Les tableaux de taille fixe


```
#include <array>
```

Déclaration : `array<type, taille> identificateur;`
Déclaration/Initialisation :

```
array<type, taille> identificateur =
        {val1, ... , valtaille};
```

Accès aux éléments : `tab[i]` `i` entre **0** et **taille-1**

Fonctions spécifiques :
`size_t tab.size()` : renvoie la taille

Tableau multidimentionnel :

```
array<array<type, nb_colonnes>, nb_lignes>
identificateur;
tab[i][j] = ...;
```


Jamila Sam
& Jean-Cédric Chappelier
ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Objets et Classes en C++

`class MaClasse { ... };` déclare une classe.
`MaClasse obj1;` déclare une instance (objet) de la classe `MaClasse`

Les attributs d'une classe se déclarent comme des champs d'une structure : `class MaClasse { ... type attribut; ... };`

Les méthodes d'une classe se déclarent comme des fonctions, mais dans la classe elle-même :

```
class MaClasse { ... type methode(type1 arg1, ...); ... };
```

Encapsulation et interface :

```
class MaClasse {
private: // attributs et methodes privees
...
public: // interface : attributs et methodes publiques
...
};
```

L'attribut particulier `this` est un pointeur sur l'instance courante de la classe. Exemple d'utilisation : `this->monattribut`

Construction/Destruction

Méthode **constructeur**(initialisation des attributs) :

```
NomClass(liste_arguments)
: attribut1(...), /* bloc optionel:
  ....              appels aux constructeurs
  attributN(...)    des attributs */
{
 // autres opérations
}
```

Méthode **constructeur de copie**:

```
NomClasse(const NomClasse& obj)
:...
{...}
```

Méthode **destructeur**(ne peut être surchargée) :

```
~NomClasse() {
  // opérations (de libération)
}
```

Des versions par défaut (minimales) de ces méthodes sont générées automatiquement par C++ si on ne les fournit pas

Règle: si on en définit une explicitement, il vaut mieux toutes les redéfinir !

Les structures de contrôle

les branchements conditionnels : *si ... alors ...*

```
if (condition)
    instructions
.....................
if (condition 1)
    instructions 1
...
else if (condition N)
    instructions N
else
    instructions N+1
```

```
switch (expression) {
   case valeur:
       instructions;
       break;
   ...
   default:
       instructions;
}
```

les boucles conditionnelles : *tant que ...*

```
while (condition)
    Instructions
```

```
do
    Instructions
while (condition);
```

les itérations : *pour ... allant de ... à ...*

```
for (initialisation ; condition ; increment)
    instructions
```

les sauts : `break;` et `continue;`

Note : `instructions` représente une instruction élémentaire ou un bloc.
`instructions;` représente une suite d'instructions élémentaires.

Espaces de nommage

Nommage d'un espace de noms :

```
namespace nom { ... corps de l'espace de noms
...}
```

Utilisation d'un/des objet(s) d'un espace de noms :

```
nom::objet;
using namespace nom;
using nom::objet;
```


Jamila Sam
ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Exceptions

`throw expression;` lance l'exception définie par l'expression

`try { ... }` introduit un bloc sensible aux exceptions

`catch (type& nom) { ... }` bloc de gestion de l'exception

Tout bloc `try` doit toujours être suivit d'un bloc `catch` gérant les exceptions pouvant être lancées dans ce bloc `try`.

Si une exception est lancée mais n'est pas interceptée par le `catch` correspondant, le programme s'arrête (« `Aborted` »).

Les fonctions

Prototype (à mettre **avant** toute utilisation de la fonction) :

```
type nom ( type1 arg1, ..., typeN argN [ = valN ] );
```

`type` est `void` si la fonction ne retourne aucune valeur.

Définition :

```
type nom ( type1 arg1, ..., typeN argN )
{
    corps
    return value;
}
```

Passage par **valeur** :

```
type f(type2 arg);
```

`arg` ne peut pas être modifié par `f`

Passage par **référence** :

```
type f(type2& arg);
```

`arg` peut **être modifié** par `f`

Surcharge (exemple) :

```
void affiche (int arg);
void affiche (double arg);
void affiche (int arg1, int arg2);
```


Jamila Sam
& Jean-Cédric Chappelier
ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Héritage

Spécifier un *lien d'héritage* :

```
class Sousclasse  : [public] SuperClass {...}
```

