Programmation Orientée Objet :
Gestion des erreurs
et
Divers compléments

Jamila Sam

Laboratoire d'Intelligence Artificielle
Faculté I&C

EPFL

Les acquis du premier semestre

Programmer c'est décomposer une **tâche** à automatiser en une séquence d'instructions (**traitements**) et des **données**

Traitements	Données
Algorithmes	S.D.A.
Expressions & Opérateurs Structures de contrôle Fonctions	Variables Portée Chaînes de caractères Tableaux de taille fixe Tableaux dynamiques Structures Pointeurs
Entrées/Sorties	

Jamila Sam
& Jean-Cédric Chappelier

Objectifs du second semestre

1. Apprendre à **programmer** de façon plus concise et modulaire mais aussi plus fiable et élégante en exploitant les concepts orientés-objets
   - ☞ au moyen du langage C++
2. Approfondir quelques notions de structuration des données (algorithmique, généricité)

Présentation générale du cours

Public : Cours obligatoire pour les étudiants de 2ème semestre de la section des Sciences de la Vie. Connaissances supposées acquises : bases de la programmation procédurale en C++

Langue : Français

Moyens :

Concepts théoriques introduits ou complémentés lors de **cours** magistraux ex-cathedra
(Mardi $13^{15}/14^{15}$–15^{00})

mis en pratique, de manière guidée, lors de **séances d'exercices** sur machines
(Mardi 10^{15}–13^{00})

Compléments en lignes : vidéos et quizzes (disponibles pour les 8 premières semaines).

Jamila Sam
& Jean-Cédric Chappelier

Couplage au MOOC (1)

MOOC d'introduction à la programmation orientée-objets en C++ :

www.coursera.org/learn/programmation-orientee-objet-cpp/

A utiliser comme au semestre passé

Matériel MOOC utilisé :

1. Vidéos
2. Quizzes
3. Devoirs (mais ne comptent pas)
   - ☞ à considérer comme des exercices supplémentaires

Couplage au MOOC (2)

- Avant le cours : visionner les vidéos, faire les quizzes et comprendre certains exercices de niveau 0
- Compléments de cours : résumé et approfondissements
  - ☞ seulement une heure en direct ou pré-enregistrés
- Exercices/projet : mise en pratique

Couplage au MOOC (3)

Charge de travail :

- 1 heure de compléments de cours : récapitulation et approfondissements ;
- 3 heures d'exercices en salle de TP : mise en pratique ;
- 6 heures de travail à la maison :
  - 1 :30-1 :45 sur les vidéos de la semaine suivante
  - 0 :15-0 :30 sur les quizzes de la semaine suivante
  - 4 heures pour commencer à préparer la série d'exercices de la semaine en cours, finaliser celle de la semaine passée ou programmer le projet noté.

Notes et examens

Les épreuves de contrôle continu seront les suivantes :

- Examen théorique **individuel**, 2h
- Projet **en binômes**, environ 8 semaines

Calcul de la note

- La note finale, N, est calculée comme suit :

$$N = N_{Examen} * 0.4 + N_{projet} * 0.6$$

- Les notes intermédiaires ne sont pas arrondies.
- Les cours ICC et Programmation Orientée Objet sont indépendants pour ce qui est de la note. La moyenne arrondie de chaque cours est transmise au SAC à la fin de chaque semestre.

Notes et examens
Examen

Le semestre sera clôturé par un examen écrit portant sur le contenu du cours et les séances d'exercices.

Date :

Mardi 20 Mai

Notes et examens
« Défense » du projet

La dernière séance de TP sera consacrée aux défenses de projets (pas vraiment de préparation nécessaire si vous avez suivi l'échéancier proposé)

Date :

Mardi 27 Mai

Vous présenterez :

- ce que vous avez fait (petite démonstration)
- comment vous avez procédé : choix, méthodes, organisation
- quelles difficultés vous avez rencontrées et comment vous les avez traitées
- en conclusion : ce que vous avez retenu du projet.

Erreurs en programmation

Il existe plusieurs types d'erreurs :

1. erreurs de syntaxe : le programme est mal écrit et le compilateur ne comprend pas ce qui est écrit.
   Erreurs relativement faciles à trouver : le compilateur signale le problème, indiquant souvent l'endroit de l'erreur.
2. erreurs d'implémentation : la syntaxe du programme est correcte (il compile), mais ce que fait le programme est erroné (par exemple une division par zéro se produit, ou une variable n'a pas été initialisée correctement).
   Ces erreurs ne se détectent qu'à l'exécution du programme, soit par un arrêt prématuré (e.g. cas de la division par zéro), soit par des résultats erronés (e.g. cas de la mauvaise initialisation).

Jamila Sam
& Jean-Cédric Chappelier

Erreurs en programmation

Il existe plusieurs types d'erreurs :

③ erreurs d'algorithme : l'algorithme implémenté ne fait pas ce que l'on croit (ce qu'il devrait)
assez proche du cas précédent. Mais ici, c'est plus la méthode globale qui est erronée, plutôt qu'une étourderie ou un manque de précision dans une des étapes du codage de l'algorithme.
Il existe pour ce type d'erreurs des tests formels permettant de trouver les erreurs.
Mais ce genre de techniques est trop complexe pour être abordé dans ce cours.

④ erreurs de conception : ici c'est carrément l'approche du problème qui est erronée, souvent en raison d'hypothèses trop fortes ou non explicitées.
Elles relèvent du domaine de l'ingénierie informatique (le « génie logiciel »), et ne seront pas traitées dans ce cours.

Erreurs en programmation

Il existe plusieurs types d'erreurs :

① erreurs de syntaxe
② erreurs d'implémentation
③ erreurs d'algorithme
④ erreurs de conception

Pour traiter les erreurs d'implémentation (②) et d'algorithme (③), d'un point de vue pratique :
c'est-à-dire mise en œuvre de **procédures de déverminage**.

Comment trouver la sources des erreurs **lors de l'exécution** du programme ?

Dévermineur

L'utilisation d'un « dévermineur » (« debugger » en anglais) permet d'ausculter en détail l'exécution d'un programme, et en particulier

- localiser les erreurs
- exécuter un programme pas à pas
- suivre la valeur de certaines variables

☞ suivre les tutoriels donné en début de la série de la semaine prochaine

En C++

① Pour pouvoir utiliser un dévermineur, il faut **compiler avec l'option** `-g`
Cela indique au compilateur de rajouter des informations supplémentaires dans le programme, utiles au dévermineur.

```
c++ -g -o monprogramme monprogramme.cc
```

② Il faut ensuite exécuter le programme à corriger/étudier dans le dévermineur (`ddd` sur les VMs de l'EPFL ou celui intégré à l'outil QtCreator).

Jamila Sam
& Jean-Cédric Chappelier

Dévérmineurs : fonctionnalités typiques

③ On peut décider de suspendre l'exécution du programme à des endroits précis en y plaçant des breakpoints (points d'arrêt)

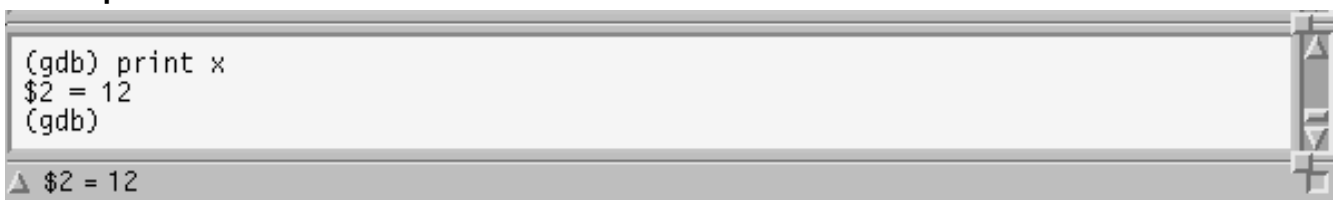
④ Une fois le programme stoppé à un point d'arrêt, on peut continuer à l'exécuter

- soit *pas à pas* avec la commande `next` qui exécute les pas de programme au **même niveau** que le point d'arrêt (mais ne « descend » pas dans les fonctions appelées)
- soit *pas à pas* avec la commande `step` qui exécute les pas élémentaires de programme et donc **entre dans les fonctions appelées**
- soit *en continu* jusqu'au prochain point d'arrêt avec la commande `cont`

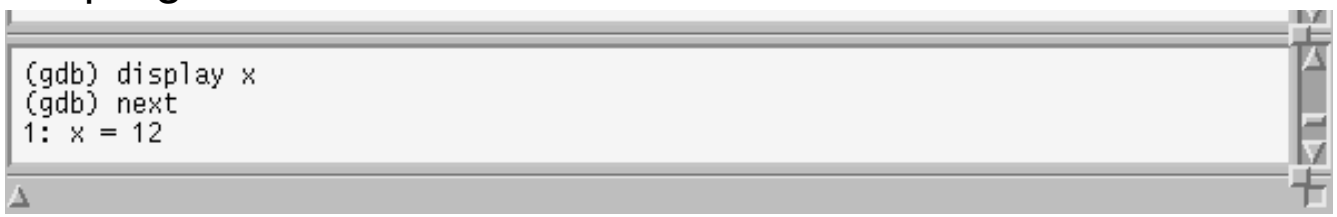
Dévérmineurs : fonctionnalités typiques (2)

⑥ On peut regarder le contenu d'une variable

- soit en mettant la souris dessus
- soit à l'aide de la commande `print`
  qui affiche la valeur de la variable à ce moment là



- soit à l'aide de la commande `display`.
  La valeur de la variable est alors affichée à chaque pas de programme.



Jamila Sam
& Jean-Cédric Chappelier

Gestion des erreurs

Les **exceptions** permettent d'anticiper les erreurs qui pourront potentiellement se produire lors de l'utilisation d'une portion de code.

Exemple : on veut écrire une fonction qui calcule l'inverse d'un nombre réel quand c'est possible :

f
entrée : x sortie : $1/x$
Si $x = 0$ *erreur* ***Sinon*** *retourner 1/x*

mais que faire **concrètement** en cas d'erreur ?

Gestion des erreurs (2)

① retourner une valeur choisie à l'avance :

```
double f(double x) {
    if (x != 0.0) return 1.0 / x;
    else
        return numeric_limits<double>().max();
}
```

Mais cela

1. **n'indique pas** à l'utilisateur potentiel qu'il a fait une erreur
2. retourne de toutes façons un **résultat inexact** ...
3. suppose une **convention arbitraire** (la valeur à retourner en cas d'erreur)

Gestion des erreurs (3)

② afficher un message d'erreur
mais que retourner effectivement en cas d'erreur ?...
on retombe en partie sur le cas précédent

```
double f(double x) {
  if (x != 0.0) return 1.0 / x;
  else {
    cerr << "Erreur d'utilisation de f :"
       << "division par 0"
       << endl;
    return numeric_limits<double>().max();
  }
}
```

De plus, cela est **très mauvais** car cela produit de gros effets de bord : modifie `cerr` alors que ce n'est pas du tout dans le rôle de `f`

Pensez par exemple au cas où l'on veut utiliser `f` dans un programme avec une interface graphique... on ne veut alors plus utiliser `cerr` (mais plutôt ouvrir une fenêtre d'alerte par exemple)

Gestion des erreurs (4)

③ retourner un code d'erreur :

```
int f(double x, double& resultat) {
  if (x != 0.0) {
    resultat = 1.0 / x;
    return PAS_D_ERREUR;
  }
  else return ERREUR_DIV_ZERO;
}
// PAS_D_ERREUR, ERREUR_DIV_ZERO :
// constantes definies plus haut
```

Cette solution est déjà **beaucoup mieux** car elle laisse à la fonction qui appelle `f` le soin de décider quoi faire en cas d'erreur.
Cela présente néanmoins l'inconvénient d'être assez lourd à gérer pour finir :

- cas de l'appel d'appel d'appel.... ...d'appel de fonction,
- mais aussi écriture peu intuitive :

  ```
  if (f(x,y) == PAS_D_ERREUR) //...
  ```

  au lieu de

  ```
  y=f(x);
  ```

Exceptions

Il existe une solution permettant de généraliser et d'assouplir cette dernière solution : déclencher une **exception**

☞ mécanisme permettant de prévoir une erreur à un endroit et de **la gérer à un autre endroit**

Principe :

- lorsque qu'une erreur a été détectée à un endroit, on la signale en « lançant » un objet contenant toutes les informations que l'on souhaite donner sur l'erreur
  (« lancer » = créer un objet disponible pour le reste du programme)
- à l'endroit où l'on souhaite gérer l'erreur (au moins partiellement), on peut « attraper » l'objet « lancé »
  (« attraper » = utiliser)
- si un objet « lancé » n'est pas attrapé du tout, cela provoque l'arrêt du programme : toute erreur non gérée provoque l'arrêt.

Un tel mécanisme s'appelle une exception.

Exceptions (2)

Avantages de la gestion des exceptions par rapport aux codes d'erreurs retournés par des fonctions :

- écriture plus facile, plus intuitive et plus lisible
- la propagation de l'exception aux niveaux supérieurs d'appel (fonction appelant une fonction appelant ...) est fait **automatiquement**

  plus besoin de gérer obligatoirement l'erreur au niveau de la fonction appelante
- une erreur peut donc se produire à n'importe quel niveau d'appel, elle sera toujours reportée par le mécanisme de gestion des exceptions

(**Note :** si une erreur peut être gérée localement, alors il faut le faire localement et ne pas utiliser le mécanisme des exceptions.)

Syntaxe de la gestion des exceptions

On cherche à remplir 3 tâches élémentaires :

1. signaler une erreur
2. marquer les endroits réceptifs aux erreurs
3. leur associer (à chaque endroit réceptif aux erreurs) un moyen de gérer leurs erreurs

On a donc 3 mots-clés du langage C++ dédiés à la gestion des exceptions :

`throw` indique l'erreur (i.e. « lance » l'exception)

`try` indique un bloc réceptif aux erreurs

`catch` gère les erreurs associées (i.e. les « attrape »)

Notez bien que :

- L'indication des erreurs (`throw`) et leur gestion (`try`/`catch`) sont le plus souvent à des endroits bien séparés dans le code
- Chaque bloc `try` possède son/ses `catch` associé(s)

throw

`throw` est l'instruction qui **signale l'erreur** au reste du programme.

Syntaxe : `throw expression`

l'expression peut être de tout type : c'est le résultat de son évaluation qui est « lancé » au reste du programme pour être « attrapé »

```
throw 21; // "lance" un entier
// "lance" une string:
throw string("quelle erreur !");
struct Erreur {
  int code;
  string message;
};
//...
Erreur faute;
//...
faute.code = 12; faute.message = "Division par 0";
throw faute;
```

throw (2)

`throw`, en « lançant » une exception, interrompt le cours normal d'exécution et :

- saute au bloc `catch` du bloc `try` directement supérieur (dans la pile des appels), si il existe ;
- quitte le programme (« abort ») si l'exécution courante n'était pas dans au moins un bloc `try`.

Exemple :

```
try {
    ...
    // appel contenant un throw int
    ...
}
catch (int i) {
    ...
}
```

En cas d'erreur, saute ici

En cas d'erreur, ce code n'est pas exécuté

try

`try` (*lit.* « essaye ») introduit un **bloc réceptif aux exceptions** lancées par des instructions, ou des fonctions appelées à l'intérieur de ce bloc (ou même des fonctions appelées par des fonctions appelées par des fonctions... ... ... à l'intérieur de ce bloc)

Exemple 1 :

```
try {
   ...
   if (x == 0.0) throw string("valeur nulle");
   //...
}
```

Exemple 2 :

```
try {
      //  ...
   y = f(x); // f pouvant lancer une exception
      //  ...
}
```

catch

`catch` est le mot-clé introduisant un **bloc dédié à la gestion** d'une ou plusieurs **exceptions**.

Tout bloc `try` doit toujours être suivi d'au moins 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` »).

Syntaxe :

```
catch (type nom) {
  //...
}
```

intercepte toutes les exceptions de type `type` lancées depuis le bloc `try` précédent

Exemple d'utilisation de `catch`

```
try {
   ...
   if (x == 0.0) throw string("valeur nulle");
   ...
     if (j >= 3) throw j;
}
// capture les exceptions lancees sous forme de string
catch(string const& erreur) {
   cerr << "Erreur : " << erreur << endl;
}
// capture les exceptions lancees sous forme d'int
catch(int erreur) {
   cerr << "Avertissement : je n'aurais pas du avoir"
        << " la valeur "
        << erreur
        << endl;
}
```

`catch` (flot d'exécution 1/3)

Un bloc `catch` n'est exécuté **que** si une exception de type correspondant a été lancée depuis le bloc `try` correspondant.

Sinon le bloc `catch` est simplement ignoré.

Si un bloc `catch` est exécuté, le déroulement continue ensuite normalement **après** ce bloc `catch` (ou après le dernier des blocs `catch` du même bloc `try` lorsqu'il y en a plusieurs).

En aucun cas l'exécution ne reprend après le `throw` !

`catch` (flot d'exécution 2/3)

Exemple :
en cas d'erreur (lancement d'une exception) :

```
try {
    ...
    // appel contenant un throw int
    ...
}
catch (int i) {
    ...
    ...
    ...
}
...
```

En cas d'erreur, ce code n'est pas exécuté

En cas d'erreur, saute ici

puis on continue ensuite ici.

`catch` (flot d'exécution 3/3)

Exemple :
si il n'y a pas d'erreur (**pas** de lancement d'exception) :

```
try {
    ...
    // appel contenant un throw int
    ...
}
catch (int i) {
    ...
    ...
    ...
}
...
```

S'il n'y a *pas* d'erreur, ce code **est** exécuté...

... puis on saute ici

`catch` (Remarques)

Notes :

- « `catch(...)` » permet d'intercepter n'importe quel type d'exceptions
  mais, dans le cas où il y a plusieurs `catch` associés à un même `try`, « `catch(...)` » doit être le dernier.
-  comme pour les fonctions, on préférera passer les exceptions de type complexe par *références constantes* :
  `catch (Erreur const& e)`

« Relancement »

Une exception peut être **partiellement traitée** par un bloc `catch` et attendre un traitement plus complet ultérieur (c'est-à-dire à un niveau supérieur).

Il suffit pour cela de **« relancer » l'exception** au niveau du bloc n'effectuant que le traitement partiel.
(Il faudra bien sûr pour cela que l'appel à ce bloc `catch` soit lui-même dans un autre bloc `try` à un niveau supérieur).

Pour « relancer » une exception, il suffit simplement d'écrire `throw` (i.e. sans argument)

Exemple :

```
catch (int erreur) {
// traitement partiel :
  cerr << "Hmm... pour l'instant je ne sais pas trop "
       << "quoi faire" << endl
       << "avec l'erreur " << erreur << endl;
// relance l'exception capt'ee:
  throw;
}
```

Exemple (1/4)

```
#include <iostream>
#include "mesures.h"
#include "acquisition.h"
#include "plot.h"
using namespace std;

void plot_temp_inverse(Mesures const&);
double inverse(double);

int main() {
  Mesures temperatures;
  acquerir_temp(temperatures);
  plot_temp_inverse(temperatures);
  return 0;
}

void plot_temp_inverse(Mesures const& t) {
  for (unsigned int i(0); i < t.size(); ++i) {
    plot(inverse(t[i]));
  }
}

double inverse(double x) {
  return 1.0/x;
}
```

Exemple (2/4)

```
...
using namespace std;

const int DIVZERO(33);

void plot_temp_inverse(Mesures const&);
double inverse(double);

int main() {
  Mesures temperatures;
  acquerir_temp(temperatures);
  plot_temp_inverse(temperatures);
  return 0;
}

void plot_temp_inverse(Mesures const& t) {
  for (unsigned int i(0); i < t.size(); ++i){
    plot(inverse(t[i]));
  }
}

double inverse(double x) {
  if (x == 0.0) throw DIVZERO;
  return 1.0/x;
}
```

Exemple (3/4)

```
...
int main() {
  Mesures temperatures;
  acquerir_temp(temperatures);
  try {
    plot_temp_inverse(temperatures);
  }
  catch (int i) {
    if (i == DIVZERO) {
      cerr << "Courbe des températures erronée" <<endl;

      /* on fait quelque chose, par exemple refaire
       * les mesures, mais à ce stade le programme
       * n'est pas stoppé.
       */
    }
  }
  return 0;
}
...
```


Jamila Sam
& Jean-Cédric Chappelier

Exemple (4/4)

```
int main() {
  ...
  try {
    plot_temp_inverse(temperatures);
  }
  catch (int i) {
    if (i == DIVZERO) {
      cerr << "Courbe des températures erronée" <<endl;

      // effectue ici un traitement de plus haut niveau
      ...
    }
...
void plot_temp_inverse(Mesures const& t) {
  for (unsigned int i(0); i < t.size(); ++i) {
    try {
      plot(inverse(t[i]));
    }
    catch (int j) {
      /* Traiter partiellement le problème et relancer l'exception.
       * Cette partie du programme peut par exemple signaler
       * l'indice de la valeur erronée.
       */
      cerr << "problème avec la valeur " << i << endl;
      throw;
    }
  }
}
```

Exemple complet avec reprise (1/3)

```
#include <iostream>
#include "mesures.h"
#include "acquisition.h"
#include "plot.h"
using namespace std;

const int DIVZERO(33);

void plot_temp_inverse(Mesures const&);
double inverse(double);

int main()
{
  Mesures temperatures;
  unsigned int const MAX_ESSAIS(2);
  unsigned int nb_essais(0);
  bool restart(false);

  do {
    ++nb_essais; restart=false;
    acquerir_temp(temperatures);
    try {
      plot_temp_inverse(temperatures);
    }
```

Exemple complet avec reprise (2/3)

```
    catch (int i) {
      if (i == DIVZERO) {
        if (nb_essais < MAX_ESSAIS) {
          cout << "Il faut re-saisir les valeurs" << endl;
          restart = true;
        } else {
          cout << "Il y a déjà eu au moins " << MAX_ESSAIS
               << " essais." << endl;
          cout << " -> abandon" << endl;
        }
      } else {
        cout << "Ne sais pas quoi faire -> abandon" << endl;
      }
    }
  } while (restart);

  return 0;
}

void plot_temp_inverse(Mesures const& t)
{
  for (unsigned int i(0); i < t.size(); ++i) {
```

Exemple complet avec reprise (3/3)

```
    // Exemple de traitement local partiel du problème
    // (ce n'est pas obligatoire).
    try {
      plot(inverse(t[i]));
    }
    catch (int j) {
      cerr << "Erreur : ";
      if (j == DIVZERO) {
        cerr << "la valeur " << i << " est nulle.";
      } else {
        cerr << "problème avec la valeur " << i;
      }
      cerr << endl;
      throw;
    }
  }
}

double inverse(double x)
{
  if (x == 0.0) throw DIVZERO;
  return 1.0/x;
}
```

Exception lancée par `new`

`new` (allocation dynamique de pointeur), retourne une exception de type `bad_alloc` (défini dans la bibliothèque « `new` ») si l'allocation dynamique ne se passe pas correctement.

Il est donc conseillé d'écrire par exemple :

```
#include <new>
try {
   ...
   ptr = new ...;
   ...
     }
catch (std::bad_alloc const& e) {
   cerr << "Erreur : plus assez de memoire !" << endl;
   exit 1;
}
```

C++11 Spécification des exceptions

Il est toujours bon en programmation d'**être** le plus **explicite** possible, en particulier sur ce que fait chaque chaque fonction/méthode.

Dans cet esprit, *dans un contexte où l'on prévoit d'introduire/de gérer des exception*, il est utile d'indiquer les fonctions/méthodes qui ne lancent pas d'exception.

Cela se fait au niveau de leur prototype en ajoutant

`noexcept`

derrière le **prototype** de la fonction.

Cela indique simplement que la fonction ne peut pas lancer d'exception (et si elle le fait, le programme se termine en fait immédiatement (par un appel à la fonction `terminate()`).

Exemple :

```
double f(double) noexcept;
```

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` »).

Espaces de noms

(Rappel) **Portée** d'un objet = région du programme où l'objet peut être utilisé

Exemples de portées : un bloc, le corps de fonction, tout le programme (variable globale), ...

☞ Qu'en est-il en cas de compilation séparée ?

Les portées locales restent inchangées (puisqu'elles sont « locales » par définition !)

Un **espace de noms** est justement un moyen de faire un regroupement logique de divers objets (variables, fonctions, ...)

Cela permet de partager des objets tout en **évitant les conflits** au niveau des noms...

...et donc de distinguer clairement deux objets portant le même nom, mais n'étant pas dans le même « espace de noms »

Espaces de noms (2)

Un espace de noms est simplement le nom donné à une portée : c'est l'espace regroupant tous les noms des objets dans cette portée.

On distingue :

- l'espace de noms global (qui a un nom vide) :
  c'est celui qui regroupe tous les objets déclaré en dehors de tout autre espace de noms
  les **variables globales** appartiennent par exemple à cet espace de noms
- les espaces de noms explicitement nommés
- les espaces de noms non nommés (ils n'ont pas de nom, même pas un nom vide !)
  Par exemple les blocs dans votre code (par exemple sous un `if`)

Jamila Sam
& Jean-Cédric Chappelier

Définition d'un espace de noms nommé

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

Exemple :

```
namespace outils {
    int compteur;
    double moyenne;
    int fonction(double x);
}
```

Note : un espace de noms **n'**est **pas** une structure, un type ou un objet quelconque, c'est juste un **nom de regroupement**, une « *étiquette* ».

L'objet `compteur` existe (il est déclaré) et sa **portée** s'appelle `outils`, mais `outils` n'est pas un objet manipulable en soi. C'est **juste un nom**.

Utilisation des objets appartenant à un espace de noms nommé

Pour référencer explicitement un objet `X` d'un espace de noms `nom`, on écrit : `nom::X`.

Exemple :
```
++(outils::compteur);
```

Si l'on veut utiliser plus librement tous les noms d'un espace de noms :

```
using namespace nom;
```

Exemple :
```
using namespace outils;
compteur += 3;
```

On peut aussi expliciter un objet particulier, ce qui évite de spécifier l'espace de noms à chacune de ses utilisations, mais n'autorise pas l'utilisation des autres objets du même espace de noms.

```
using nom::X;
```

Attention ! L'utilisation des namespace **ne change pas** les règles de résolutions de portée... en cas d'ambiguité, c'est toujours la variable « *la plus proche* » qui est choisie.

Exemple complet

```cpp
#include <iostream>
using namespace std; // utilisation des objets standards (std)

namespace test {
  int i;  // ceci sont des variables utilisables
  int j;  // uniquement dans la portee nommee "test"
}

int i(3);   // ceci est une variable globale

int main(int argc, char* argv[])
{
  int i(1); // voici une variable locale

  test::i = 5; // utilisation des variables de l'espace
  test::j = 6; // de noms "test"

  cout << i << ' ' << ::i << ' ' << test::i << endl;  // 1 3 5
  // cout << j << endl;   // ERREUR: j undeclared

  using test::j;
  cout << j << endl; // signifie test::j

  // using test::i;   // ERREUR: redefinition of i (i local)

  using namespace test;
  cout << i << endl; // c'est quand meme le i local !
  cout << j << endl; // c'est test::j
}
```

Un programme dans son environnement

Un programme est executé dans un **environnement :**
interpréteur de commandes / *système d'exploitation*
Il peut donc interagir avec eux (cf par exemple les flots).

Votre programme est un *processus* du système, une sorte de « fonction ».

En fait, `main()` **est une fonction** (presque) comme les autres...

☞ Elle a juste la spécificité d'être toujours appelée en premier.

Jamila Sam
& Jean-Cédric Chappelier

Valeur de retour de `main()`

Le prototype de `main` tel qu'utilisé jusqu'ici est : `int main()`

A quoi sert le `int` retourné par `main` ?

☞ C'est le « *signal* » retourné au système d'exploitation par le processus correspondant au programme.
(dépasse largement le cadre de ce cours)
choisir 0 si tout va bien, autre chose si il y a une erreur.

Arguments de `main()`

Et si `main()` est une fonction, ...peut-elle prendre des arguments ?

☞ oui

```
int main(int argc, char* argv[])
```

Ces arguments sont les **paramètres** donnés par l'**environnement** (système d'exploitation, interpréteur de commandes, ...) qui appelle la fonction `main`, c'est-à-dire, qui exécute le programme.

Exemple : passer une option « `-v` » et un fichier à un programme

```
monprogramme -v fichier
```


Jamila Sam
& Jean-Cédric Chappelier

Arguments de `main()` (2)

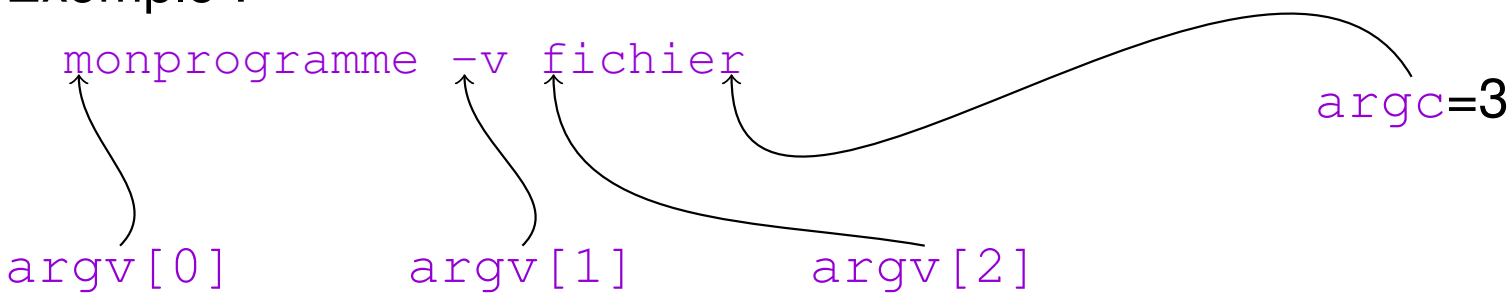
Dans le prototype

```
int main(int argc, char* argv[])
```

`argc` est un entier comptant le nombres d'arguments (+1) passés au programme
`argv` est un tableau de pointeurs sur des caractères : tableau des arguments

☞ `argv[0]` correspond au nom du programme.

Exemple :



Traitement des arguments de `main()`

```
int main(int argc, char* argv[])
{
  int erreur(traite_arguments(argc, argv));
  if (erreur != OK) { // constante OK definie au prealable
    ...
  }
  return erreur;
}
```

On peut distinguer 3 types d'arguments

- obligatoires
  e.g. un nom de fichier : `rm fichier`
- optionnels
  e.g. une option d'affichage : `ls -l`
- optionnels avec arguments
  e.g. changer une valeur par défaut : `dvips -o masortie.ps`

Exemple

```
int traite_arguments(int& nb, char* argv[])
{ int required(0);   // nb d'arguments obligatoires deja traites
  string argument;
  const string pgm_name(argv[0]); // le nom du programme
  --nb;          // passe a l'argument suivant
  size_t i(1);
  while (nb) { // tant qu'il y a des arguments
    argument = argv[i];

    if (argument == "-P") // option
      option_P = true;

    else if (argument == "-i") { // option avec 1 argument : -i nom
      option_I = true;
      ++i; --nb; // passe a l'argument suivant
      if (!nb) { // si l'argument de l'option n'est pas la
        cerr << "ERREUR: pas d'argument pour l'option -i" << endl;
        return ERREUR;
      } else
        fait_ce_qui_faut(argv[i]); // traite l'argument de l'option
    } else { // traite les arguments obligatoires
                // suite ->
```

Exemple (suite)

```
        if (required >= NB_REQUIRED) {
          cerr << "ERREUR: je ne comprend pas l'option " << argument
               << endl;
          return ERREUR;
        } else {
          // arguments obligatoires
          soccupe_argument_obligatoire(argument);
          ++required;
        }
      }
      ++i; --nb; // passe a l'argument suivant
    }

    if (required != NB_REQUIRED) {
      // verifie qu'on a bien eu tous les arguments obligatoires
      cerr << "ERREUR: il manque des arguments" << endl;
      return ERREUR;
    }
    return OK;
  }
```

Pour préparer le prochain cours

- Vidéos et quiz du MOOC semaine 1 :
  - Introduction [20 :48]
  - Classes, objets, attributs et méthodes en C++ [16 :07]
  - public : et private : [18 :59]
  - Encapsulation et abstraction : résumé [10 :28]
  - Encapsulation et abstraction : étude de cas [23 :33]

  ☞ A échelonner sur plusieurs jours pour éviter l'« overdose »

- Le prochain cours :
  - de 14h15 à 15h (résumé et compléments)