Programmation avancée en C++
GIF-1003

Thierry EUDE

Révision Intra

UNIVERSITÉ LAVAL
Département d'informatique
et de génie logiciel

Révision

Codage des algorithmes

- Introduction
- Environnement de développement et processus de compilation
- Éléments de syntaxe et sémantique
  - La représentation des données en C++
  - Instructions et séquencement
  - Conversion de types
  - Entrées-Sortie console
  - Fonctions
  - Pointeurs
  - Tableaux
  - Chaines de caractères type "C"
  - Entrées-Sortie fichiers

Implantation de classe

Considérations de génie logiciel

Environnement de développement intégré (IDE) et étude du cycle de programmation

Théorie du contrat (principe)

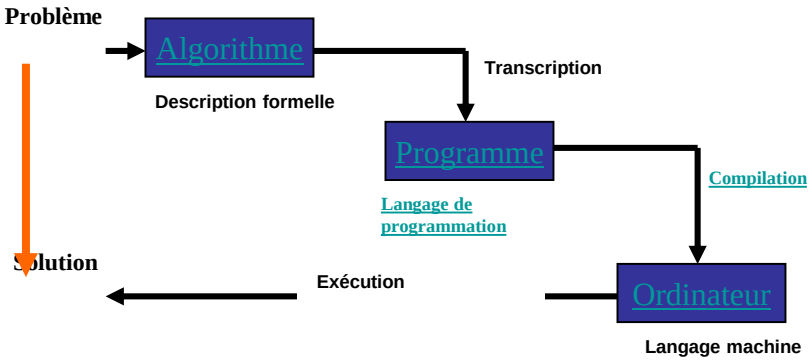
Programmation avancée en C++
GIF-1003

Thierry EUDE

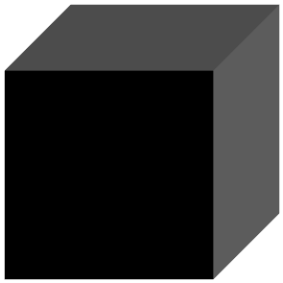
Module 1 : Codage des algorithmes
1.1 Introduction

UNIVERSITÉ LAVAL
Département d'informatique
et de génie logiciel

Qu'est-ce qu'un programme?



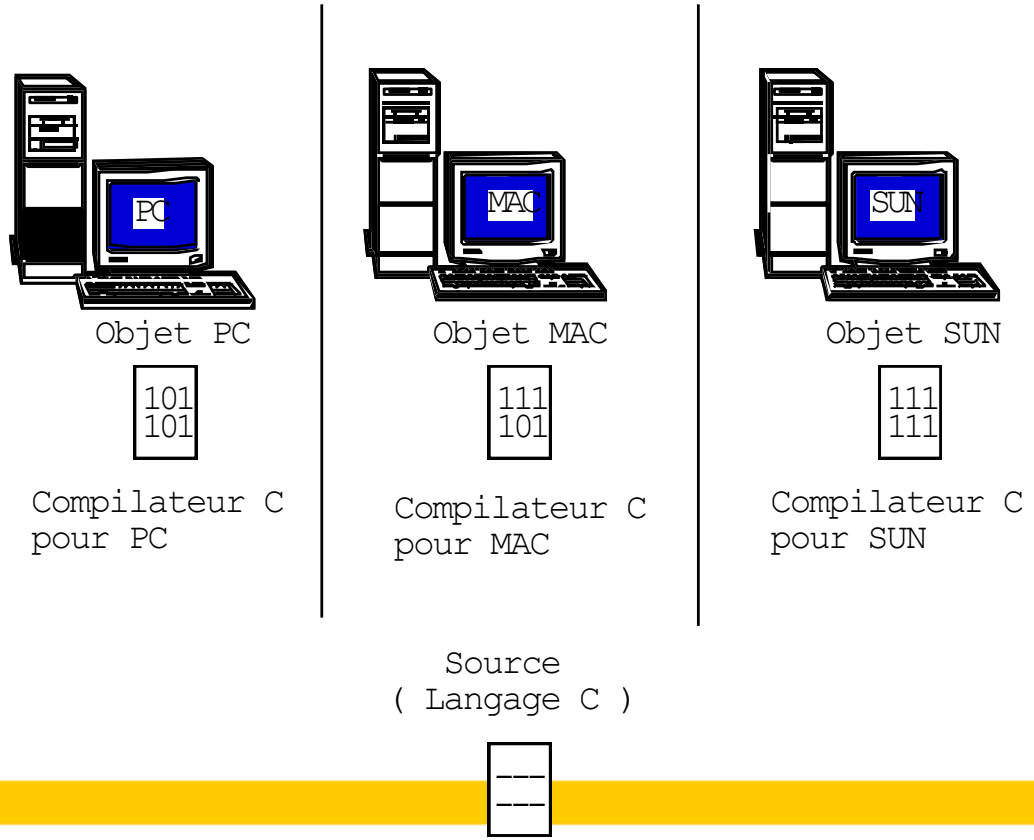
Données en entrée



Données (résultat) en sortie

Les compilateurs

La portabilité d'un programme



Environnements de développement

- Anjuta DevStudio
- Code::Blocks
- C++ Builder
- Dev-C++ et son probable successeur WxDev-C++
- **Eclipse avec le plugin CDT**
- KDevelop
- Microsoft Visual C++
- NetBeans
- QtCreator
- Sun Studio
- Xcode

Programmation avancée en C++
GIF-1003

Module 1 : Codage des algorithmes

1.2 Environnement de développement et processus de compilation

T. Eude

UNIVERSITÉ LAVAL
Département d'informatique
et de génie logiciel

Programmation avancée en C++
GIF-1003

Thierry EUDE

Module 1 : Codage des algorithmes
1.4 Éléments de syntaxe et sémantique

UNIVERSITÉ LAVAL
Département d'informatique
et de génie logiciel

La représentation des données de base en C++

Déclarations de variables

Syntaxe: type identificateur ;

Pourquoi?