Droits d'accès : `protected` accès autorisé au sein de la hiérarchie

Masquage : un attribut/méthode peut être redéfini dans une sous-clas

Accès à un *membre caché* : `SuperClasse::membre`

Le constructeur d'une sous classe doit faire appel au *constructeur de la super classe* :

```
class SousClasse: SuperClasse
{
   SousClasse(liste de paramètres)
   : SuperClasse(Arguments),
    attribut1(valeur1), ..., attributN(valeurN) {...}
};
```

Héritage multiple


```
class nomSousClasse: [public] nomSuperClasse1, ...
                     [public] nomSuperClasseN
```

Collision de noms d'attributs/méthodes : c'est la sous-classe qui hérite de ces attributs/méthodes qui doit définir *le sens de leur utilisation*

Classe virtuelle : pour éviter qu'une sous-classe *hérite plusieurs fois d'une même super-classe*, il faut déclarer les dérivations concernées comme **virtuelles**

```
NomSousClasse: [public] virtual NomSuperClasseVirtuelle
```

Constructeur :

```
SousClasse(liste de parametres)
  : SuperClasse1(arguments1),
    ...
    SuperClasseN(argumentsN),
    attribut1(valeur1)
    ...
    attributK(valeurK)
{}
```

Les entrées/sorties

Clavier / Terminal : `cin` / `cout` et `cerr`

Fichier de définitions : `#include <iostream>`

Utilisation :

écriture : `cout << expr1 << expr2 << ...;`

lecture : `cin >> var1 >> var2 >> ...;`

Saut à la ligne : `endl`

Lecture d'une ligne entière : `getline(cin, string);`

. . . . . .

Formatage :

Manipulateurs	
`#include <iomanip>` `cout << manip << expr << ...`	
`dec`, `oct`, `hex`	changement de base
`setprecision(int)`	nombre de chiffres à afficher
`setw(int)`	largeur de colonne nombre de caractères à lire
`setfill(char)`	caractère utilisé dans l'alignement
`cin >> ws`	saute les blancs

Options	
`setf(ios::option)` `unsetf(ios::option)`	
`ios::left`	alignement à gauche
`ios::showbase`	afficher la base
`ios::showpoint`	afficher toujours la virgule
`ios::fixed`	notation fixe
`ios::scientific`	notation scientifique

Les entrées/sorties (2)

Fichiers :

Fichier de définitions : `#include <fstream>`

Flot d'**entrée** (similaire à `cin`) : `ifstream`
Flot de **sortie** (similaire à `cout`) : `ofstream`

Création : `type_flot nom_de_flot;`

Lien (ouverture) : `flot.open("fichier");`
ouverture en binaire :
- pour lecture : `ifstream flot("fichier", ios::in|ios::binary);`
- pour écriture : `ofstream flot("fichier", ios::out|ios::binary);`

Utilisation : comme `cin` et `cout` :

```
flot << expression << ...;
flot >> variable_lue >> ...;
```

Test de fin de fichier : `flot.eof()`

Fermeture du fichier : `flot.close()`

Test d'échec de lecture/écriture/ouverture sur le flot : `flot.fail()`

Opérateurs

Operateurs arithmétiques

`*`	multiplication	
`/`	division	
`%`	modulo	
`+`	addition	
`-`	soustraction	
`++`	incrément	*(1 opérande)*
`--`	décrément	*(1 opérande)*

Operateurs de comparaison

`==`	teste l'égalité logique
`!=`	non égalité
`<`	inférieur
`>`	supérieur
`<=`	inférieur ou égal
`>=`	supérieur ou égal

Operateurs logiques

`&&`	"et" logique	
`||`	ou	
`^`	ou exclusif	
`!`	négation	*(1 opérande)*

Priorités (par ordre décroissant, tous les opérateurs d'un même groupe sont de priorité égale) :

`!` `++` `--`, `*` `/` `%`, `+` `-`, `<` `<=` `>` `>=`, `==` `!=`, `^` `&&`, `||`

J. Sam

Pointeurs & références

Déclaration : `type* pointeur;`

Déclaration/Initialisation :

```
type* pointeur(adresse);
unique_ptr<type>(new type(valeur));
type& reference(objet);
```

Adresse d'une variable : `&variable`

Accès au contenu pointé par un pointeur : `*pointeur`

Allocation mémoire :

```
pointeur = new type;
pointeur = new type(valeur);
```

Libération de la zone mémoire allouée :

`delete pointeur` (pour les « pointeurs classiques », obligatoire)

Jamila Sam
ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE


Pointeurs (avancés)


Pointeur sur une constante : `type const* ptr;`

Pointeur constant : `type* const ptr(adresse);`

Pointeur sur une fonction :

```
type_retour (*ptrfct)(arguments...)
```

```
function<type_retour(arguments...)> ptrfct
```


