît pas la définition de toutes les fonctions virtuelles pures est également une classe abstraite

```
class Shape {
      Color color;
      public :
          virtual void draw () = 0;
          virtual void rotate (int) = 0;
};
class Rectangle : public Shape {
      public :
          void draw ();
};
Rectangle r(); // erreur ! classe abstraite
```

- Une application importante des classes abstraites consiste à fournir une interface sans exposer de détails des différentes implémentations

Héritage multiple

- Une classe dérivée peut elle-même être une classe de base

```
class Employee { };
class Manager : public Employee { };
class Director : public Manager { };
```

- Une classe peut aussi posséder plus d'une classe de base directe : héritage multiple

```
class Temporary { };
class Consultant : public Temporary, public Manager { };
```

Ambiguïtés dans l'héritage multiple

- Deux classes de base peuvent contenir des fonctions membres de même nom
- Il est donc nécessaire de lever l'ambiguïté pour sélectionner l'une de ces fonctions

```
class Temporary {
      ...
      void print() const;
};
class Manager {
      ...
      void print() const;
};
class Consultant : public Temporary, public Manager { ... };

void f (Consultant *c) {
     c->print(); // erreur : request is ambiguous
     c->Manager::print(); // ok
     c->Temporary::print(); //ok
};
```

Ambiguïtés dans l'héritage multiple

- Il est généralement préférable de résoudre tels problèmes en définissant une nouvelle fonction dans la classe dérivée

```
class Consultant : public Temporary, public Manager {
      ...
      void print const () {
           Temporary::print();
           Manager::print();
      }
};

void f (Consultant *c) {
     c->print();
}
```

- Si *Consultant* :: *print*() n'est pas résolue au niveau de la classe elle même, le compilateur cherchera dans la pile de classes et utilisera la première définition trouvée

Partie III

Programmation générique et *Standard Template Library (STL)*

Programmation générique

1. Lorsqu'on a besoin d'une liste, il ne s'agit pas toujours d'une liste de caractères : une liste représente un concept général, indépendamment de la notion de caractère
   - afin de pouvoir définir de types abstraits et manipuler leurs données sans avoir besoin de connaître leur type, le mécanisme de modèles est fourni
   - un type 'liste de caractères' peut être généralisé en un type 'liste d'entiers' en utilisant un paramètre du modèle
   - une classe contenant un ensemble d'éléments d'un certain type est nommée conteneur
2. Lorsqu'on a besoin d'un algorithme de trie, il ne s'agit pas toujours de trier une liste : on peut vouloir trier des vecteurs, des tableaux, etc.
   - le paradigme de programmation générique en C++ permet de paramétrer les algorithmes à l'aide des conteneurs
   - la méthode adoptée (pour éviter d'écrire des fonctions pour chaque type de conteneur ou de faire des conversions vers une structure de données spécifique) consiste à s'appuyer sur la notion de séquence et à les manipuler via des itérateurs

Modèles

- Les modèles permettent de supporter directement la programmation générique, où les paramètres représentent de types de données
- Ce mécanisme autorise un type à apparaître sous la forme d'un paramètre de la définition d'une classe ou d'une fonction
- Il permet la généricité au niveau de types et de dissocier le traitement des objets manipulés
- Chaque abstraction importante de la bibliothèque standard est représentée sous la forme d'un modèle (ostream, complex, list, map, etc.) – *Standard Template Library* (STL)

Modèles

```
template<class T>class Stack {
      T* v;
      int max_size;
      public :
          Stack (int s);
          ~Stack();
          void push(T);
};
...
Stack<char> sc(200);
Stack<int> si(45);
```

- Le préfixe template<class T> signale que T est un paramètre de la déclaration dont ce préfixe fait partie
- Ce préfixe indique que l'on déclare un modèle et qu'un type T apparaîtra en argument dans la déclaration
- L'argument T étant introduit, il est ensuite utilisé de façon transparente (exactement de la même façon que les autres noms de types)

Modèles

- À partir de la définition d'un modèle de classe et d'arguments de modèle, le compilateur génère la définition d'une classe (ou de fonctions dans le cas de modèle de fonctions)
- La génération de ces classes (ou fonctions) est appelée instanciation (l'instantiation de modèles est *Turing complete !*)
- On nomme spécialisations les classes et fonctions générées
- Une classe générée à partir d'un modèle de classe est une classe ordinaire
- L'utilisation d'un mécanisme de modèle n'entraîne aucun mécanisme d'exécution autre que ceux utilisés pour une classe créée 'à la main'
- Les modèles représentent un moyen puissant de générer du code à partir des définitions

Modèles

- Les membres d'une classe modèle sont déclarés et définis exactement de la même façon que ceux d'une classe ordinaire
- Ils ne sont pas nécessairement définis dans la classe modèle elle-même
- Lorsque l'un d'entre eux est défini en dehors de la classe, il doit être explicitement déclaré comme modèle

```
template<class T>class Stack {
      T* v;
      int max_size;
      public :
          Stack (int s);
          ~Stack();
          void push(T);
};

template<class T> void Stack<T>::push(T) {
    ..
}
```

- Dans la définition de la fonction *push*, notez les mots-clés template<class T> avant le type de retour de la fonction et <T> avant l'opérateur de portée : :

Modèles : exemple

```
template<class T> class CTableau {
private :
   T m_elements[10];
public :
   CTableau();
   void setElement(int indice, T element);
   T getElement(int indice);
};

template<class T> CTableau<T>::CTableau() { ... }

template<class T> T CTableau<T>::getElement(int indice) {
       return m_elements[indice];
}

template<class T> void CTableau<T>::setElement(int indice, T element) {
        m_elements[indice] = element;
}


 ...
 CTableau<int> tableauInt;
 CTableau<char> tableauChar;
 tableauInt.setElement(0,2017);
 tableauChar.setElement(0,'c');
 ...
```

Paramètres de modèles

- Un modèle peut contenir plusieurs paramètres de types ordinaires (int, short, etc.), de paramètres typés, et de paramètres modèles

```
template<class T,class C> class MaClasse {
      T* v;
      C* c;
      ...
};
```

```
template<class T,int max> class CTableau {
private :
   T m_elements[max];
public :
   CTableau();
   void setElement(int indice, T element);
   T getElement(int indice);
};
```

Paramètres de modèles

- Un type utilisé comme paramètre doit fournir l'interface attendue par le modèle (par exemple, ==, », >, <, etc.)
- Par exemple, pour trier une list<Personne>, les opérateurs de comparaison doivent être implémentés pour ce type d'objet
- La valeur de l'argument de modèle peut être une expression constante, l'adresse d'un objet ou d'une fonction, ou un pointeur de membre

```
template<class T,int max> class CTableau {
private :
   T m_elements[max];
public :
   CTableau();
   void setElement(int indice, T element);
   T getElement(int indice);
};
```

```
CTableau<int,10> tableauInt;
CTableau<char,20> tableauChar;
```

- Les erreurs liées à un paramètre de modèle ne peuvent pas être détectées avant la première utilisation de ce modèle pour un argument en particulier !

Paramètres de modèles

- Pour la définition de fonctions membres (en dehors de la classe) dans les modèles recevant plusieurs paramètres : notez void CTableau<T,max> au lieu de void CTableau<T,**int** max> dans la signature de la fonction

```
template<class T,int max> class CTableau {
private :
   T m_elements[max];
public :
   CTableau();
   void setElement(int indice, T element);
   T getElement(int indice);
};


...
template<class T,int max> void CTableau<T,max>::setElement(int indice, T element) {
        m_elements[indice] = element;
}
```

Paramètres de modèles

- Pour chaque paramètre, une valeur par défaut peut être définie

```
template<class T,int max=10> class CTableau { ... }
...
CTableau<int> tableauInt;
CTableau<char,20> tableauChar;
CTableau<> tableauInt2;


// ou encore

template<class T,class C=Cmp<T>> class MyClass {
        ...
        public :
            // tout type C doit fournir eq et lt !!
            int compare(const T& p1, const T& p2) {
                if (!C::eq(p1,p2))
                    return C::lt(p1,p2) ? -1 : 1;
            }
};
...
```

Modèles de fonctions

- Une autre application des modèles consiste à définir des modèles de fonctions
- Ces modèles jouent un rôle essentiel dans les algorithmes génériques s'appliquant à une grande variété de conteneurs (voir plus loin)

```
// declaration
template<class T> void sort(vecteur<T>&);

// definition Shell sort (Knuth, Vol. 3. pag 84)
template<class T> void sort(vecteur<T>& v) {
   const size_t = v.size();
   for (int gap=n/2;0<gap;gap/=2)
       for (int i=gap;i<n;n++)
           for(int j=i-gap;0<=j;j-= gap)
               if (v[j+gap] < v[j]
                   swap(v[j], v[j+gap]); // modele de la biblioteque standard C++
}

//utilisation
void f(vector<int>& vi,vector<string>& vs) {
     sort(vi);
     sort(vs);
     ...
}
```

- Comme pour les classes modèles, plusieurs paramètres peuvent être utilisés : template<class T, class C> void mon_sort(vecteur<T>&, C) ;

Polymorphisme : modèles vs. héritage

- Le polymorphisme fourni par les fonctions virtuelles est nommé polymorphisme d'exécution
- Ce qui est fourni par les modèles est nommé polymorphisme de compilation ou paramétrique
- Quand faut-il utiliser l'un ou l'autre ?
  - dans les deux cas, les objets partagent un ensemble commun d'opérations
  - si aucune relation hiérarchique n'est pas requise entre ces objets, il est préférable de les utiliser sous forme d'arguments de modèles
  - si les types réels ne peuvent pas être connus à la compilation, il est préférable de les représenter comme des classes dérivées d'une classe de base commune

Standard Template Library

- La *Standard Template Library* (STL), mise en oeuvre à l'aide de modèles, est l'un des composants principaux de la *Standard Library* (STD)
- Elle fournit différents supports et implémentations, tels que :
  - la classe string : qui permet la manipulation efficace de chaînes de caractères
  - des conteneurs : qui facilitent la manipulation d'ensemble d'objets (*array*, *vecteur*, *list*, *map*, etc.)
  - des itérateurs : qui permettent de parcourir de conteneurs
  - des algorithmes : qui peuvent être utilisés sur les conteneurs (*for_each*, *find*, *sort*, etc.)
- Ne réinventez pas la roue ! Faites appel aux bibliothèques ! !

Classe string

- La classe string permet de gérer efficacement des chaînes de caractères
- La gestion d'allocation mémoire est faite par la classe
- Elle implémente plusieurs fonctions membres qui permettent le traitement de chaîne de caractères :
  - itérateurs : begin, end, rbegin (reverse), rend, etc. (voir plus tard)
  - capacité : size, length, max_size, resize, clear, empty, etc.
  - accès aux éléments : operator[], at, back, front
  - modificateurs : operator+=, append, insert, erase, replace, swap
  - opérations : copy, find, find_first_of, find_last_of, substr, compare

Itérateurs

- Un itérateur est tout objet pointant vers un élément dans un ensemble d'éléments, ayant la capacité de parcourir ces éléments en utilisant un ensemble d'opérateurs (au moins l'opérateur d'incrément ++ et l'opérateur d'indirection *)
- Les itérateurs peuvent être utilisés pour parcourir des conteneurs sans que le programmeur ait besoin de connaître le type réel employé pour identifier les éléments
- La forme la plus évidente d'itérateur est un pointeur : un pointeur peut pointer vers des éléments dans un tableau, et peut les parcourir à l'aide de l'opérateur d'incrément ++
- Il existe plusieurs types de itérateurs : chaque type de conteneur a un type spécifique d'itération destiné à itérer à travers ses éléments

Itérateurs : exemple avec vector

- Déclaration d'un itérateur

```
vector<int> vi (10);
vector<int>::iterator it;
vector<int>::reverse_iterator rit;
```

- Parcours d'un conteneur à l'aide d'un itérateur

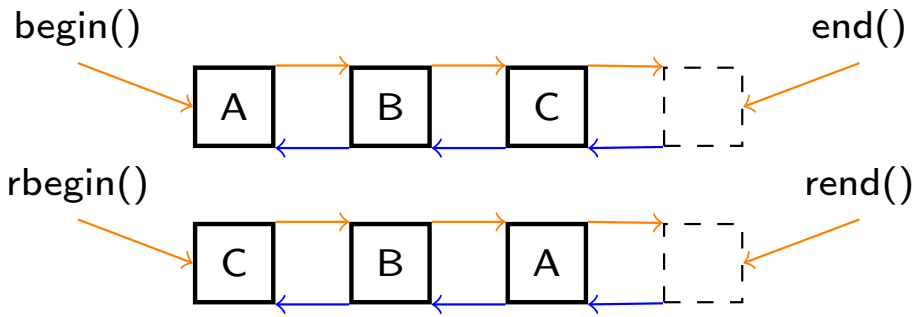
```
//------------------------
int i = 0;
it = vi.begin();
while (it != vi.end()) {
   *it = ++i;
   it++;
}

//------------------------
int total = 0;
for (it = vi.begin(); it != vi.end(); it++) {
    total += *it;
}

// ou
total = 0;
for (rit = vi.rbegin(); rit!= vi.rend(); rit++) {
     total += *rit;
}
```

Itérateurs : exemple avec vector

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

using namespace std;

int main() {
    vector<int> vi; // ou vector<int> vi {1,2,3};
    vi.push_back(1);
    vi.push_back(2);
    vi.push_back(3);

    vector<int>::iterator it;
    it = vi.begin();

    while (it != vi.end()) {
       int a = *it;
       cout << a << endl;
       it++;
    }
}
```

Itérateurs

- Pour la plupart des conteneurs, les fonctions membres publiques suivantes sont disponibles (à quelques exceptions près) :
  - begin() : pointe sur le premier élément
  - end() : pointe sur l'élément suivant le dernier élément
  - rbegin() : pointe sur le premier élément de la séquence inverse (le dernier)
  - rend() : pointe sur l'élément suivant le dernier élément de la séquence inverse (avant premier)
  - cbegin() : iterateur const de begin() – un *const_iterator* pointe vers un élément const et il ne peut être utilisé pour modifier le contenu pointé (même si l'objet n'est pas lui-même const)
  - cend() : iterateur const de end()
  - crbegin() : iterateur const rbegin()
  - crend() : iterateur const rend()



Conteneurs

- Une classe dont l'objectif principal est de stocker des objets est couramment nommée conteneur
- Les conteneurs peuvent être classés dans 2 catégories :
  - conteneurs séquentiels : ils rangent leurs éléments de façon linéaire (array, vector, list, deque), où l'ajout et la suppression sont facilitée
  - conteneurs associatifs : les éléments sont associés à une clé (set, map, multiset, multimap), où la recherche est facilitée

Conteneurs séquentiels

- array : un tableau de taille fixe
- vector : un vecteur dynamique de taille variable
- forward_list : une liste simplement chaînée
- list : une liste doublement chaînée
- deque : une file à double sens (double accès)

- à partir de ces conteneurs, queue (une file FIFO), stack (une pile LIFO) et priority_queue (une file avec priorité qui conditionne l'ordre pour atteindre le sommet), sont définis en fournissant des interfaces appropriées

array

- Un array est un conteneur de séquence de taille fixe
- Internement, un array ne stock aucune information d'autre que les éléments qu'il contient
- L'efficience d'un array est comparable à celle d'un tableau ordinaire
- Cette classe ajoute simplement un ensemble de fonctions pour faciliter la manipulation de ce type d'objet

array

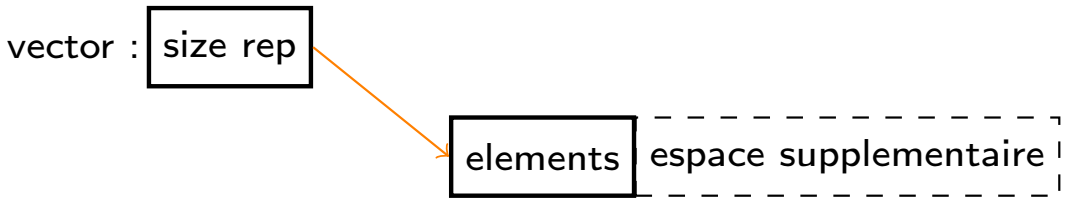
- Constructeurs
  - utilise des constructeurs (et destructeurs) implicite
  - la taille de l'array est indiquée en tant que paramètre du modèle
  - le constructeur d'initialisation de listes {} facilite l'initialisation des arrays
- Fonctions membres principales
  - operator[] : permet d'accéder à l'élément indiqué par l'indice (pas de contrôle de la plage de valeurs d'indice)
  - at : permet d'accéder à l'élément indiqué par l'indice (contrôle d'indice)
  - front : renvoie une référence sur le premier élément
  - back : renvoie une référence sur le dernier élément
  - empty : vérifie si l'array est vide
  - size : renvoie le nombre d'éléments effectivement stockés dans l'array
  - swap(array<T>&) : échange le contenu de deux array
- Itérateurs : tous les itérateurs vus précédemment

array

```cpp
#include <iostream>
#include <array>

using namespace std;

int main () {
  array<int,5> myarray = {10,22,24,6,9};
  cout << "Mon array contient : " << myarray.size() << " elements" << endl;
  for (auto it = myarray.begin(); it != myarray.end(); ++it)
       cout << *it << ' ';
  cout << endl;

  // Traite comme un tuple => fonction non-membre get<indice>(array)
  get<0>(myarray) = 45;
  cout << get<0>(myarray) << endl;
  return 0;
}
```

vector

- Internement, les vecteurs utilisent un tableau alloué dynamiquement pour stocker leurs éléments
  - ce tableau peut être réaffecté (ajouts de nouveaux éléments), ce qui implique l'attribution d'un nouveau tableau et le déplacement de ses éléments – tâche coûteuse et donc les vecteurs ne sont pas réaffectés à chaque ajout d'élément
  - au lieu de cela, les vecteurs allouent un espace de stockage supplémentaire pour répondre à une possible croissance, et ils peuvent donc avoir une capacité de stockage réelle supérieure au nombre d'éléments strictement nécessaire pour contenir leurs éléments
  - les bibliothèques peuvent mettre en œuvre différentes stratégies pour équilibrer l'utilisation de la mémoire et les réaffectations, mais en tout cas, les réaffectations doivent se faire en temps log par rapport à la taille du vecteur
  - par rapport aux arrays, les vecteurs consomment plus de mémoire, en échange de la possibilité de gérer le stockage dynamique



vector

- Par rapport aux autres conteneurs séquentiels (list, par exemple), les vecteurs sont très efficaces dans l'accès à leurs éléments – accès continu en mémoire
- Ils sont relativement efficaces pour l'ajout et la suppression d'éléments (qui sont faits toujours à la fin du vecteur)
- Par contre, pour les insertions et les suppressions dans d'autres positions que la fin, cela implique déplacer tous les éléments qui étaient après la position donnée (si l'espace supplémentaire alloué le permet), et sur ces conditions les vecteurs sont moins efficaces que les list

vector

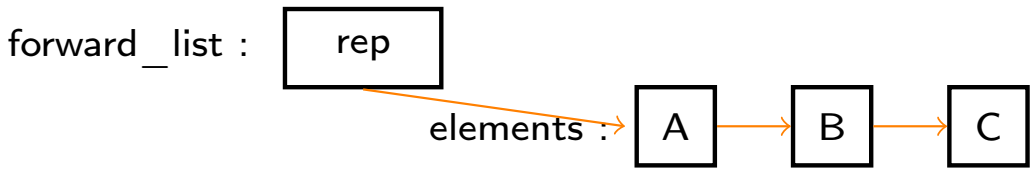
- Constructeurs
  - défaut, vector(size), vector(size,value), vector(iterator,iterator), vector(vector&), etc.
- Fonctions membres principales : sauf indication contraire, tout ce qui concerne vector s'applique à l'ensemble des conteneurs standards
  - push_back(T) : ajoute un élément à la fin du tableau dynamique
  - pop_back(T) : supprime le dernier élément
  - insert(iterator,T) : ajoute un élément dans la position de l'itérateur
  - erase(iterator) : supprime l'élément dans la position de l'itérateur
  - clear : supprime tous les éléments
  - front : renvoie une référence sur le premier élément
  - back : renvoie une référence sur le dernier élément
  - empty : vérifie si le vecteur est vide
  - capacity : renvoie le nombre d'éléments stockable (l'allocation actuelle)
  - size : renvoie le nombre d'éléments effectivement stockés dans le vecteur
  - swap(vector<T>&) : échange le contenu de deux vecteurs
- Itérateurs : tous les itérateurs vus précédemment

vector

```
#include<vector>

vector<int> v1 {1,2,3,4,5,6,7,8,9,10};

...

struct Entry { .. };
vector<Entry> phone_book {}; // pas d'elements

Entry e;
e.nom = "Trojahn";
e.prenom = "Cassia";
phone_book.push_back(e); // ou vector<Entry> phone_book {e};

void print_entry(int i) {
     cout << phone_book[i].nom << endl;
}

void add_entries(int n) {
     phone_book.resize(phone_book.size()+n);
}
```

forward_list

- Il s'agit d'une liste simplement chaînée, où l'insertion et la suppression se font en temps constant
- L'accès aux éléments n'est pas réalisé à l'aide d'un indice, comme dans le cas des *array*, *vector* ou *deque*
- On recherchera plutôt un élément à l'aide d'une valeur donnée
- L'insertion et la suppression à n'importe quelle position sont efficaces (pas de réallocation comme pour les vecteurs)



forward_list

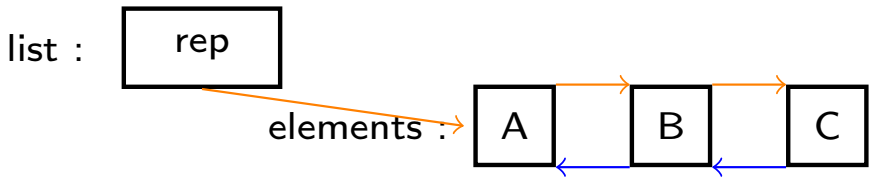
- Constructeurs
  - défaut, forward_list(size), forward_list(forward_list&), etc.
- Fonctions membres principales (la plupart de celles fournies par *vector* sont aussi valables pour *forward_list*) :
  - emplace_front(T) (emplace_after) : construit et ajoute un élément au début
  - push_front(T) : ajoute un élément en début de liste
  - pop_front(T) : supprime le premier élément
  - insert_after(T)(erase_after(T)) : ajoute un élément après une position donnée
  - emplace(T,iterator) : construit et ajoute un élément dans la position de l'itérateur
  - splice_after(iterator,iterator) : transfère les éléments d'une liste à une autre
  - merge(list,list) : fusionne deux listes
  - remove(T) : supprime le noeud contenant une valeur spécifique
  - remove_if(predicat) : supprime les éléments remplissant la condition exprimée par *predicate* (un prédicat est implémenté comme une fonction lambda, une fonction ou une classe)
  - reverse : reverse l'ordre des éléments de la liste
- Itérateurs : pas d'itérateurs *reverse* (rbegin et rend) (+ before_begin et cbefore_begin)

forward_list

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

int main () {
    forward_list<pair<int,char>> liste;
    liste.emplace_front(30,'c');

    auto it = liste.before_begin();
    liste.emplace_after(liste.before_begin(),20,'b');
    liste.emplace_front(10,'a');

    for (auto elem : liste)
        cout << elem.first << ' ' << elem.second << endl;
    return 0;
}
```

list

- Il s'agit d'une liste doublement chaînée : séquence optimisée pour l'insertion et la suppression d'éléments
- L'accès aux éléments n'est pas réalisé à l'aide d'un indice, comme dans le cas des vecteurs ou deques
- On recherchera plutôt un élément à l'aide d'une valeur donnée
- L'insertion et la suppression à n'importe quelle position est efficace (pas de réallocation comme pour les vecteurs)



list

- Constructeurs
  - défaut, list(size), list(size,value), list(iterator,iterator), list(list&), etc.
- Fonctions membres principales (celles fournies par *vector* sont aussi valables pour *list*, sauf *capacity*) :
  - emplace_back(T) : construit et ajoute un élément à la fin de la liste
  - emplace_front(T) : construit et ajoute un élément au début de la liste
  - emplace(T,iterator) : construit et ajoute un élément dans la position de l'itérateur
  - splice(list,list) : transfère les éléments d'une liste à une autre
  - merge(list,list) : fusionne deux listes
  - remove(T) : supprime le noeud contenant une valeur spécifique
  - remove_if(predicate) : supprime les éléments remplissant la condition exprimée par *predicate* (un prédicat est implémenté comme une fonction lambda, une fonction ou une classe)
  - reverse : reverse l'ordre des éléments de la liste
- Itérateurs : tous les itérateurs vus précédemment

list

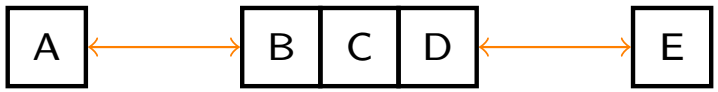
```
#include <iostream>
#include <list>
using namespace std;

bool sup (const pair<int,char> paire) {
     return (paire.first>10);
}
void print_list(const list<pair<int,char>> liste) {
     cout << "ma liste contient:";
     for (auto& x: liste)
          cout << " (" << x.first << "," << x.second << ")";
     cout << endl;
}
int main () {
    list<pair<int,char>> liste;
    liste.emplace_front(10,'a');
    liste.emplace_front(20,'b');
    liste.emplace_front(30,'c');
    print_list(liste);
    liste.remove_if(sup); // fonction
    // lambda fonction : [capture-list] (params) mutable(optional) -> ret { body }
    int x = 20;
    liste.remove_if([x](pair<int,char> p) { return p.first > x; });
    print_list(liste);
    return 0;
}
```

list

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

struct Entry {
      int id; string nom; string prenom; string phone;
};

int main () {
   list<Entry> phone_book;
   Entry e, e1, e2;
   e.id = 1; e.nom = "Trojahn"; e.prenom = "Cassia"; e.phone = "0611234516";
   e1.id = 2; e1.nom = "Clemente"; e1.prenom = "Gilles"; e1.phone = "0656456788";
   e2.id = 3; e2.nom = "Dupont"; e2.prenom = "Marie"; e2.phone = "0678678890";

   phone_book.push_front(e); // ajout au debut
   phone_book.push_back(e1); // ajout a la fin

   list<Entry>::iterator it = phone_book.begin();
   phone_book.insert(it,e2); // ajout avant l'element auquel it (iterator) fait reference

   for (it=phone_book.begin(); it != phone_book.end();it++)
        cout << it->nom << " " << it->prenom << endl;
}
```

deque

- Un conteneur du type deque est une file à double accès (*double-ended queue*)



- Il s'agit d'une séquence optimisée de telle sorte que les opérations s'appliquant à ses deux extrémités soient presque aussi efficaces que pour une liste et que l'accès à un élément ait une efficacité proche de celle d'un vecteur
- L'insertion et la suppression d'éléments situés au milieu ont une efficacité plus proche de celle d'un vecteur que celle d'une liste
- Par conséquent, on utilise généralement une deque lorsque les ajouts et les suppressions se situent aux extrémités

deque

- Les deques fournissent une interface très similaire aux vecteurs, mais avec des modes de fonctionnement très différents :
  - les vecteurs utilisent un seul tableau qui est éventuellement réaffecté – au contraire, les éléments d'une deque peuvent être dispersés dans différents emplacements de mémoire (non continus), avec un mécanisme pour permettre l'accès direct à l'un de ses éléments (interface d'accès séquentielle)
  - par conséquent, les deques sont plus complexe à l'intérieur que les vecteurs, mais cela leur permet d'agrandir de manière plus efficace dans certaines circonstances, en particulier avec de très longues séquences (où les réaffectations deviennent coûteuses)

deque

- Constructeurs
  - défaut, deque(size), deque(size,value), deque(iterator,iterator), deque(deque&), etc.
- Fonctions membres principales : mêmes types et opérations qu'un vector et mêmes opérations sur élément de premier rang qu'une liste
  - push_back(T)(push_front) : ajoute un élément à la fin (début) du deque
  - pop_back(T) (pop_front) : supprime le dernier (premier) élément
  - insert(iterator,T) : ajoute un élément dans la position de l'itérateur
  - erase(iterator) : supprime l'élément dans la position de l'itérateur
  - clear : supprime tous les éléments
  - front : renvoie une référence sur le premier élément
  - back : renvoie une référence sur le dernier élément
  - empty : vérifie si le deque est vide
  - capacity : renvoie le nombre d'éléments stockable (l'allocation actuelle)
  - size : renvoie le nombre d'éléments effectivement stockés
  - swap(deque<T>&) : échange le contenu de deux deques
  - operator[], at : accès à un élément
  - emplace_front (emplace_back) : construit et ajoute un élément au début (fin)
- Itérateurs : les mêmes que pour les vector

deque

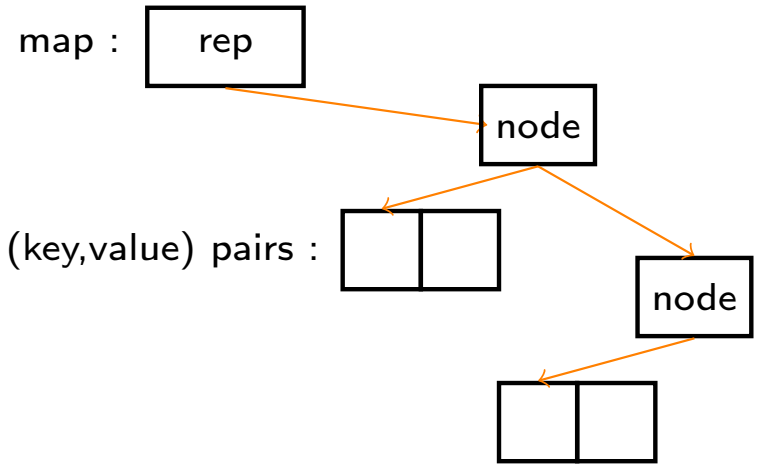
```
#include<deque>
#include<iostream>
using namespace std;

struct Entry {
      int id; string nom; string prenom; string phone;
};

int main () {
   deque<Entry> phone_book;

   Entry e, e1, e2;
   e.id = 1; e.nom = "Trojahn"; e.prenom = "Cassia"; e.phone = "0611234516";
   e1.id = 2; e1.nom = "Clemente"; e1.prenom = "Gilles"; e1.phone = "0656456788";
   e2.id = 3; e2.nom = "Dupont"; e2.prenom = "Marie"; e2.phone = "0678678890";

   phone_book.push_front(e); // ajout au debut
   phone_book.push_back(e1); // ajout a la fin

   deque<Entry>::iterator it = phone_book.begin();
   phone_book.insert(it,e2); // ajout avant l'element auquel it (iterator) fait reference

   for (int i=0; i < phone_book.size(); i++)
        cout << phone_book[i].nom << " " << phone_book[i].prenom << endl; // access indice
}
```

Conteneurs associatifs

- Les conteneurs associatifs gèrent de paires de valeurs (généralement selon un ordre précis)
- À partir d'une valeur nommée clé, il est possible d'accéder à une autre, nommée valeur mappée
  - map : un tableau associatif traditionnel dans lequel une seule clé est associée à une valeur
  - multimap : un *map* dans lequel la duplication des clés est autorisée
  - set : un *map* "dégénéré" dans lequel la valeur (unique) est la clé
  - multiset : un *set* dans lequel la duplication des valeurs (clés) est autorisée

map

- Un map est une séquence de pairs (clé,valeur) offrant une recherche rapide au moyen de la clé
- Chaque clé est associée à une seule valeur (chaque clé est unique)
- Les éléments doivent être classés et une opération inférieur-à doit exister pour les types clés (les éléments sont gardés donc en ordre de telle sorte que le parcours d'un map se fasse dans l'ordre)
- Lorsque aucune raison n'exige le tri du conteneur, il est préférable d'envisager l'utilisation d'un unordered_map
- Un map est le plus souvent implémenté à l'aide d'une forme d'arborescence



map

- Constructeurs
  - défaut, map(key_compare), map(iterator,iterator), map(map&), etc.
- Fonctions membres principales
  - insert(T) : ajoute un élément dans le conteneur – l'opération n'est pas garantie (retourne pair<iterator,bool>)
  - emplace(T) : construit et ajoute un élément dans le conteneur
  - erase(T) : supprime un élément donné (ou l'élément dans la position d'un itérateur ou plusieurs éléments pointés entre deux itérateurs)
  - clear : supprime tous les éléments
  - empty : vérifie si le conteneur est vide
  - size : renvoie le nombre d'éléments effectivement stockés dans le conteneur
  - count : renvoie le nombre d'éléments effectivement stockés dans le conteneur
  - swap(map&) : échange le contenu de deux map
  - operator[], at : accès à un élément
  - find(T) : renvoie un itérateur pointant vers l'élément correspondant à la clé donnée
- Itérateurs : les mêmes que pour les autres conteneurs

pair et tuple

- La bibliothèque standard définit la structure modèle pair avec 2 champs : first et second
- Cette struct est souvent utilisée (implicitement) dans les conteneurs associatifs et peut être explicitement utilisée pour créer une entrée dans un map

```
#include <utility>
...
pair<int,string> pGC (1,"Gilles Clemente");

pair<int,string> pCT;
p.first = 2;
p.second = "Cassia Trojahn";

pair<int,string> pMD;
pMD = make_pair(3,"Marie Dupont");
```

- La structure tuple est également disponible

```
tuple<int,char,int> t (10,'a',50);
cout << get<0>(t) << ' ' << get<1>(t) << ' ' << get<2>(t);
get<0>(t) = 2;
```

map

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

struct ordreInv { // comparison des valeurs cles
  bool operator() (const int& int1, const int& int2) const {
       return int1 > int2;
    }
};
 ...
map<int,string> id_book;
map<int,string>::iterator it;
pair<int,string> p1 (1,"Trojahn");
id_book.insert(p1); // l'ajout n'est pas garanti (verification de la cle)
cout << id_book[1] << endl;
id_book[2] = "Clemente";
for (auto it = id_book.begin(); it != id_book.end(); ++it)
     cout << " [" << it->first << ':' << it->second << ']';
cout << '\n';
map<int,string,ordreInv> id_book_compare; // speficie le type de comparaison (map<int,
     string, greater<int>>)
id_book_compare.insert(id_book.begin(),id_book.end());
id_book_compare.emplace(3,"Arnaud");
for (auto it = id_book_compare.begin(); it != id_book_compare.end(); ++it)
     cout << " [" << it->first << ':' << it->second << ']';
```

multimap

- Un multimap est un map dans lequel la duplication des clés est autorisée
- Cela signifie que, contrairement à un map, un multimap ne peut pas prendre en charge l'indiçage (opérateur[])
- Internement, les éléments d'un multimap sont toujours triés par sa clé selon un critère d'ordre spécifique
- Les opérations suivantes sont les principaux outils permettant d'accéder à plusieurs valeurs d'une même clé :
  - lower_bound(k) : renvoie un itérateur vers le premier élément de clé k
  - upper_bound(k) : renvoie un itérateur pointant vers le premier élément ayant une clé supérieur à k
  - equal_range(k) : renvoie les itérateurs lower_bound et upper_bound de clé k
- En d'autres termes, lower_bound(k) et upper_bound(k) permettent de fournir le début et la fin de la sous-séquence d'éléments de clé k (la fin est un itérateur sur l'élément suivant le dernier de la séquence)
- L'interface de multimap est très similaire à celle de map (à quelques exceptions près, comme par exemple : *insert(value_type)* (garantie) retourne un itérateur au lieu d'un pair, comme pour map)

multimap

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

using namespace std;

int main () {
   multimap<string,string> phone_book;
   phone_book.emplace("Trojahn","0645678900");
   phone_book.emplace("Arnaud", "0611111100");
   phone_book.emplace("Trojahn","0569090909");
   phone_book.emplace("Arnaud", "0589898989");

   typedef multimap<string,string>::iterator I;

   pair<I,I> bounds = phone_book.equal_range("Trojahn");

   for(I it=bounds.first; it != bounds.second;it++)
       cout << it->second << endl;
}
```

set

- Un set peut être considéré comme un map dans lequel les valeurs n'ont aucune importance
- Les clés sont les valeurs (la clé elle-même est la valeur)
- Les objets d'un set sont classés à partir d'une fonction de comparaison (<)
- Il ne peut pas y avoir plusieurs éléments avec la même clé
- L'interface de set est très similaire à celle des autres conteneurs associatifs

set

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

using namespace std;

int main () {
   set<int> si;
   si.insert(4);
   si.insert(1);
   si.insert(2);
   si.insert(3);
   si.insert(0);

   si.erase(0);
   set<int>::iterator it;
   cout << "Dans le set : " << (si.find(0) != si.end()) << endl;

   for(it=si.begin(); it != si.end(); it++) {
       cout << *it << endl;
   }
}
```

multiset

- Un multiset est un set dans lequel la duplication des clés est autorisée
- Comme pour un multimap, les opérations lower_bound(k), upper_bound(k) et equal_bound(k) permettent d'accéder à la sous-séquence de valeurs k

```
multiset<int> msi;
for (int x=100;x>0; x--) {
     msi.insert(x);
     msi.insert(x);
}

set<int>::iterator itlow,itup;
itlow = msi.lower_bound (50);
itup = msi.upper_bound (90);
msi.erase(itlow,itup);

for(set<int>::iterator it=msi.begin(); it != msi.end(); it++) {
    cout << *it << endl;
}
```

Conteneurs associatifs non triés

- Pour chacun des conteneurs associatifs, il existe 'une version' dans laquelle les éléments du conteneur sont organisés à l'aide de tables de hachage
  - unordered_map
  - unordered_multimap
  - unordered_set
  - unordered_multiset
- Cela permet un accès rapide à un élément particulier
- Cependant, ils sont généralement moins efficaces pour les itérations à travers un sous-ensemble de leurs éléments

Algorithmes de la STL

- La STL fournit les algorithmes les plus courants pour des manipulations sur les conteneurs
- Ces algorithmes s'expriment sous la forme d'une fonction ou d'un ensemble de fonctions modèles
  - opérations de séquence sans modification (for_each, find, count, etc.)
  - opérations de séquence avec modifications (replace, copy, remove, etc.)
  - opérations de tri et sur séquences triées (sort, merge, partition, etc.)
  - opérations ensemblistes (includes, set_union, set_intersection, etc.)
  - sélection selon une comparaison (min, max, min_element, max_element)

Opérations de séquence sans modification

- for_each : exécute une opération sur chaque élément d'une séquence
- find : recherche la première occurrence d'une valeur dans une séquence
- find_if : recherche la première correspondance d'un prédicat dans une séquence
- find_first_of : recherche une valeur provenant d'une séquence dans autre
- find_end : recherche la dernière occurrence d'un prédicat dans une séquence
- count : compte les occurrences d'une valeur dans une séquence
- count_if : compte les correspondances d'un prédicat dans une séquence
- equal : vrai si les éléments de deux séquences sont égaux au niveau des paires
- mismatch : recherche les premiers éléments pour lesquels deux séquence différent
- search_n : recherche la n^{eme} occurrence d'une valeur dans une séquence

Algorithmes de la STL : find_if, for_each, count_if

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

bool isOdd (int i) { return ((i%2)==1); }
void printDouble (int i) { cout << (i*2) << ' '; }

int main () {
   vector<int> vec;
   vec.push_back(10);
   vec.push_back(25);
   vec.push_back(40);
   vec.push_back(55);
   vector<int>::iterator it;
   // avec fonction nommee
   it = find_if (vec.begin(), vec.end(), isOdd);
   // avec fonction lambda
   it = find_if (vec.begin(), vec.end(), [](int i){ return ((i%2)==1); });
   cout << "The first odd value is " << *it << '\n';
   // avec fonction nommee
   for_each(vec.begin(),vec.end(),printDouble);
   // avec fonction lambda
   for_each(vec.begin(),vec.end(),[](int i) { cout << "i = " << (i*2) << ' ';});
   cout << endl;
   cout << "The number of odd values is " << count_if (vec.begin(),vec.end(),isOdd);
   return 0;
}
```

Opérations de séquence avec modification

- transform : applique une opération sur tous les éléments d'une séquence (transformation de l'entrée)
- copy : copie une séquence à partir de son premier élément
- copy_backward : copie une séquence à partir de son dernier élément
- copy_if : copie une séquence correspondant à un prédicat
- swap : échange deux éléments
- swap_ranges : échange des éléments de deux séquences
- replace : remplace les éléments par une valeur donnée
- replace_if : remplace les éléments correspondant à un prédicat
- generate : remplace tous les éléments par le résultat d'une opération
- remove : supprime les éléments ayant une valeur donnée
- remove_if : supprime les éléments correspondant à un prédicat – attention : en effet, remplace la valeur à supprimer par une valeur 'indéfinie' et les placent à la fin de la liste (normalement utilisé avec `erase` du conteneur en question)
- unique : supprime les éléments contigus égaux
- reverse : inverse l'ordre des éléments

Algorithmes de la STL : transform, replace_if, remove, generate

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

bool isOdd (int i) { return ((i%2)==1); }
void print(int i) { cout << i << ' '; }
int multi (int i) { return (i*3); }

int main() {
    vector<int> vec {1,2,4,5};
    //transform (vec.begin(),vec.end(), vec.begin(), multi);
    transform (vec.begin(),vec.end(), vec.begin(), [](int i){ return i*3;});
    //replace_if (vec.begin(), vec.end(), isOdd, 0);
    replace_if (vec.begin(), vec.end(), [](int i) {return ((i%2)==1);}, 0);
    //for_each(vec.begin(),vec.end(),print);
    for_each(vec.begin(),vec.end(),[](int i) { cout << "i = " << i << ' ';});
    cout << endl;
    remove(vec.begin(), vec.end(),12); // vector n'est pas realloue !
    for_each(vec.begin(),vec.end(),print);
    cout << endl;
    generate (vec.begin(), vec.end(), []() { return rand()%100; });
}
```

Opérations de tri et sur séquences triées

- sort : trie d'une séquence dans l'ordre croissant (operator< ou fonction comparaison)
- stable_sort : trie en conservant l'ordre des éléments égaux
- partial_sort : trie la première partie d'une séquence
- partial_sort_copy : copie en classant la première partie
- lower_bound : recherche la première occurrence d'une valeur
- upper_bound : recherche la dernière occurrence d'une valeur
- equal_range : recherche une sous-séquence d'une valeur donnée
- binary_search : vérifie s'une valeur se trouve dans une séquence triée
- merge : fusionne deux séquences triées
- partition : place en premier les éléments correspondant à un prédicat, tout en préservant l'ordre relatif

Algorithmes de la STL : sort et binary_search

```
class Entry {
      int id; string nom; string prenom; string phone;
      public :
          Entry(int id, string nom, string prenom, string phone);
          string getNom() const;
          string getPhone() const;
          friend bool operator<(Entry a, Entry b);
};
...
bool operator<(Entry a, Entry b) {
          return (a.id < b.id);
}
...
Entry e (3,"Trojahn", "Cassia","06xxxxxx");
Entry e1 (2,"Clemente","Gilles","06xxxxxx");
Entry e2 (1,"Arnaud","Marie","09xxxxxx");
vector<Entry> vec {e,e1,e2};

sort(vec.begin(),vec.end());

if (binary_search(vec.begin(),vec.end(),e))
    cout << "Found !";
else
    cout << "Not found !";
```

Algorithmes de la STL : list et sort

- Les classes forward_list et list ont leur fonctions membres *sort* (le type d'élément donné doit surcharger l'opérateur <)

```
class Entry {
      int id; string nom; string prenom; string phone;
      public :
          Entry(int id, string nom, string prenom, string phone);
          string getNom() const;
          friend bool operator<(Entry a, Entry b);
};
...
bool operator<(Entry a, Entry b) {
        if (a.id < b.id) return true; else return false;
}
int main () {
   list<Entry> phone_book;
   Entry e (3,"Trojahn", "Cassia","06xxxxxx");
   Entry e1 (2,"Clemente","Gilles","06xxxxxx");
   Entry e2 (1,"Clemente","Pierre","09xxxxxx");
   phone_book.push_front(e); // ajout au debut
   phone_book.push_back(e1); // ajout a la fin
   phone_book.push_back(e2); // ajout a la fin
   phone_book.sort(); // <==== fonction membre de list
   list<Entry>::iterator it;
   for (it=phone_book.begin(); it != phone_book.end();it++)
      cout << it->getNom()<< endl;
}
```

Références

- Stroustrup, Bjarne. “Le Langage C++”. Pearson Education, 2003 (1098 pages).
- Stroustrup, Bjarne. “What is C++0x” ? White paper, 2009.
- Standard C++ (https ://isocpp.org/), consulté février 2015.
- Stroustrup, Bjarne. FAQ [Bjarne’s web site] (http ://www.stroustrup.com/C++11FAQ.html), consulté février 2017.