- taille de l'espace mémoire requis
- opérations associés
- codage, décodage
- lisibilité (choix des identificateurs)
- détection d'erreurs

Types de base

- **Types entiers:**
  - *(signed) char* *8 bits* *[-127 , 127]*
  - *unsigned char* *8 bits* *[0 , 255]*
  - *int* *32 bits* *[-2147483648 , 2147483647][*
  - *unsigned int* *32 bits* *[0 , 4294967295]*
  - *signed int* *32 bits* *[-2147483648 , 2147483647]*
  - *short int* *16 bits* *[-32768 , 32767]*
  - *unsigned short int* *Range* *[0 , 65,535]*
  - *long int* *32 bits* *[-2,147,483,647 to 2,147,483,647]*
  - *unsigned long int* *32 bits* *[0 , 4,294,967,295]*
  - *long long* 64 bits - [-9 223 372 036 854 775 807,9 223 372 036 854 775 807]
- **Types réels:**
  - *float* - 32 bits -- 3.4 E +/- 38 (7 chiffres significatifs)
  - *double* - 64 bits -- 1.7 E +/- 308 (15 chiffres significatifs)
  - *long double* - 64 bits -- 1.7 E +/- 308 (15 chiffres significatifs)

sizeof()

Les blocs d'instructions

- **Exemple**

```
{
   int nb_etudiants;
      . . .
            {
               int nb_profs;
               nb_etudiants = 0;
                        . . .
               nb_profs = nb_etudiants/100;
                        . . .
            }
      . . .
}
```

Expressions vs. Instructions

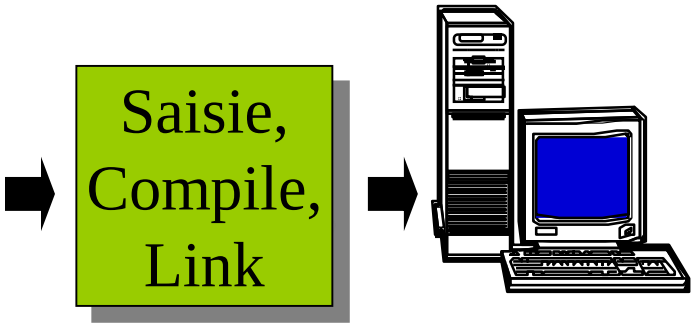
- Une expression a toujours une valeur.
- Une instruction fait référence à la notion de commande (quelque chose qu'on exécute): il n'y a pas de notion de valeur.
- La plupart des composantes syntaxiques du langage C++ sont des expressions.
- Parmi les instructions, on peut citer les structures de contrôle (séquence, itération, choix, etc.).

Exemple

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

static char FIN_SEQUENCE = '.';
char convertionMajuscule(char p_caractere);

int main(void)
/* Connus: une constante FIN_SEQUENCE égale à '.' */
{
      char caractere;
      char caractereMajuscule;
      cout << "Bienvenue!" << endl;
      cin >> caractere;
      while (caractere != FIN_SEQUENCE)
      {
            caractereMajuscule = convertionMajuscule(caractere);
            cout << caractereMajuscule << endl;
            cin >> caractere;
      }
      cout << "Terminé!" << endl;
      return 0;
}
/**
 * \brief convertit un carctère en majuscule
 * \param[in] p_caractere le caractère à convertir
 * \return le caractère converti en majuscule
 */
char convertionMajuscule(char p_caractere)
{
      char caractere;
      caractere = p_caractere - 'a' + 'A';
      return caractere;
}
```



Conversion de types

- La compilation d'un programme
  - Afin de parvenir à générer le code d'un programme, les compilateurs procèdent en 4 phases:
    - l'analyse lexicale;
    - le crible;
    - l'analyse syntaxique;
    - l'analyse sémantique.

Conversion de types

- Les opérateurs arithmétiques sont prévus pour fonctionner sur des opérandes de même type.

- Qu'arrive-t-il lorsque:
  - Les opérandes ont des types différents?
  - Un des opérandes sort du domaine de définition d'un opérateur?

➔ appel au mécanisme de conversions des types des opérandes.

Conversion de types

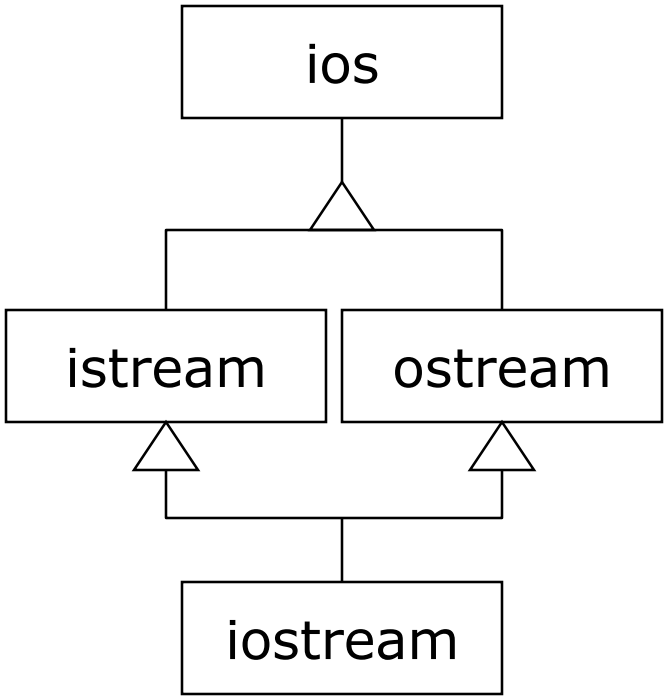
- Une conversion peut être forcée par une affectation:

```
int n;
float x;
n = x + 5.3;
```

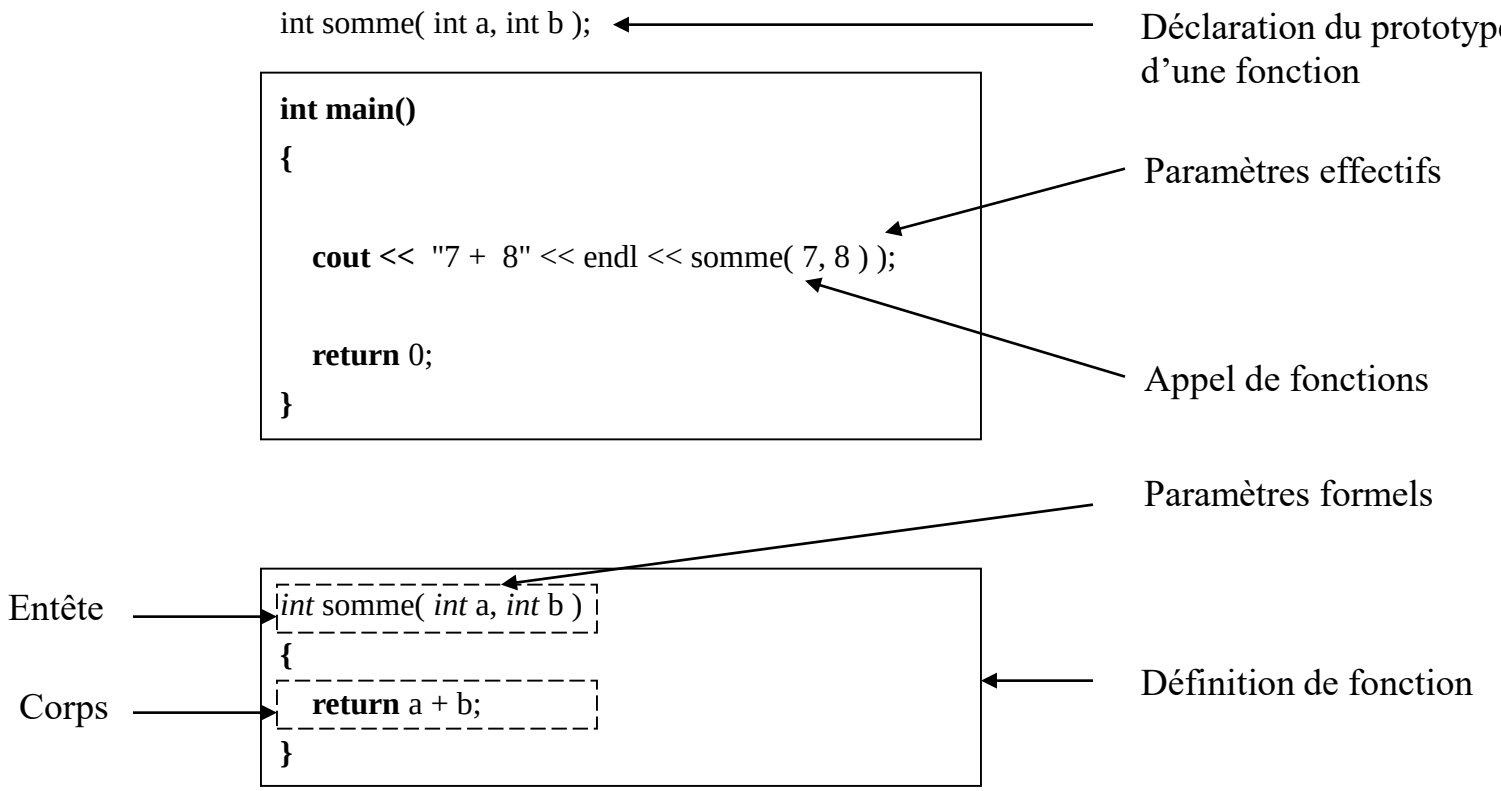
- Comment s'effectue la conversion?
  - La valeur doit être tronquée.
  - On ne conserve que les bits les moins significatifs (les bits de poids faible).

Entrées et sorties : La famille des iostream

- Une **stream** en C++ :
  - séquence de caractères permettant de faire les opérations d'entrées/sorties.
- Une **ostream** gère les sorties
  - inclut la définition de l'opérateur <<.
- Une **istream** gère les entrées
  - inclut la définition de l'opérateur >>.



Déclaration/définition de fonctions 3/3



Variables locales

- Les *variables locales* à une fonction sont dites de classe automatique
- Leur portée est limitée à cette fonction:
  - ne sont connues qu'à l'intérieur de la fonction où elles sont définies.
- À chaque appel de fonction, les variables locales sont définies, initialisées, utilisées et détruites après l'exécution de la fonction.

Exemple…

Passage de paramètres par valeur

```
void echange (int a, int b);

int main()
{
   int x = 10;
   int y = 20;

   cout <<"avant appel: \t" << x << y << endl;

   echange (x, y);

   cout <<" après appel: \t" << x << y << endl;

   return 0;
}

void echange (int a, int b)
{
    int temp;
    cout << "début échange: \t" << a << b << endl;
    temp = a;
    a = b;
    b = temp;
    cout << "fin échange: \t" << a << b << endl;
}
```

```
avant appel:      10 20
début échange:  10 20
Fin échange:      20 10
Après appel:      10 20
```

Étapes d'évaluation:

1. Évaluations des paramètres effectifs (on obtient des valeurs).
2. Appel de la fonction.
   a. Les variables locales et les paramètres formels sont créés.
   b. Les paramètres formels prennent les valeurs résultants de l'évaluation 1.
   c. Exécution du corps de la fonction.
   d. Retour d'une valeur en résultat
   e. Les variables créées en a sont détruites.
3. Affectation de la valeur retournée à une variable.
4. on reprend l'exécution à l'endroit où s'est effectué l'appel de la fonction.

Passage de paramètres par adresse

```
void echange (int *ptr_a, int *ptr_b);

int main()
{
   int x = 10;
   int y = 20;

   cout <<"avant appel: \t" << x << y << endl;

   echange ( &x, &y );

   cout <<" après appel: \t" << x << y << endl;

   return 0;
}

void echange (int *ptr_a, int *ptr_b)
{
    int temp;
    cout <<"début échange: \t" << *ptr_a << *ptr_b << endl;
    temp = *ptr_a;
    *ptr_a = *ptr_b;
    *ptr_b = temp;
    cout <<" fin échange: \" << *ptr_a << *ptr_b << endl;
}
```

```
avant appel:      10 20
début échange:  10 20
Fin échange:      20 10
Après appel:      20 10
```

Solution$_3$: référence (on y reviendra)

Les pointeurs ☹

- Définition
  - Chaque variable, chaque espace mémoire dans un ordinateur possède une **adresse**.
  - L'adresse d'une variable permet donc de la retrouver (l'identifier) dans la mémoire principale de l'ordinateur.
  - Un **pointeur** est une variable qui contient l'adresse d'une autre variable.

L'arithmétique des pointeurs

```
int * ad1, * ad2,  * ad;
int n = 10, p = 20;
```

- Considérons maintenant ces instructions :

```
ad1 = &n;
ad2 = &p;

*ad1 = * ad2 + 2;        (même effet que n = p + 2;)
(*ad1) += 3              (même effet que n = n + 3
(*ad1)++                 (même effet que n++)
```

Résumé

(&ad)++ ou (&p)++ seront rejetées à la compilation.

int * ad1; réserve un emplacement pour un pointeur sur un entier. Elle ne réserve pas en plus un emplacement pour un entier.

ad + 1; a un sens en C.

double * ad2; ad2 occupe le même espace mémoire que ad1

ad++; ad désigne l'objet suivant en mémoire.

ad += 10; instruction légale

ad -= 25; instruction légale

Tableau

- **Quoi?**
  - Un tableau est une collection (un regroupement) de variables de même type.
- **Quand?**
  - Ils sont généralement utilisés pour conserver les mêmes informations sur plusieurs objets différents. Par exemple, les notes des étudiants d'une classe.
  - Ils sont aussi fort utiles pour mémoriser des informations et revenir les traiter plus tard.
- **Comment?** →

Déclaration

- Un tableau est une variable comme une autre. On doit donc lui donner un *type* et un *nom*.
- Les **crochets** indiquent que la variable est un tableau.
- Particularité des tableaux, on doit leur donner une taille (un nombre de cases). Cette valeur doit être une constante explicite:

```
int monTableau[50];

const int MAX_CASES = 256;
float monAutreTableau[MAX_CASES];
```

Accès aux cases d'un tableau

- On accède (pour lire ou écrire) à une case d'un tableau par son indice:

```
int monTableau[10];
int maVariable= 5;

monTableau[0] = maVariable;
 maVariable = monTableau[1];
```

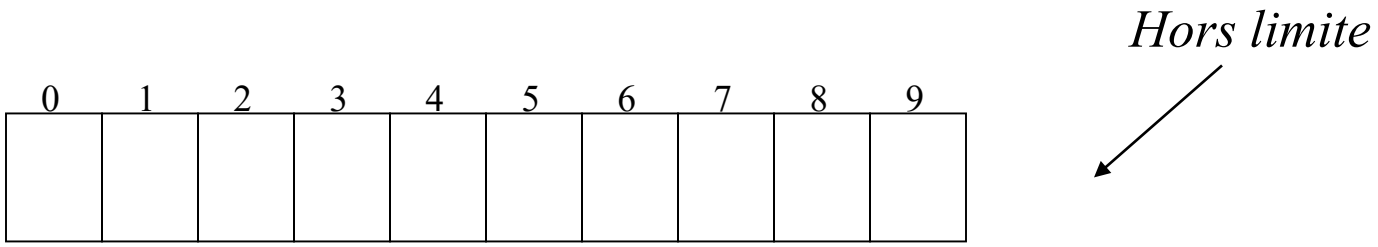
- Dans ce cas, on peut utiliser une variable:

```
int i = 0;
monTableau[i] = 3;
monTableau[i + 1] = 0;
```

Bornes d'un tableau

- Tous les tableaux en C commencent à **l'indice 0**. Ainsi, la première case d'un tableau de 10 cases sera désignée par l'indice 0 et la dernière par l'indice 9.

- On doit porter une grande attention aux accès ***hors limites*** (indices invalides).



Initialisation (suite)

- Ou bien toutes à la fois:

```
int monTableau[5] = {0, 0, 0, 0, 0};
int tab[5] = { 10, 20 };
int tab[] = { 10, 20, 5, 0, 3 };
```

- Il est impossible d'assigner un tableau à un autre:

```
int monTableau[5] = {0, 0, 0, 0, 0};
int monAutreTableau[5];
monAutreTableau = monTableau;          /* ERREUR */
```

il faut écrire explicitement :

```
for (i = 0; i < n; i++)
    {
        monAutreTableau[i] = monTableau[i];
    }
```

Pointeurs et tableaux

- Si *t* est un tableau de taille 10, ses éléments sont notés *t*[0], *t*[1] ... *t*[9] ;

mais que signifie le symbole *t* seul ?

- Le nom d'un tableau désigne son adresse.
- L'adresse d'un tableau est aussi l'adresse de son premier élément ;

on dit parfois que c'est l'adresse de la ***base*** du tableau.

- On peut faire des calculs sur les adresses : *t*+*i* est l'adresse de *t*[*i*]. Cette convention concorde avec celle de démarrer la numérotation des éléments à 0.

&(monTableau[0]) ➔ monTableau
monTableau[5] ➔ *(monTableau + 5)

Un nom de tableau est un pointeur

```
int t[10];


int i;
for (i = 0; i < 10; i++)
{
        * (t + i) = 1;
}
```

```
int t[10];
int i;
int* p;
p = t;
for (i = 0; i < 10; i++)
{  *p = 1;
                p++;
}
```

```
int t[10];
int* p;
for (p = t; p < t + 10; p++)
{
        *p = 1;
}
```

```
int t[10];
        int i;
for (i = 0; i < 10; i++)
        {
                t[i] = 1;
        }
```

Tableaux et fonctions

- Pour qu'une fonction puisse manipuler un tableau, il suffit d'en déclarer un dans sa liste de paramètres:

```
void  MaFonction(int unTableau[10]);
```

- Et, à l'appel:

```
int monTableau[10];

MaFonction(monTableau);
```

Les tableaux à une dimension transmis en argument à une fonction

```
void fct (int t[ ])
{ /* . . . . . */
        int i;
        for (i = 0; i < 10; i++)  t[i] = 1;
}

int main()
{
      int tab1[10], tab2[10];
            . . .
      fct (tab1);
            . . .
      fct (tab2);
}
```

void fct (int t[10])
void fct (int* t)
void fct (int t[ ])

Ou bien..
for (i = 0; i < 10; i++) *(t+i) = 1;

Tableaux à 2 dimensions

- Ou un tableau de tableaux…
- Se déclarent:

```
int monTableau[5][3];
```

- Pour accéder à une case du tableau, on doit spécifier ses 2 indices:

```
monTableau[0][1] = 5;
```

Initialisation

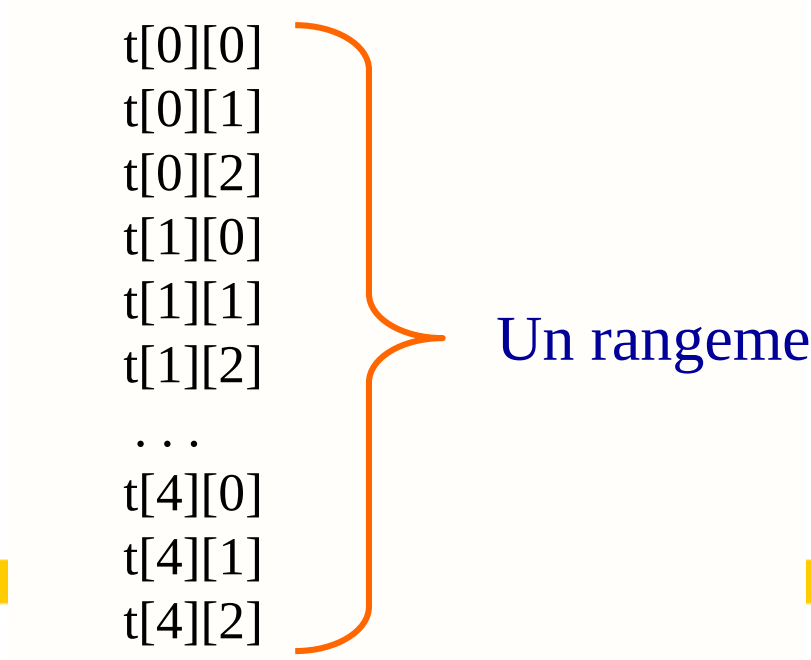
```
int tab [3] [4] = {{ 1, 2, 3, 4 }, {5, 6, 7, 8}, {9, 10, 11, 12}};
int tab [3] [4] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 };
```

```
int tab [3] [4] = {{ 1, 2, 3, 4}, {5, 6, 7, 8}};
int tab [3] [4] = {1, 2, 3, 4, 5, 6, 7, 8};
```

Les tableaux à plusieurs dimensions

```
int t[5][3];
t[3][2]          t[i][j]          t[i-3][i +j]
```

Arrangement en mémoire des tableaux à plusieurs indices



Tableaux et fonctions

```
void initialise(int tab[10][15]);

int main(void)
{
   int monTableau2[10][15];

   initialise(monTableau2);
   …
}

void initialise(int tab[10][15])
{
   int i, j;

   for (i = 0; i < 10; i++)
   {
       for (j = 0; j < 15; j++)
       {
            tab[i][j] = 0;
       }
   }
```

Ou bien

void initialise (int t[ ]**[15]**)

Important de fixer le nombre de colonnes.

Pointeurs et tableaux

```
void init (int t[ ][NBCOLONNES], int r, int c)
   { /* . . . . . . . . . . . . */
        int i, j;
        for (i = 0; i <r; i++)
                    for (j = 0; j < c; j++)
                              t[i][j] = 1;
   }

int main()
{ int mat [12][20];
            . . .
      init (mat, 12, 20);
      ..
}
```

```
void init (int *t, int r, int c)
{/*. . . . . . . . . . . . . . . . . . */
   int i,j;
   for (i = 0; i < r; i++)
            for (j = 0; j < c; j++)
                 *(( t +(i * c)) + j )= 1;
}
```

Les chaînes de caractères type "C"

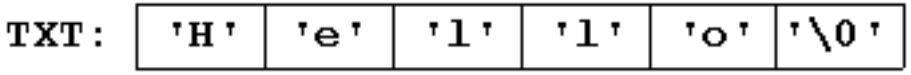
- Une chaîne de caractère est un tableau de caractères qui, par convention, se termine par le caractère spécial '\0'.

➔ pour contenir un mot de 5 caractères, on doit déclarer un tableau de 6 cases.

➔ convention possible grâce au fait que le standard ASCII prévoit qu'aucun caractère ne correspond au code 0.

➔ convention aussi fort pratique : donne une sentinelle parfaite pouvant être utilisée pour développer un ensemble de fonctions de manipulations de chaînes de caractères.

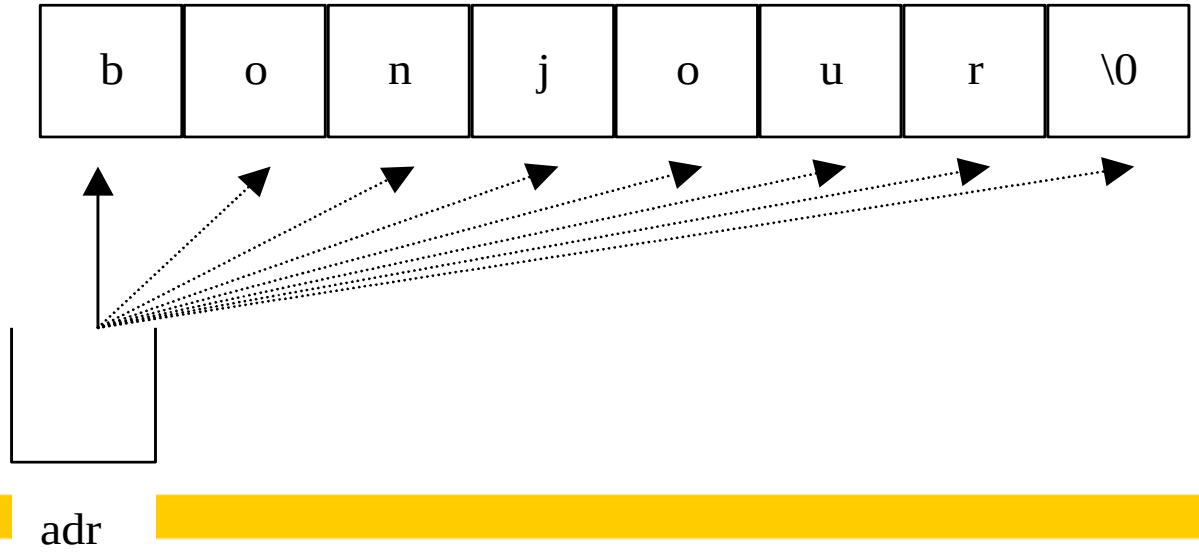
```
char TXT[] = "Hello";
```



Les chaînes de caractères

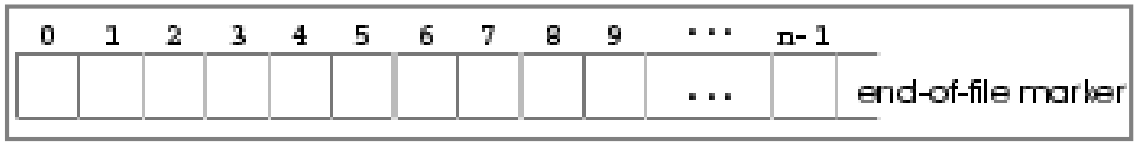
- *Cas des chaînes constantes*
  - la notation "bonjour" a comme valeur, non pas la valeur de la chaîne elle-même, mais son adresse.

➔ C'est le même principe que pour les tableaux.



Les entrée/sortie avec des fichiers

- Fichier : séquence d'octets terminée par une « *marque de fin de fichier* » *(EOF)*.



- En C++ : vus comme des flux d'octets ou de caractères.
  - **ofstream** : Pour l'écriture dans un fichier.
  - **ifstream** : Pour la lecture à partir d'un fichier.

- Deux types de lecture/écriture :
  - Mode texte : Peut être lu comme un document texte, caractère par caractère, mot par mot, ligne par ligne, etc.
  - Mode binaire : décoder les informations, octet par octet.

- Se trouve dans l'entête <fstream>

Écriture en mode texte

- Pour ouvrir un fichier en écriture : déclarer un objet de type ofstream :

```
ofstream ofs("FichierAEcrire.dat",ios::out);
```

  - 1er argument : le chemin et le nom du fichier.
  - 2e argument : le type d'ouverture
    - ios::out : Efface tout le contenu, si le fichier existait déjà.
    - ios::app : Écrit après ce qu'il y a déjà dans le fichier.
  - Si le fichier n'existe pas, il est créé dans les deux cas.
- On peut aussi déclarer un ofstream sans ouvrir le fichier.
  - Pour ouvrir le fichier plus tard, utilisez la méthode `open`

```
ofstream ofs;
// ...
ofs.open("FichierAEcrire.dat",ios::app);
```

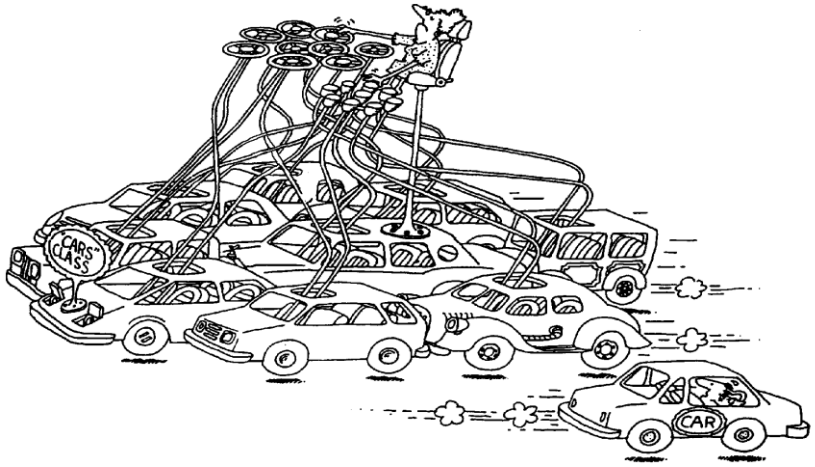
Programmation avancée en C++
GIF-1003

Thierry EUDE

Module 2 : Implantation de classe

UNIVERSITÉ LAVAL
Département d'informatique
et de génie logiciel

La classe ?



Objet = entité discrète qui
existe dans le temps
et dans l'espace.

Classe = ***abstraction*** qui n'existe que dans les programmes.

- ***doit*** représenter un ensemble d'objets qui partagent une ***structure commune*** et un ***comportement commun***.
- définit
  - les ***opérations*** permises sur les objets de la classe.
  - les ***états*** qu'ils peuvent prendre (transitions d'états).

instance, ***occurrence*** et ***objet*** sont synonymes

Interface et implémentation

- L'***interface*** d'une classe
  - définit ce qui est vu de l'***extérieur***.
    - renforce la notion d'abstraction en ne montrant que ce qui est ***nécessaire***.
  - contient principalement les déclarations de toutes les opérations applicables sur les instances de cette classe.
- L'***implémentation*** d'une classe
- est ce qui est caché à l'***intérieur***,
  - principalement le code réalisant les opérations définies dans l'interface.

La classe et ses objets

- La classe
  - responsable d'assurer la validité de tous les objets créés à partir d'elle-même.
- Un objet doit être ***valide*** de sa **création** jusqu'à sa **destruction**.
  - ne pas permettre l'accès direct aux attributs!
    - ➔ Encapsulation
- La théorie du contrat pourrait gérer cette validité (nous y reviendrons).

Principe d'encapsulation

- Problème :

- Accès aux attributs **non contrôlée**.
  - Possibilité de non-initialisation.
  - Possibilité de mauvaise assignation.
- Pas de garantie sur la **validité** des données.

Implémentation d'une classe : les étapes

Déclaration de la classe:

- attributs dans la section **privée**
- *méthodes* dans la section **publique**
  - ✓ Norme : Toujours spécifier la visibilité explicitement.

Implantation des méthodes

- après la déclaration de la classe
  - ➢ dans un fichier séparé (pour le C++).

Utiliser la classe

  - ➢ en créant des objets de ce type dans un programme.

Initialisation

- création d'un objet,
  - ses attributs doivent être initialisés pour que celui-ci soit dans un état cohérent.
    - ➢ constructeur.

- **constructeur = méthode spéciale**.
  - a le même nom que la classe.
  - est automatiquement appelé à la création d'un objet de la classe.
  - méthode qui n'a pas de type de retour
  - ne retourne pas de valeur.

Les constructeurs

- Il est possible d'avoir plusieurs constructeurs dans une même classe.

- Caractéristique du langage :
  - Plusieurs méthodes du même nom
    = **surcharge de méthode**.

- Lorsqu'une fonction surchargée est appelée, le compilateur sélectionne la bonne fonction selon les arguments: nombre, type, ordre..

Le destructeur

- méthode particulière de la classe.
- porte le même nom que la classe mais précédé par un tilde ~.
- est exactement le complément du constructeur
  - Le constructeur initialise les attributs et alloue des ressources s'il y a lieu,
  - le destructeur libère les ressources si nécessaire.
- Ne détruit pas l'objet:
  - il fait le ménage à l'intérieur avant la destruction.

2. Concepts orientés objets

2.1 Le modèle objet

2.2 La classe

2.3 Méthodes par catégorie

Département d'informatique - Université Laval

UNIVERSITÉ LAVAL

Méthodes par catégorie

- Accès
- Assignation
- Comparaison
- Utilitaires
- Statiques

Méthodes constantes

- Outil permettant d'améliorer la qualité du code de façon très importante
  - ➢il force la classification.
- Il faut adhérer à cette façon de faire si on désire partager du code et utiliser des librairies qui ont adhéré à cette norme.
- La librairie standard adhère à cette norme.

Méthodes statiques

- Méthode de classe ne nécessitant pas la présence d'un objet pour le traitement
  - méthode statique.

- Comme une fonction mais associée à la classe.
- Ne touche pas aux attributs de la classe, à moins que ceux-ci ne soient aussi statiques (attribut commun à tous les objets de la classe).

2. Concepts orientés objets
    2.1 Élément du modèle objet
    2.2 La classe
    2.3 Méthodes par catégorie

2.4 Implantation avancée

Département d'informatique - Université Laval

UNIVERSITÉ LAVAL

Implantation avancée

- Problème de collision de noms
- Surcharge de méthode
- Le passage de paramètres
- Le retour

Bon usage du using

- Inutile de mettre des using pour chacun des namespaces disponibles :
  - l'ambiguïté revient.
- Un using peut-être local :

```
// Quelque part
  {
    using namespace Util;
    // Util est accessible directement dans ce bloc
  }
```

- **Ne jamais utiliser** de using dans un .h :
  - tous les fichiers qui l'incluront seront *pollués*.

Surcharge de méthode

- Les langages orientés objet permettent une première forme de polymorphisme: la surcharge de méthode (ou de fonction).
- Il est permis de créer plusieurs méthodes du même nom dans la classe pourvu qu'elles varient au niveau des paramètres.
- Lorsqu'une méthode surchargée est appelée, le compilateur sélectionne la bonne selon les paramètres: nombre, type, ordre.
- La surcharge doit être utilisée pour implanter des traitements similaires.

Paramètre par défaut (1)

- Parfois, certaines méthodes ont besoins de plusieurs paramètres pour correctement faire le travail.
- Toutefois pour les cas simples ou les plus fréquents, ces paramètres supplémentaires sont toujours les mêmes.
- Il est possible d'utiliser la technique des paramètres par défaut.

Surcharge des opérateurs

Définition:

Consiste à redéfinir des opérateurs usuels comme +, -, *, /, =, <, [], etc...
sur de nouveaux types : les classes.

Particulièrement intéressante avec
- les nombres complexes, les fractions, les vecteurs, les matrices, les chaînes de caractères...

Dans certaines situations, rend la lecture du code plus facile :

```
d = d1.plus(d2) --> d = d1 + d2
```

Le passage de paramètre

- Types de passage de paramètres en C++ ?
- trois types :
  - le passage par valeur,
  - le passage par pointeur,
  - le passage par référence (nouveauté par rapport au C).

- Quand utiliser chacun de ces types?

Retour par reference ou par valeur?

Par reference

```
Mot& getAdresse() const;
```

- Risque de retour d'une variable locale
- Non respect de l'encapsulation

Par reference constante

```
const Mot& getAdresse() const;
```

Par valeur

```
Mot reqAdresse() const;
```

- Pour retourner le contenu d'une variable locale
- Appelle le constructeur copie si c'est un objet (dépend du compilateur **RVO :** Return value optimization)
- Plus simple à utiliser

Programmation avancée en C++
GIF-1003

Module 3 :
Considérations de génie logiciel

T. Eude

UNIVERSITÉ LAVAL
Département d'informatique
et de génie logiciel

3. Considérations de génie logiciel

3.1 Environnement de développement intégré (IDE) et étude du cycle de programmation

Université Laval
Département
d'informatique

UNIVERSITÉ LAVAL

Omniprésence de la complexité

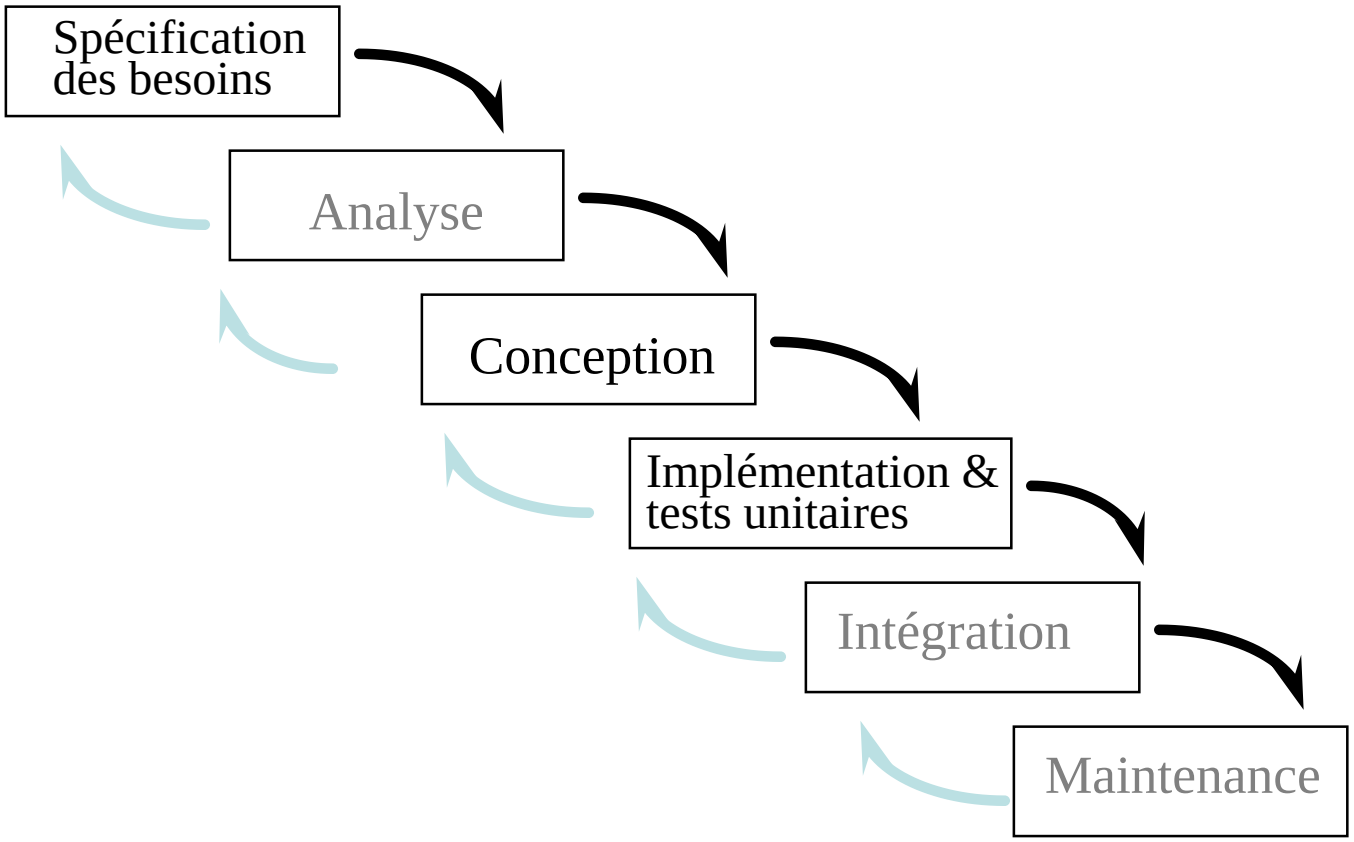
- Pour maîtriser la complexité :

  développer des façons de faire plus disciplinées et plus performantes

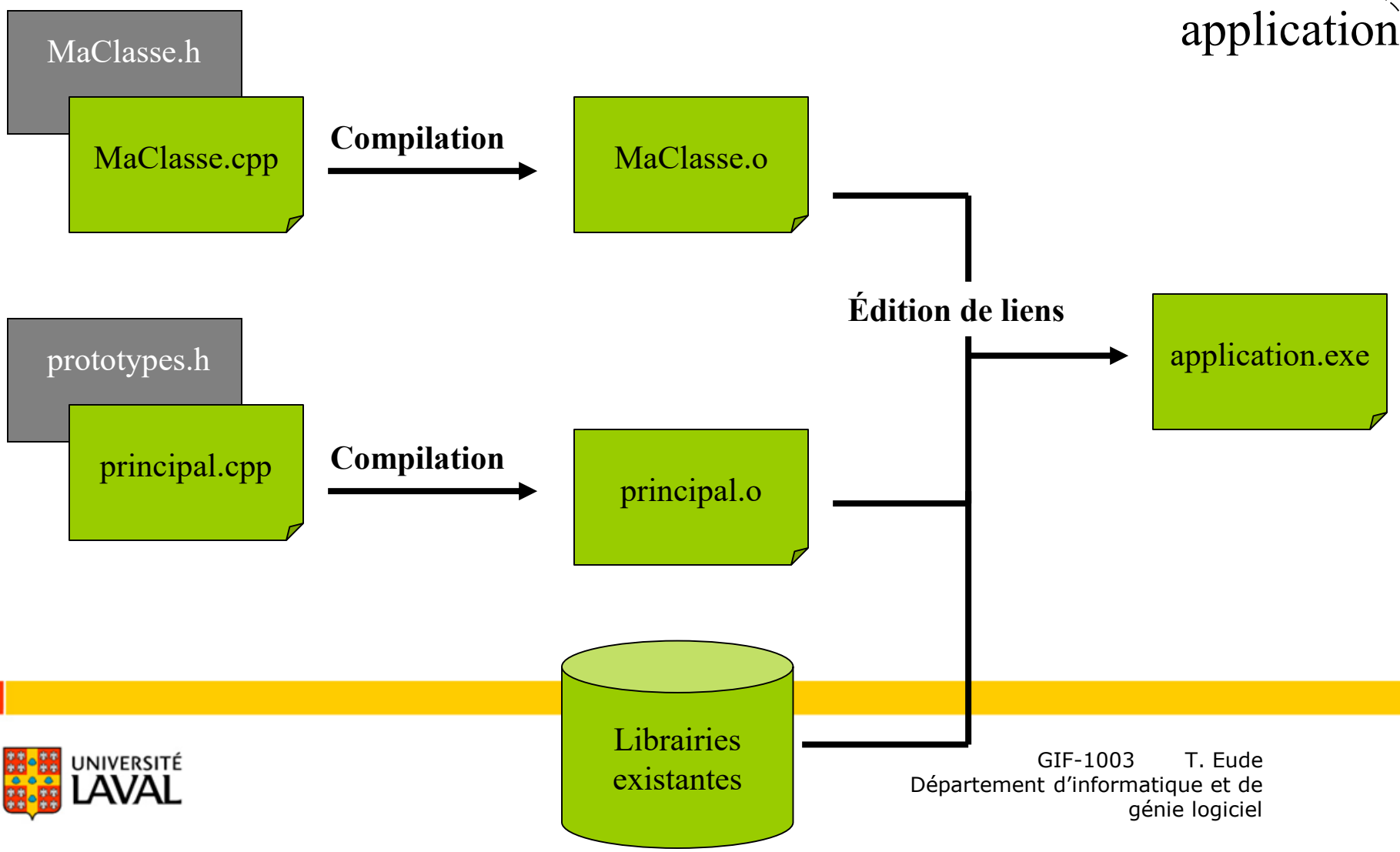
Analyse,
Design et
Programmation orientés objets

Outils performants fondés sur les avancements éprouvés des méthodes conventionnelles.

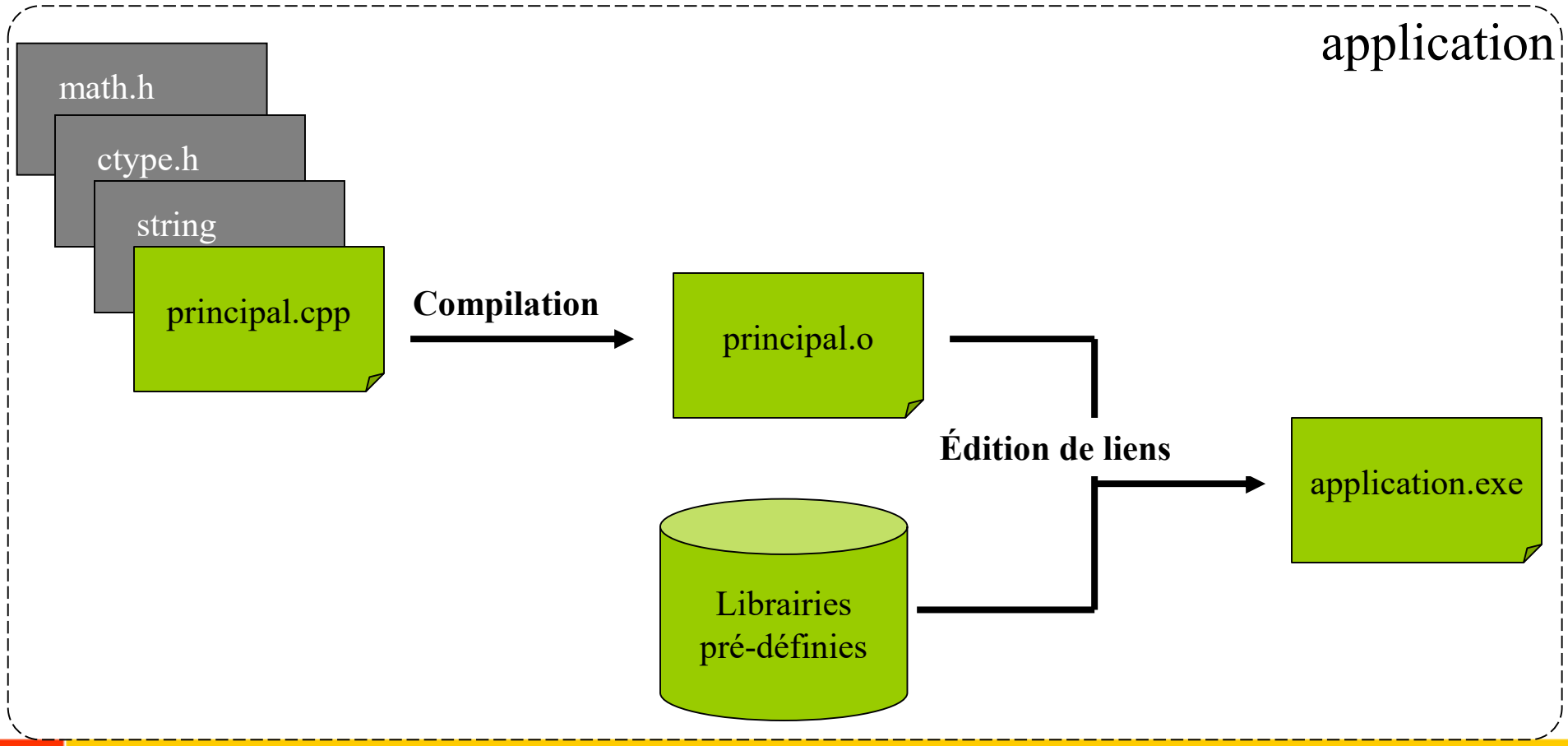
Modèle en cascade



Compilation & Édition de liens



Librairies pré-définies (#include)



3. Considérations de génie logiciel

3.1 Modélisation : Diagramme de classe

3.3 La théorie du contrat

Département d'informatique - Université Laval

UNIVERSITÉ LAVAL

Un contrat protège les deux parties

- Protège le client (dans ce cas ci vous-mêmes)
  - Spécifie quelles sont ses obligations.
  - En retour, est assuré d'un certain résultat.

- Protège le fournisseur (sous-contractant)
  - Spécifie le minimum requis (de la part du client):
    - le fournisseur ne peut être tenu responsable d'avoir manqué à des engagements ne figurant pas dans le contrat.

Les assertions : le contrat du logiciel

- Pour produire du logiciel correct et robuste :
  - rendre les obligations et les garanties explicites pour chaque appel de méthode.
- Mécanismes pour exprimer ces conditions :

❑ **assertions**

❑ **préconditions**

❑ **postconditions :**

- assertions s'appliquant à des méthodes en particulier,

❑ **invariants** de classe :

- assertions s'appliquant à toutes les méthodes d'une classe en tout temps.

Pour une spécification complète

- Une spécification plus complète d'une opération offerte par un composant deviendrait:
  - le nom de l'opération
  - le nombre, le type et l'ordre des paramètres
  - les préconditions
  - les postconditions
  - le type de retour.

BONNE RÉVISION!