Jamila Sam
ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Polymorphisme

Résolution dynamique des liens : choix des méthodes à invoquer **lors de l'exécution du programme** en fonction de la **nature réelle des instances**

2 ingrédients :

méthodes virtuelles et **références**/**pointeurs**

Méthode virtuelle :

```
virtual Type nom_fonction(liste d'arguments)
[const];
```

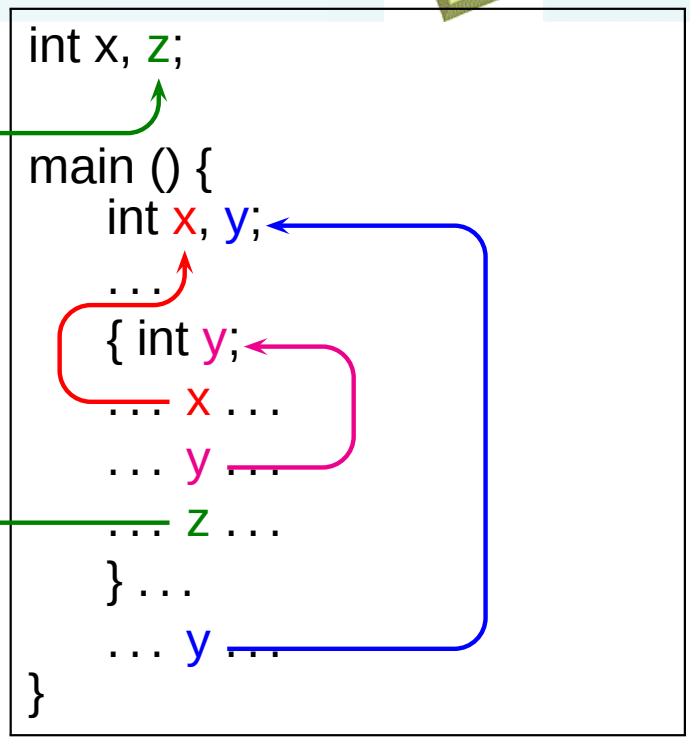
Méthode virtuelle *pure* (abstraite) :

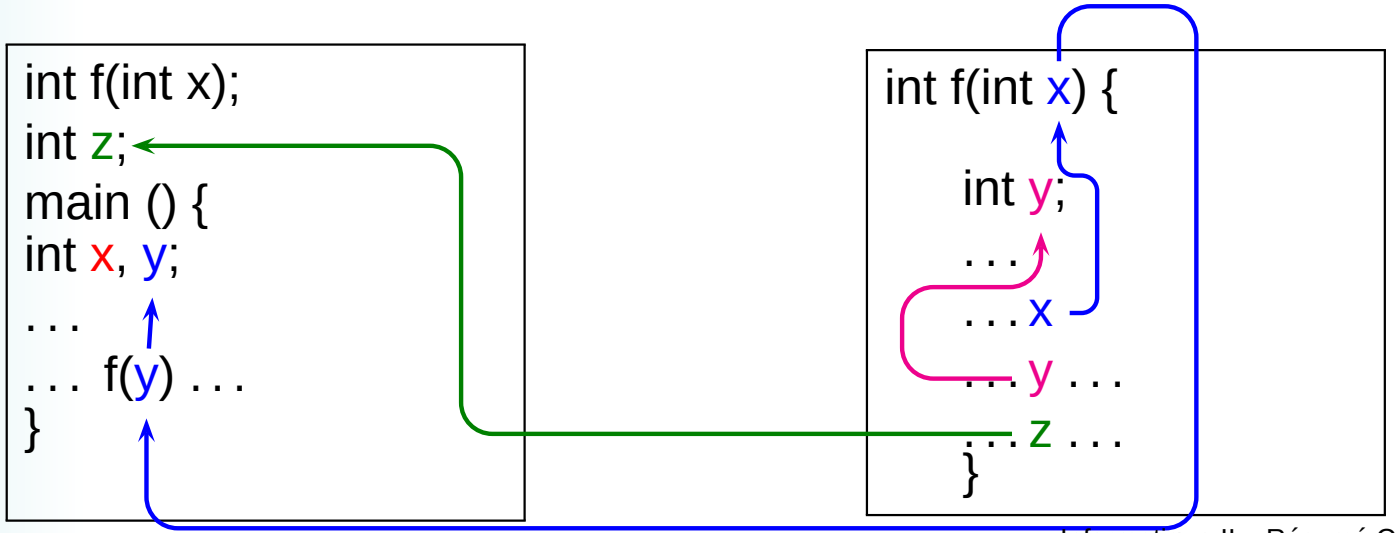
```
virtual Type nom_methode(liste d'arguments)
const =0
```

Classe abstraite : contient *au moins une méthode abstraite*

Collection hétérogène : des pointeurs sur les instances doivent être manipulés, et non pas les instances directement

Portée / Appel





Les chaînes de caractères


```
#include <string>
```

déclaration/initialisation : `string identificateur("valeur");`

Affectation :
```
chaine1 = chaine2;
chaine1 = "valeur";
chaine1 = 'c';
```

Concaténation :
```
chaine1 = chaine2 + chaine3;
chaine1 = chaine2 + "valeur";
chaine1 = chaine2 + 'c';
```

Accès au (`i`+1)-ème caractère : `chaine[i];`

Fonctions spécifiques :

taille : `chaine.size()`

insertion : `chaine.insert(position, chaine2)`

replacement : `chaine.replace(position, longueur, chaine2)`

suppression : `chaine.replace(position, longueur, "")`

sous-chaîne : `chaine.substr(position, longueur)`

recherche :
```
chaine.find(souschaine)
chaine.rfind(souschaine)
```

valeur "pas trouvé" d'une recherche : `string::npos`

Les structures

Déclaration du type correspondant :

```
struct Nom_du_type {
        type1 champ1 ;
        type2 champ2 ;
        ...
};
```

Déclaration d'une variable :

```
Nom_du_type identificateur;
```

Déclaration/Initialisation d'une variable :

```
Nom_du_type identificateur = { val1, val2, ...};
```

Accès à un champ donné de la structure :

```
identificateur.champ
```

Affectation globale de structures :

```
identificateur1 = identificateur2
```


Jamila Sam
& Jean-Cédric Chappelier
ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Surcharge d'opérateurs


```
class Classe {
   ...
   type_retour operatorOp(type_argument); // prototype de l'operateur Op
   ...
};

// définition de l'operateur Op
type_retour Classe::operatorOp(type_argument) { ... }

// operateur externe
type_retour operatorOp(type_argument, Classe&) { ... }
```

Quelques exemple de prototypes :

```
bool operator==(Classe const&) const;  // ex: p == q
bool operator<(Classe const&) const;   // ex: p < q
Classe& operator=(Classe const&);      // ex: p = q
Classe& operator+=(Classe const&);     // ex: p += q
Classe& operator++();                  // ex: ++p
Classe& operator*=(const autre_type);  // ex: p *= x;
Classe operator-(Classe const&) const; // ex: r = p - q
Classe operator-() const;              // ex: q = -p;

// operateurs externes
ostream& operator<<(ostream&, Classe const&);
Classe operator*(double, Classe const&);
```

Templates

Déclarer un modèle de classe ou de fonction :

```
template<typename nom1, typename nom2, ...>
```

Définition externe des méthodes de modèles de classes :

```
template<typename nom1, typename nom2, ...>
NomClasse<nom1, nom2, ...>::NomMethode(...
```

Instanciation : spécifier simplement les types voulus après le nom de la classe/fonction, entre `<>` (Exemple : `vector<double>`)

Spécialisation (totale) de modèle pour les types *type1*, *type2*... :

```
template<> NomModele<type1,type2,...> ...suite
de la declaration...
```

Compilation séparée : pour les templates, il faut tout mettre (déclarations et définitions) dans le fichier d'en-tête (`.h`).

Jamila Sam
ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Variables

En C++, une **valeur** à conserver est stockée dans une variable caractérisée par :

- son **type**
- et son **identificateur** ;

(définis lors de la **déclaration**)

La *valeur* peut être définie une première fois lors de l'**initialisation**, puis éventuellement modifiée par la suite.

Rappels de syntaxe :

```
type nom;                (déclaration)
type nom(valeur);        (initialisation)

nom = expression;        (affectation)
```

Types élémentaires :

```
int
double
char
bool
```

Exemples :

```
int val(2);
const double z(x+2.0*y);
constexpr double pi(3.141592653);
i = j + 3;
```

Le template vector


```
#include <vector>
```

Déclaration : `vector< type > identificateur;`
Déclaration/Initialisation :

```
vector< type > identificateur(taille);
```

Accès au (`i`+1)-ème élément : `tab[i];`

Fonctions spécifiques :
`int tab.size()` : renvoie la taille
`bool tab.empty()` : détermine s'il est vide ou non
`void tab.clear()` : supprime tous les éléments
`void tab.pop_back()` : supprime le dernier élément
`void tab.push_back(valeur)` : ajoute un nouvel élément à la fin

Jamila Sam
& Jean-Cédric Chappelier
ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE