Bertrand
LE GAL

Maître de conférences

ENSEIRB
bertrand.legal@enseirb.fr
http://www.enseirb.fr/~legal/

Laboratoire IMS
bertrand.legal@ims-bordeaux.fr
Université de Bordeaux 1
351, cours de la Libération
33405 Talence - France

Filière Electronique
2ème année
2012 / 2013

" Programmation Orientée Objets Appliquée au Langage C++ "

Plan du cours

I - Evolution de l'informatique

II - Rappel sur les flots de conception

III - Rappels sur le langage C

IV - Introduction aux objets (C++)

V - Notions objet avancées (C++)

VI - Conclusion sur le développement objet

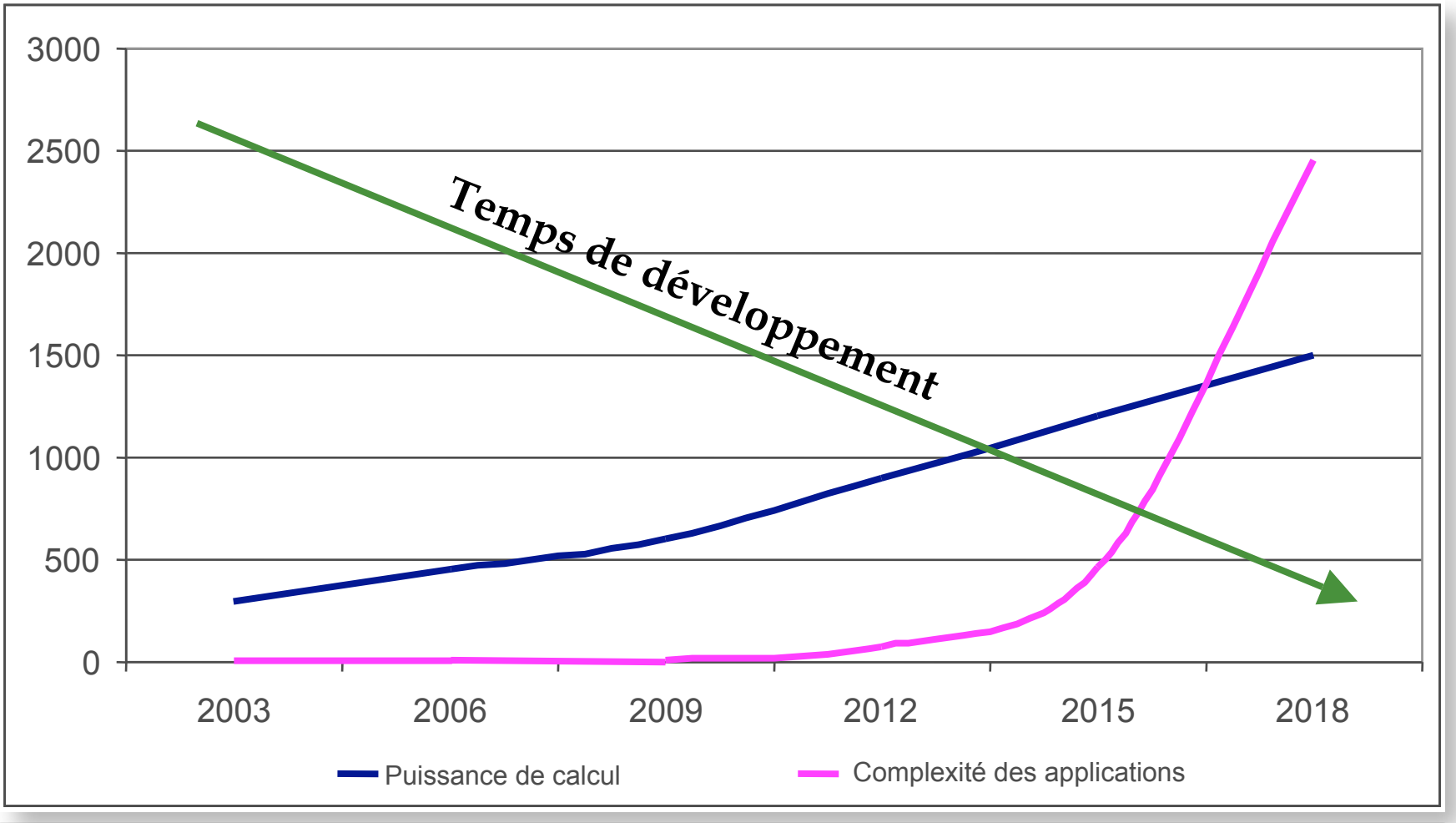
VII - Les bibliothèques

VIII - Vérification fonctionnelle

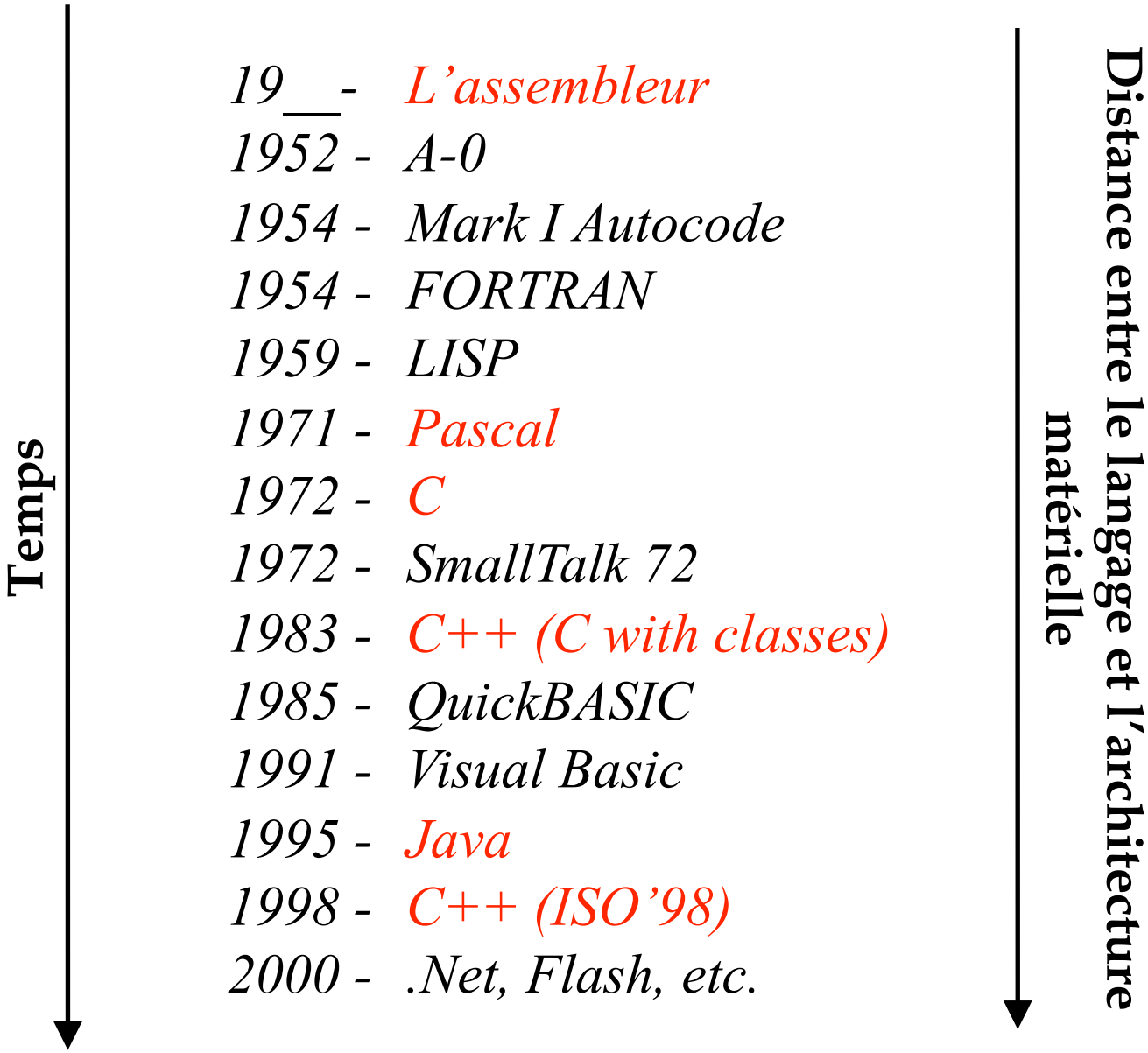
1

“ Evolution du développement informatique ”

La paradigme du développement logiciel

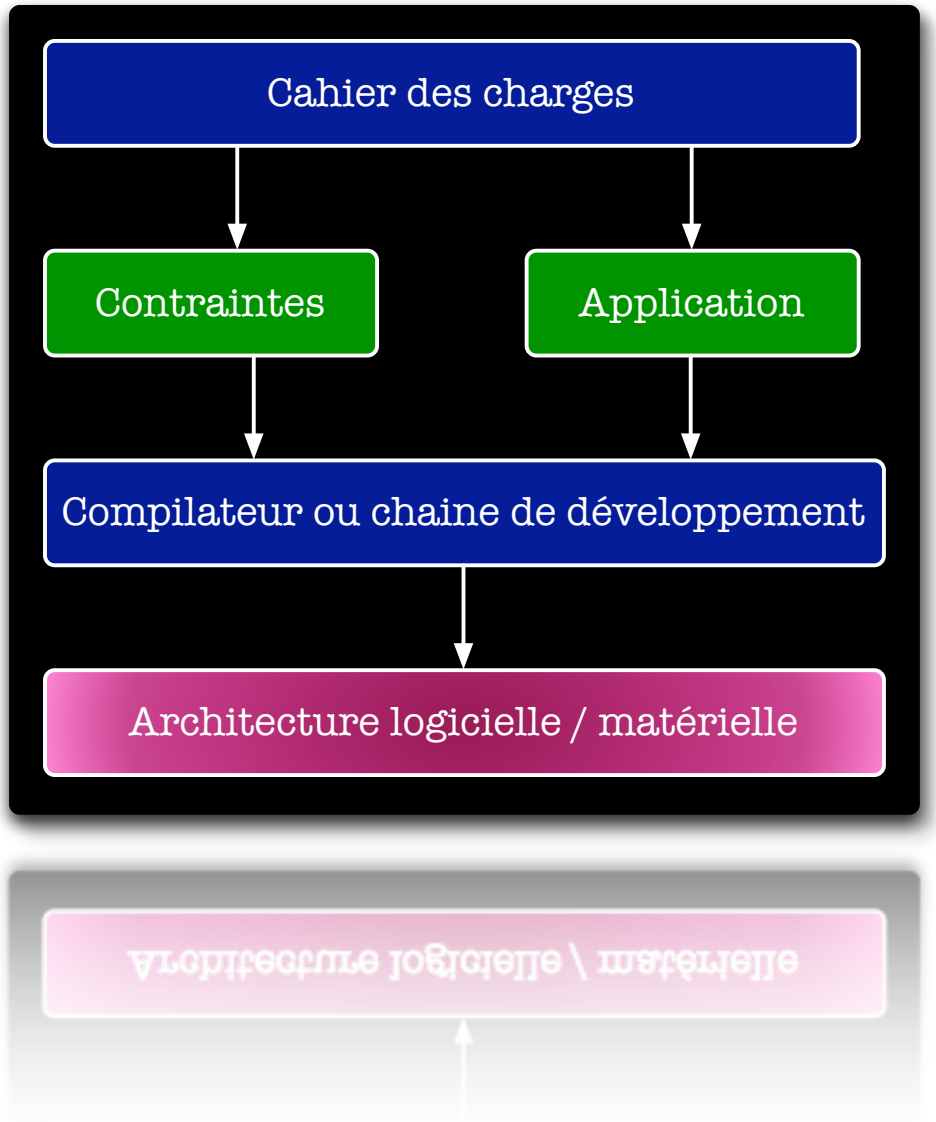


Evolution des langages informatiques



Augmenter le niveau d'abstraction

- Exploitant des langages de très hauts niveaux capables de réaliser des « méta-opérations »
  - L'outil MatLab
  - Langages WEB, Bases de données, etc.
- Utilisation de l'intelligence des compilateurs qui doivent optimiser au mieux…
  - S'éloigner le plus possible du matériel qui est hétérogène,
  - On se repose sur des outils qui savent faire...



La réduction de la taille du code source

Sommation sur les données d'une matrice à 2 dimensions

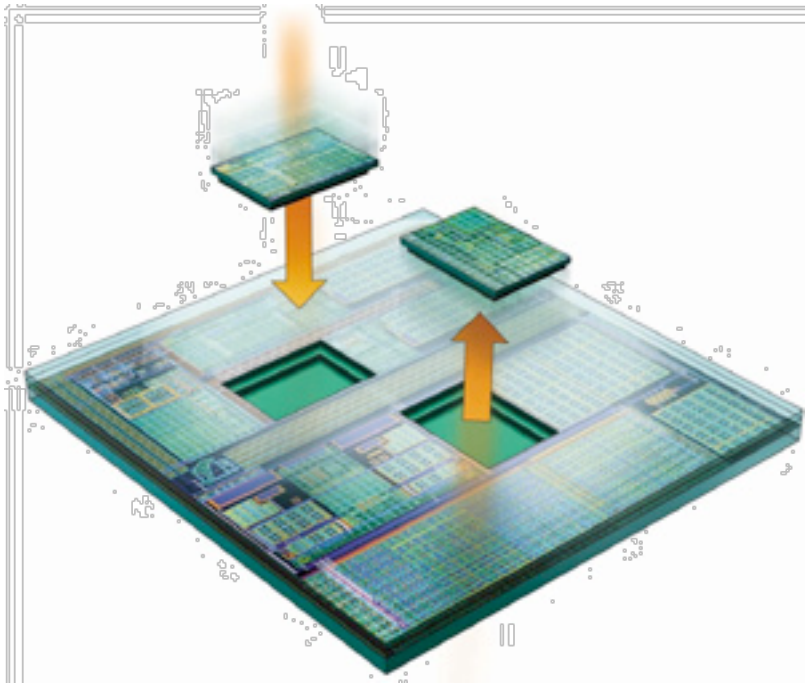
```
// Matlab code
s = sum( sum( A ) );
```

```
// C code
int s = 0;
for(int x=0; x<N; x++){
    for(int x=0; x<N; x++){
        s += a[y, x];
    }
}
```

```
// ASM Code
push    ebp         ; Save caller's stack frame
mov     ebp, esp    ; Establish new stack frame
sub     esp, 4      ; Allocate local data space of
push    esi         ; Save critical registers
load    ...
...
...
sub      eax, 1
jnz      loop
...
pop     edi         ; Restore critical registers
pop     esi          ;
mov     esp, ebp    ; Restore the stack
pop     ebp         ; Restore the frame
ret                 ; Return to caller
```

La réutilisation du préconçu

- Réutiliser les développements déjà réalisés,
  - Valable en interne entre les projets,
  - Achat de composants externes chez des marchants, fournisseurs de bibliothèques,
- Nécessité de posséder une bonne connaissance des objets (programmes) achetés pour les réemployer
  - Temps d'exécution,
  - Complexité algorithmique,
  - Consommation mémoire,
  - Interface d'utilisation,



Méthodologie valable dans le cadre du développement de systèmes logiciels et matériels

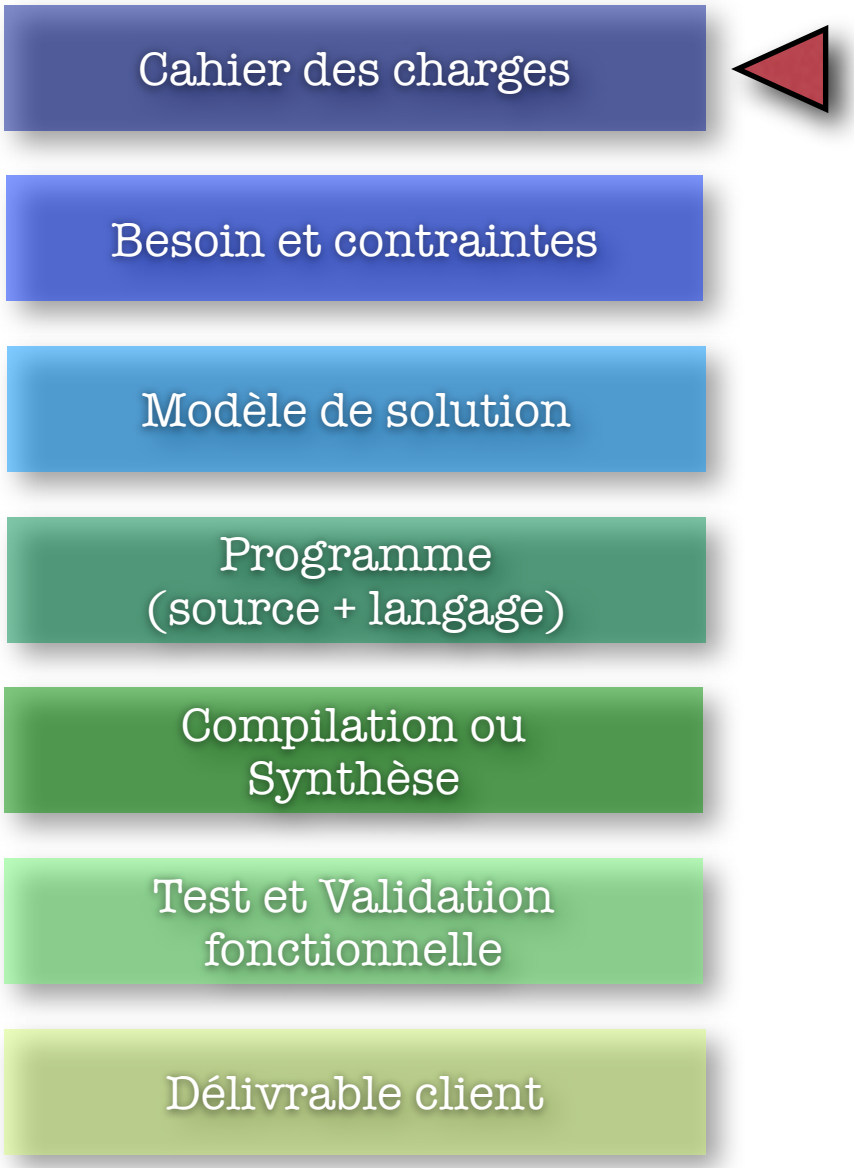
Exigences de qualité

- Intérêt de la dernière piste de progrès présentée :
  - Les composants (objets) que nous avons achetés ont été testés et validés,
  - Ils permettent d'implémenter des parties de l'applications sans besoin de connaissances dans le domaine,
  - Ils respectent certaines contraintes de conception et peuvent être flexibles (compromis temps d'exécution / occupation mémoire).
- Ils nécessitent tout de même pour être réemployé *une documentation suffisante* !

2

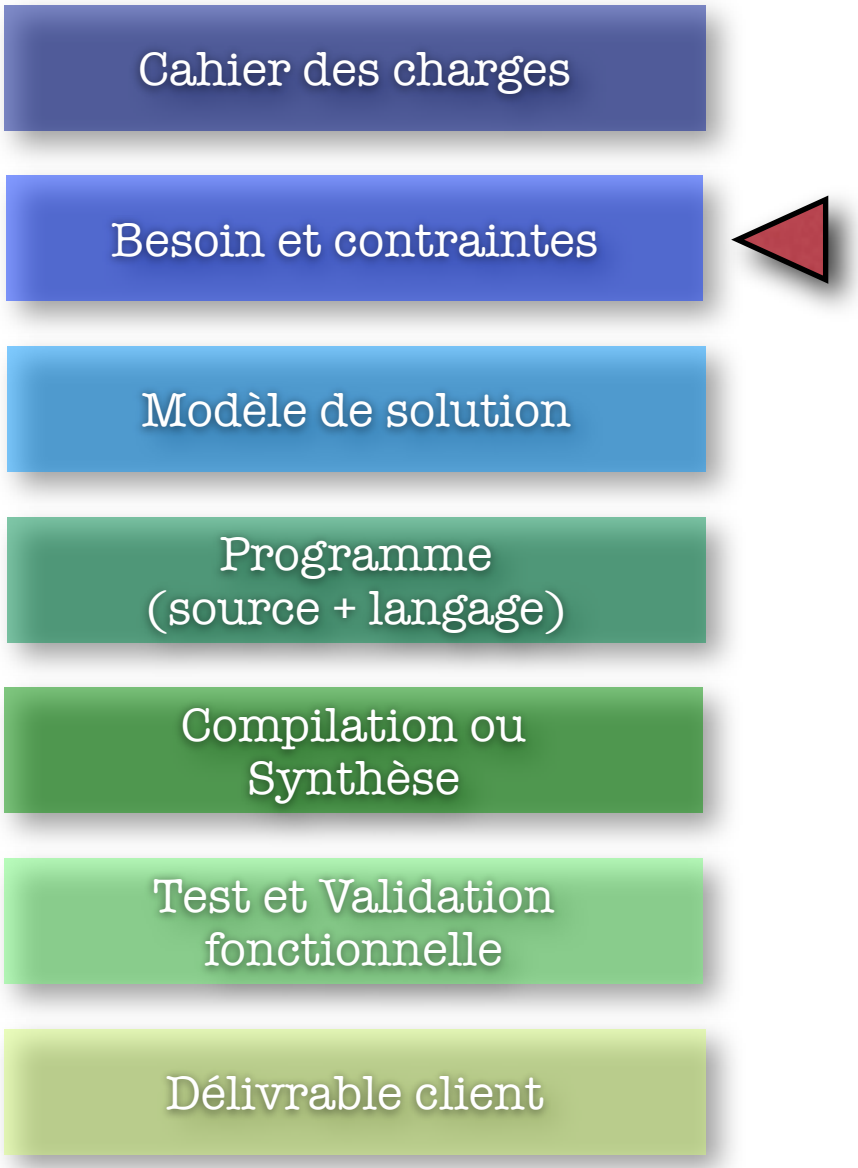
“ Introduction aux flots de conception ”

Le flot de conception en informatique



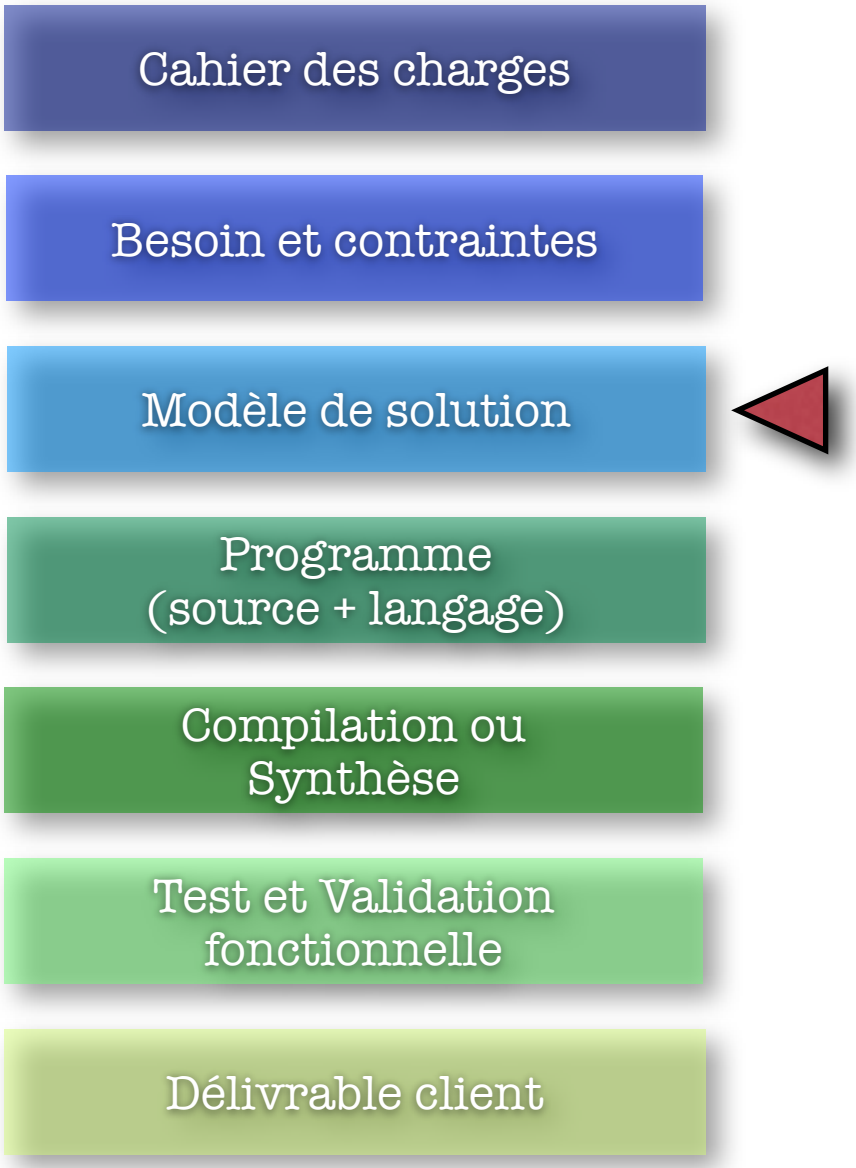
- La première étape est l'analyse du cahier des charges,
  - Analyse des besoins,
    - Que veut le client ?
    - Quelles fonctions sont à mettre en œuvre ?
  - Evaluation des contraintes imposées,
    - Vitesse d'exécution,
    - Consommation mémoire,
    - Architecture matérielle,

Le flot de conception en informatique



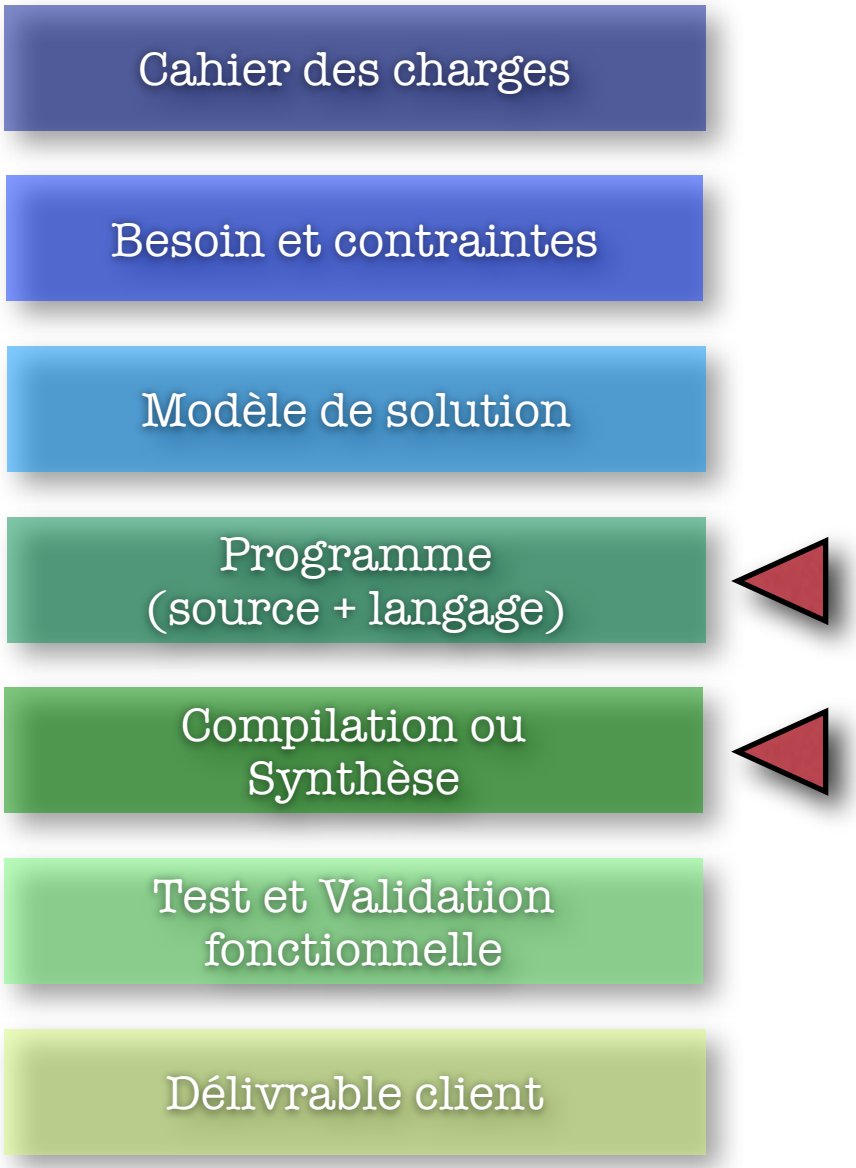
- Avant de coder, il est nécessaire de réfléchir !
- A l'aide d'*un papier et d'un crayon*, on va concevoir un modèle de solution,
  - Modèle fonctionnel : décomposition du problème en sous-fonctions élémentaires,
- Etape essentielle, une erreur à ce niveau là se paye très cher par la suite !

Le flot de conception en informatique



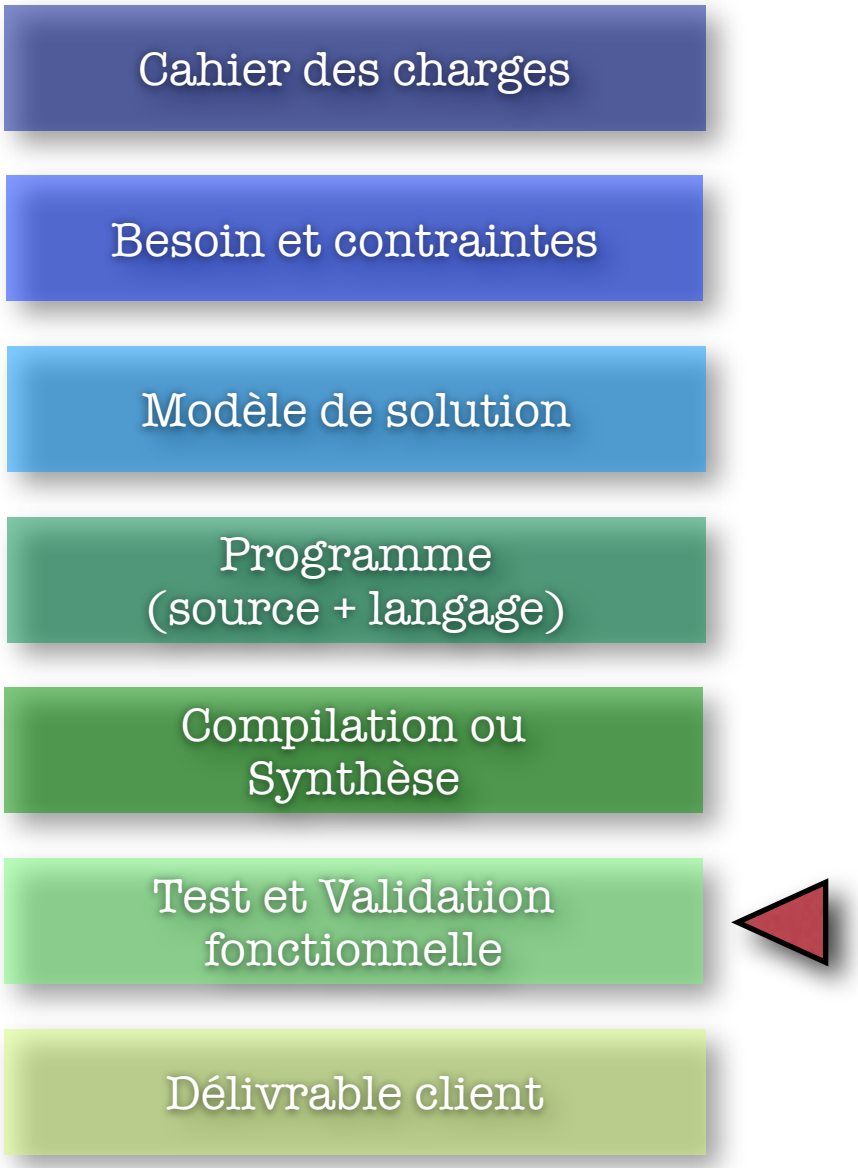
- Ensuite on choisit le langage adapté à ses besoins (*C*, *C++*, *Java*, *C#*, *PHP*, *ASP*, etc.),
- On ouvre son éditeur préféré et on développe le code nécessaire à l'application,
- Des lors on prévoit les procédure de test et de débogage pour la suite du processus de développement,

Le flot de conception en informatique



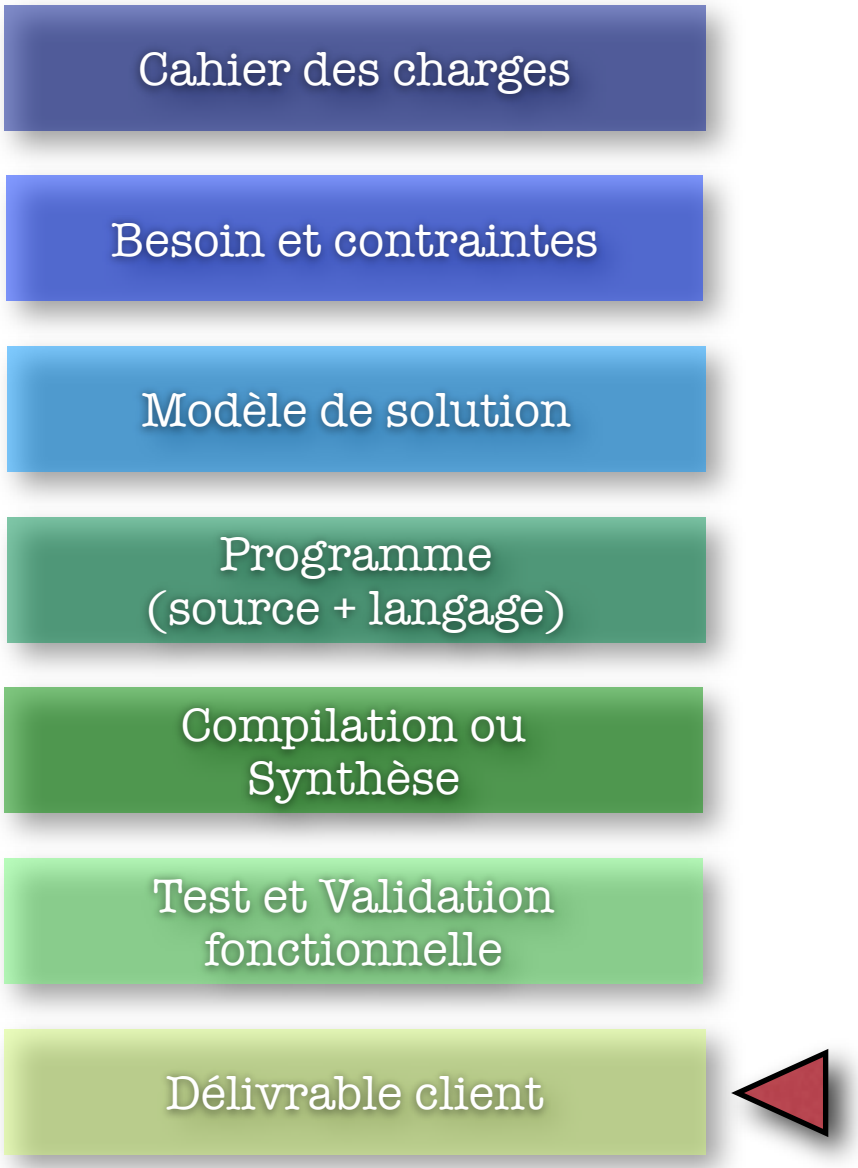
- Une fois le programme écrit on utilise un compilateur pour transformer le langage de haut niveau en code exécutable par l'architecture matérielle cible,
- Durant cette étape, il est possible d'interagir avec l'outil pour optimiser automatiquement certaines caractéristiques de l'application,
- L'outil permet de vérifier la *validité sémantique* de votre code, pas la *validité fonctionnelle*.
  - Compiler n'équivaut pas à fonctionner correctement !

Le flot de conception en informatique



- Cette étape est généralement la plus longue de tout le flot de conception,
- Il faut vérifier que les cas d'utilisation sont respectés par l'application,
  - Vérification aussi des contraintes d'implémentation,
- Les erreurs trouvées ici peuvent remettre en cause certains choix réalisés lors de l'élaboration de la solution.

Le flot de conception en informatique



- Une fois l'ensemble des tests passés avec succès, il faut alors packager l'application pour le client,
  - Regrouper les ressources nécessaires pour le déploiement,
  - Rédiger la documentation relative à l'outil,
- On peut alors penser à prendre des vacances !

Conception d'un programme

- La conception d'un programme est un processus itératif basé sur 4 étapes :
  1. **Editer** un programme avec son éditeur préféré,
  2. **Compiler** son programme (avec un compilateur objet),
  3. **Exécuter** son programme pour vérifier ...
  4. **Tester** et **débugger** ses erreurs (retourner en 1),
- *Cela peut être long, en fonction de la complexité du programme.*

3

“ Rappels sur le langage C ”

III. Rappels sur le langage C

- Les variables -

Les types de données élémentaires

- **Nombres entiers**
  - *int* (au minimum 16 bits, pour des valeurs 32767),
  - *long int* (min 32 bits),
  - *short*, *int*, *unsigned int* . . .
- **Nombres flottants**
  - *float*, *double* et *long double*
- **Caractères**
  - *char* (’a’,’b’,. . . ’A’,. . . ,’:’,. . .).

Les variables - Définition

- **Syntaxe : type v;**
  - **int** p ;
  - **double** x ;

- Toute variable doit être définie avant d'être utilisée !
  - Une définition peut apparaître n'importe ou dans un programme,
  - Une variable définie en dehors de toute fonction, est une variable globale.

Les variables - Initialisation

- Une variable peut être initialisée lors de sa déclaration. Deux notations sont alors possibles en *C++* (une seule notation existe en *C*) :

```
int    variable = -4;      int    variable(-4);
double nombre   = 2.3;     double nombre(2.3);
```

- Une variable d'un type élémentaire non initialisée peut contenir *n'importe quelle valeur*.
  - La majorité des compilateurs récents informent de ce genre de problèmes, mais ce n'est pas une raison...

Autres types de données…

- Dans les bibliothèques livrées avec la majorité des compilateurs, des types de données avancés sont disponibles pour aider les développeurs :
  - ***Le type string*** : ce type permet de modéliser les chaînes de caractères (tableau de char) et propose des fonctionnalités avancées pour les manipuler (comparaison, recherche, concaténation, ...).
  - ***Le type vecteur*** : ce type (ou cette classe) va permettre de remplacer les tableaux de données fournissant des services bien pratiques pour éviter les dépassements.
- *Nous détaillerons certains de ces types non primitifs à la fin du cours car ces « types » sont des objets.*

III. Rappels sur le langage C

- Les opérateurs -

Les opérateurs – Opérateurs de base

- Les opérateurs arithmétiques
  - +, -, ×, /, %(modulo)
- Les opérateurs de comparaison
  - < (inférieur), <= (inférieur ou égal), == (égal),
  - > (supérieur), >= (supérieur ou égal), != (différent)
- Les opérateurs logiques
  - && représente l'opérateur "ET"
  - || représente le "OU"
  - ! représente le "NON".

Les opérateurs - incrémentations (pré et post)

- Les opérateurs (++) et (--) permettent respectivement d'incrémenter et décrémenter une variable.

- Pré-incrémentation
  - « *t* = ++*var* », l'exécution est équivalente à :
  - (*var* = *var* + *1*) puis (*t* = *var*)

- Post-incrémentation
  - « *t* = *var* ++ » l'exécution est équivalente à :
  - (*t* = *var*) puis (*var* = *var* + *1*)

- *Hors affectation de variable, le comportement des 2 méthodes d'incrémentation est identique.*

III. Rappels sur le langage C

- Les fonctions -

Les fonctions - Définition

- *type* **nom**(*liste des paramètres*) { *corps* }
  - *type* est le type du résultat de la fonction.
    (*void* si il s'agit d'une procédure)
  - La *liste des paramètres* (paramètres formels):
    ($type_1$ p_1, ..., $type_n$ p_n)
  - Le *corps* décrit les instructions à effectuer.
    - Le corps utilise ses propres variables locales, les éventuelles variables globales et les paramètres formels.
  - Si une fonction renvoie un résultat, il doit y avoir (au moins) une instruction *« return expr »* ;
    - Dans le cas d'une procédure, on peut utiliser *« return; »* pour quitter la procédure

Les fonctions - Exemples

- *type* **nom(** *liste des paramètres* **)** { *corps* }

```
void afficheNombre( int nombre ){
     printf("Le nombre est : %d\n", nombre);
     return;
}

int max( int a, int b ){
     if( a > b ) return a;
     return b;
}
```

Les fonctions – Le mot clef « inline »

- Le mot clef « inline » permet de mettre en ligne la fonction dans le code exécutable lorsque cette dernière est appelée,
  - Gain en temps d'exécution (pas de branchement du PC),
    - Le gain est relatif en fonction des défauts de cache et des autres paramètres architecturaux de la cible,
  - Augmentation de l'occupation mémoire car le code ASM de la fonction est dupliqué (rupture de pipeline),

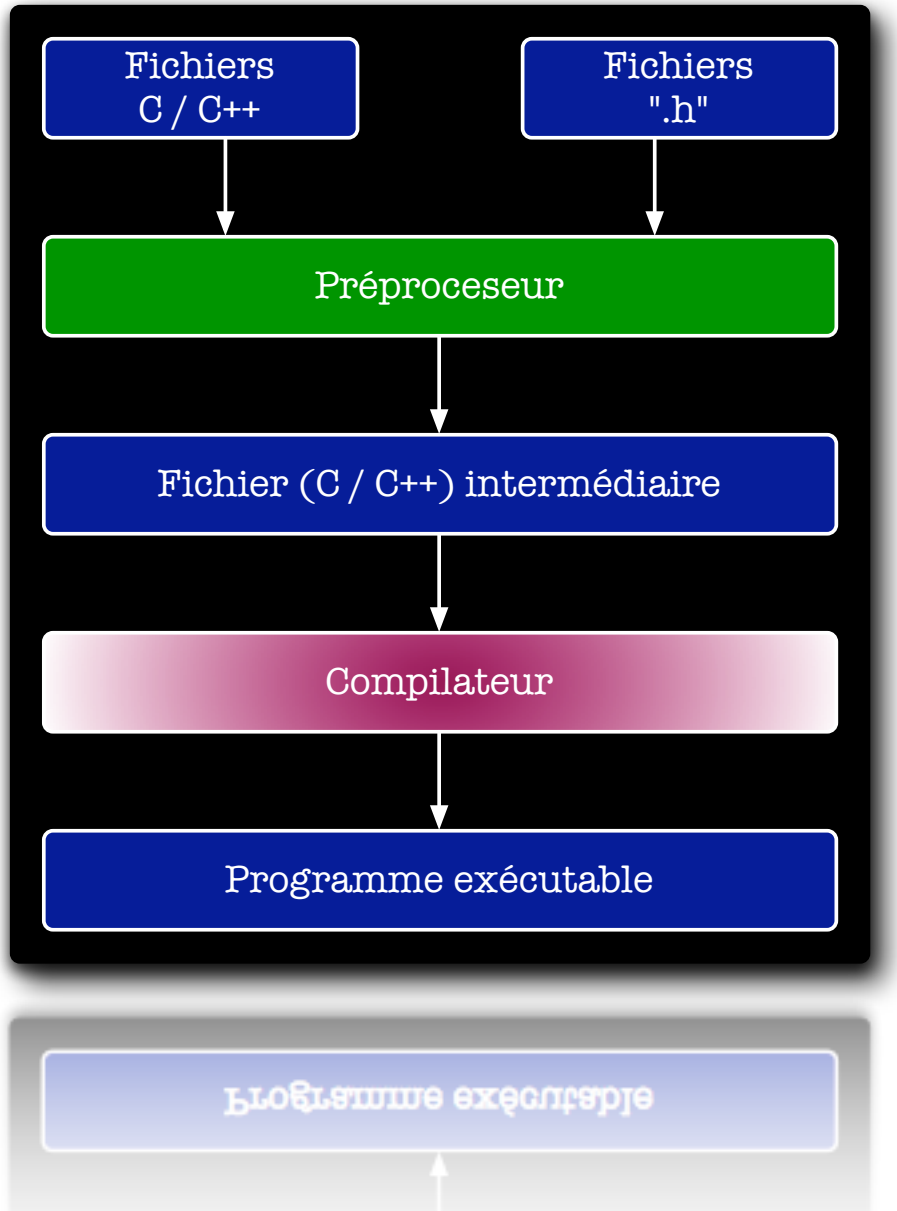
```
inline int max( int a, int b ){
  if( a > b ) return a;
  return b;
}
```

III. Rappels sur le langage C

~ Le préprocesseur ~

Explication du flot de « compilation »

- Un fichier « C/C++ » n'est pas immédiatement traduit en code exécutable,
- Le *préprocesseur* est un programme qui analyse votre fichier et qui lui fait subir certaines transformations,
  - *Inclusion d'un fichier,*
  - *Suppression* ou remplacement de zones de texte,
- Le préprocesseur effectue ces opérations séquentiellement,
- Appelé automatiquement par le compilateur, avant la « compilation ».



Interagir avec le préprocesseur

- Toute commande du préprocesseur commence par un caractère dièse « # » en début de ligne,

- Inclusion de fichiers

  - #include <stdio.h>

    - On demande au préprocesseur d'insérer le contenu du fichier « stdio.h » dans les sources courantes,

- Déclaration de constantes

  - #define Pi 3.141596

    - On demande au préprocesseur de remplacer le texte « Pi » contenu dans le code source sa vraie valeur numérique.

Interagir avec le préprocesseur

- Les constantes propres au préprocesseur :
  - __FILE__
    - Cette constante contient le nom du fichier en cours de traitement (char*),
  - __LINE__
    - Cette constante contient le numéro de la ligne en cours de traitement (int),
  - __DATE__
    - Cette constante contient la date du jour actuel (char*),
  - __TIME__
    - Cette constante contient l'heure actuelle (char*),

- Exemples

```
printf("Fichier compilé le %s à %s\n", __DATE__, __TIME__);
printf("Erreur détectée dans le fichier %s\n", __FILE__);
printf("Ce printf se trouve à la ligne %d\n", __LINE__);
```

Interagir avec le préprocesseur

- La définition des identificateurs et des constantes de compilation est très utilisée pour effectuer ce que l'on appelle la *compilation conditionnelle*,
  - La compilation conditionnelle consiste à remplacer certaines portions du code source par d'autres, en fonction valeur de certaines constantes,
- Liste des instructions conditionnelles
  - **#ifndef** (if not defined ...)
  - **#elif** (sinon, si ...)
  - **#if** (si ...)
  - **#endif**

Interagir avec le préprocesseur

- Gestion du préprocesseur lors de la phase de compilation :
  - ➡ *gcc -DWindows main.c - o mon_programme*
  - ➡ *gcc -DLinux main.c - o mon_programme*

```
#ifndef Windows
    printf("Utilisation de l'API Linux");
    // Création d'une fenetre a la sauce Linux
#elif
    printf("Utilisation de l'API Win32");
    // Création d'une fenetre a la sauce Windows
#endif
```

Les Macros ou l'utilisation du préprocesseur

- Les macros sont des « *pseudo-fonctions* » qui vont être traitées par le *préprocesseur* lors de la compilation.

  ✔ Leur utilisation permet de simplifier le code ou de le rendre plus « flexible / générique »,

  ✘ Impossibilité de réaliser un débogage des macros (pas de pas à pas),

```
#define max(a, b) (a>b)?a:b;
… … … …
int c = max(2, 4);
int d = max(-2.6, 43.9);
```

Les Macros ou l'utilisation du préprocesseur

- Utilisées afin de rendre actives ou inactives certaines portions du code source en fonction d'une déclaration,
  - Utilisé dans le cadre de codes dédiés au débogage,
  - Utilisé afin de générer des applications dédiées à plusieurs systèmes d'exploitation (API différentes).

```
#define debug
int fonction( … ){
    #ifdef debug
        printf("Passage dans la fonction xxxx\n");
    #endif
    … … … … …
    return 1;
}
```

III. Rappels sur le langage C

~ Les pointeurs ~

Les pointeurs - Piqûre de rappel

- Les *pointeurs* sont des *variables* qui ne contiennent pas une donnée, mais une *adresse mémoire* pointant sur une *donnée*,

- Les pointeurs sont utilisés *naturellement* lorsque l'on manipule des *structures*, des *tableaux*, des *objets*, etc.

Les pointeurs - Notations

- *En fonction de la déclaration des variables, leur signification et leur usage va différer,*
- *Cette flexibilité du langage a pour contrepartie, les connaissances nécessaires minimums que vous devez posséder.*

```
int a;
// a  est une donnée,
// &a est une adresse,

int *b;
//  b est une adresse,
// *b est une donnée,

int tab[16];
//  tab    est une adresse,
//  tab[i] est une donnée,
// *tab    est une donnée (tab[0])
```

Les pointeurs – Exemple illustratif

```
int a = 13;
int b = -5;
int *p = Null;
p = &b;
(*p) = 11;
int tab[2] = [8, 2];
p = tab;
*p = 0;
 p = p + 1;
*p = 1;
 p = p - 2;
*p = -1;
 a = tab[0]
```

Adresse	Contenu
0x00	(a)
0x01	(b)
0x02	(p)
0x03	(tab)
0x04	tab[0]
0x05	tab[1]
0x06	???
0x07	???
0x08	???
0x09	???
0x0A	???
0x0B	???

Les pointeurs – Exemple illustratif

```
int a = 13;
int b = -5;
int *p = Null;
p = &b;
(*p) = 11;
int tab[2] = [8, 2];
p = tab;
*p = 0;
 p = p + 1;
*p = 1;
 p = p - 2;
*p = -1;
 a = tab[0]
```

Adresse	Contenu
0x00	13 (a)
0x01	(b)
0x02	(p)
0x03	(tab)
0x04	tab[0]
0x05	tab[1]
0x06	???
0x07	???
0x08	???
0x09	???
0x0A	???
0x0B	???

Les pointeurs – Exemple illustratif

```
int a = 13;
int b = -5;
int *p = Null;
p = &b;
(*p) = 11;
int tab[2] = [8, 2];
p = tab;
*p = 0;
 p = p + 1;
*p = 1;
 p = p - 2;
*p = -1;
 a = tab[0]
```

Adresse	Contenu
0x00	13 (a)
0x01	-5 (b)
0x02	(p)
0x03	(tab)
0x04	tab[0]
0x05	tab[1]
0x06	???
0x07	???
0x08	???
0x09	???
0x0A	???
0x0B	???

Les pointeurs – Exemple illustratif

```
int a = 13;
int b = -5;
int *p = Null;
p = &b;
(*p) = 11;
int tab[2] = [8, 2];
p = tab;
*p = 0;
 p = p + 1;
*p = 1;
 p = p - 2;
*p = -1;
 a = tab[0]
```

Adresse	Contenu
0x00	13 (a)
0x01	-5 (b)
0x02	0 (p)
0x03	(tab)
0x04	tab[0]
0x05	tab[1]
0x06	???
0x07	???
0x08	???
0x09	???
0x0A	???
0x0B	???

Les pointeurs – Exemple illustratif

```
int a = 13;
int b = -5;
int *p = Null;
p = &b;
(*p) = 11;
int tab[2] = [8, 2];
p = tab;
*p = 0;
 p = p + 1;
*p = 1;
 p = p - 2;
*p = -1;
 a = tab[0]
```

Adresse	Contenu
0x00	13 (a)
0x01	-5 (b)
0x02	0x01 (p)
0x03	(tab)
0x04	tab[0]
0x05	tab[1]
0x06	???
0x07	???
0x08	???
0x09	???
0x0A	???
0x0B	???

Les pointeurs – Exemple illustratif

```
int a = 13;
int b = -5;
int *p = Null;
p = &b;
(*p) = 11;
int tab[2] = [8, 2];
p = tab;
*p = 0;
 p = p + 1;
*p = 1;
 p = p - 2;
*p = -1;
 a = tab[0]
```

Adresse	Contenu
0x00	13 (a)
0x01	11 (b)
0x02	0x01 (p)
0x03	(tab)
0x04	tab[0]
0x05	tab[1]
0x06	???
0x07	???
0x08	???
0x09	???
0x0A	???
0x0B	???

Les pointeurs – Exemple illustratif

```
int a = 13;
int b = -5;
int *p = Null;
p = &b;
(*p) = 11;
int tab[2] = [8, 2];
p = tab;
*p = 0;
 p = p + 1;
*p = 1;
 p = p - 2;
*p = -1;
 a = tab[0]
```

Adresse	Contenu
0x00	13 (a)
0x01	11 (b)
0x02	0x01 (p)
0x03	0x04 (tab)
0x04	8 tab[0]
0x05	2 tab[1]
0x06	???
0x07	???
0x08	???
0x09	???
0x0A	???
0x0B	???

Les pointeurs – Exemple illustratif

```
int a = 13;
int b = -5;
int *p = Null;
p = &b;
(*p) = 11;
int tab[2] = [8, 2];
p = tab;
*p = 0;
 p = p + 1;
*p = 1;
 p = p - 2;
*p = -1;
 a = tab[0]
```

Adresse	Contenu
0x00	13 (a)
0x01	11 (b)
0x02	0x04 (p)
0x03	0x04 (tab)
0x04	8 tab[0]
0x05	2 tab[1]
0x06	???
0x07	???
0x08	???
0x09	???
0x0A	???
0x0B	???

Les pointeurs – Exemple illustratif

```
int a = 13;
int b = -5;
int *p = Null;
p = &b;
(*p) = 11;
int tab[2] = [8, 2];
p = tab;
*p = 0;
 p = p + 1;
*p = 1;
 p = p - 2;
*p = -1;
 a = tab[0]
```

Adresse	Contenu
0x00	13 (a)
0x01	11 (b)
0x02	0x04 (p)
0x03	0x04 (tab)
0x04	0 tab[0]
0x05	2 tab[1]
0x06	???
0x07	???
0x08	???
0x09	???
0x0A	???
0x0B	???

Les pointeurs – Exemple illustratif

```
int a = 13;
int b = -5;
int *p = Null;
p = &b;
(*p) = 11;
int tab[2] = [8, 2];
p = tab;
*p = 0;
 p = p + 1;
*p = 1;
 p = p - 2;
*p = -1;
 a = tab[0]
```

Adresse	Contenu
0x00	13 (a)
0x01	11 (b)
0x02	0x05 (p)
0x03	0x04 (tab)
0x04	0 tab[0]
0x05	2 tab[1]
0x06	???
0x07	???
0x08	???
0x09	???
0x0A	???
0x0B	???

Les pointeurs – Exemple illustratif

```
int a = 13;
int b = -5;
int *p = Null;
p = &b;
(*p) = 11;
int tab[2] = [8, 2];
p = tab;
*p = 0;
 p = p + 1;
*p = 1;
 p = p - 2;
*p = -1;
 a = tab[0]
```

Adresse	Contenu
0x00	13 (a)
0x01	11 (b)
0x02	0x05 (p)
0x03	0x04 (tab)
0x04	0 tab[0]
0x05	1 tab[1]
0x06	???
0x07	???
0x08	???
0x09	???
0x0A	???
0x0B	???

Les pointeurs – Exemple illustratif

```
int a = 13;
int b = -5;
int *p = Null;
p = &b;
(*p) = 11;
int tab[2] = [8, 2];
p = tab;
*p = 0;
 p = p + 1;
*p = 1;
 p = p - 2;
*p = -1;
 a = tab[0]
```

Adresse	Contenu
0x00	13 (a)
0x01	11 (b)
0x02	0x03 (p)
0x03	0x04 (tab)
0x04	0 tab[0]
0x05	1 tab[1]
0x06	???
0x07	???
0x08	???
0x09	???
0x0A	???
0x0B	???

Les pointeurs – Exemple illustratif

```
int a = 13;
int b = -5;
int *p = Null;
p = &b;
(*p) = 11;
int tab[2] = [8, 2];
p = tab;
*p = 0;
 p = p + 1;
*p = 1;
 p = p - 2;
*p = -1;
 a = tab[0]
```

Adresse	Contenu
0x00	13 (a)
0x01	11 (b)
0x02	0x03 (p)
0x03	0xFF (tab)
0x04	~~0 tab[0]~~
0x05	~~1 tab[1]~~
0x06	???
0x07	???
0x08	???
0x09	???
0x0A	???
0x0B	???

Les pointeurs – Exemple illustratif

```
int a = 13;
int b = -5;
int *p = Null;
p = &b;
(*p) = 11;
int tab[2] = [8, 2];
p = tab;
*p = 0;
 p = p + 1;
*p = 1;
 p = p - 2;
*p = -1;
 a = tab[0]
```

Adresse	Contenu
0x00	??? (a)
0x01	11 (b)
0x02	0x03 (p)
0x03	0xFF (tab)
0x04	~~0 tab[0]~~
0x05	~~1 tab[1]~~
0x06	???
0x07	???
0x08	???
0x09	???
0x0A	???
0x0B	???

Mise en pratique de vos connaissances !

- Ecrivez une fonction nommée *swap(...)* et qui permet d'échanger le contenu de *deux variables* entières nommées *« x »* et *« y » dans le "main"*.

Réponse typique mais erronée !

```
void swap(int a, int b){
    int c = a;
    a = b;
    b = c;
}

int main ( void )
    int x = 10, y =20;
    swap(x, y);
    printf( … … … );
}
```

- Cette solution est compilable et s'exécute sans difficulté,
  - ↳ Fonctionnellement incorrecte !
- On copie dans « a » et « b » les valeurs 10 et 20. Dans la fonction, nous n'intervertissons que les copies de « a » et « b ».
- *Nécessité de passer par des pointeurs pour modifier les adresses !*

La bonne réponse…

```
void swap(int *a, int *b){
    int c = *a;
    *a = *b;
    *b = c;
}

int main ( void ){
  int x = 10, y =20;
  swap(&x, &y);
  printf( … … … );
}
```

- Cette solution est compilable et s'exécute sans difficulté,
  - Fonctionnellement correct !
- Les pointeurs « a » et « b » visent les données « x » et « y » et non des copies de ces dernières.

Les pointeurs - Gain en performance

- En fonction de votre style d'écriture d'un programme, le compilateur *C* ou *C++* ne va pas générer un code binaire identique !
  - Certaines écritures sont préférables à d'autres en terme de performance d'exécution,
- Les pointeurs permettent d'accéder rapidement à des zones mémoires linéaires,
  - L'inconvénient vient de la maîtrise nécessaire des pointeurs que doit posséder le concepteur !
  - Dans certains cas, il est aussi nécessaire de maîtriser la taille de ses données…

Les pointeurs - Gain en performance

Code source C

```
int tableau[256];
…
for(i=0; i<256; i++){
    sum += tableau[i];
}
```

Morceau de code C réalisant la somme des données contenues dans un tableau.

Equivalence avec pointeurs

```
int tableau[256];
int *p;
p = tableau;
…
for(i=0; i<256; i++){
    sum += *(p++);
}
```

Code source équivalant réalisant la somme des données mais en utilisant un pointeur

Les pointeurs - Gain en performance

Code source C

```
int tableau[256];
…
int *adr;
for(i=0; i<256; i++){
    adr = tableau + i;
    sum += * (adr);
}
```

Adr = 256 +

sum = 256 +

Cpt "i" = 256 +

Equivalence avec pointeurs

```
int tableau[256];
int *p   = tableau;
int *max = tableau + 256;
…
while(p < max){
    sum += (*p);
    p += 1;
}
```

p = 256 +

sum = 256 +

Les pointeurs - Gain en performance

Code source C

```
int image[256][256];
…
int adr;
for(x=0; x<256; x++){
     for(y=0; y<256; y++){
          // adr = tableau + (y*256) + x;
          // sum += (*adr);
          sum += image[y][x];
     }
}
```

x = 65535 (+)
y = 65535 (+)
adr = 131070 (+),
65635 (x)
Sum = 65535 (+)

Equivalence avec pointeurs

```
int image[256][256];
int *adr = image;
int *max = image + (256*256);
while(adr < max){
     sum += (*adr);
     adr += 1;
}
```

max = 1 (+), 1 (x)
adr = 65535 (+)
Sum = 65535 (+)

III. Rappels sur le langage C

- L'allocation mémoire -

Allocation et libération - Introduction

- Lorsque la taille des données n'est pas connue lors de la conception, 2 solutions s'offrent au développeur :

1. Réserver de manière statique un espace mémoire supérieur aux besoins maximums que l'on peut rencontrer,
   - Risque d'*erreur à la conception* lors du calcul de la mémoire maximum à réserver (*buffer over flow*, *buffer over run*)
   - *Code peu réutilisable* dans d'autres projets / applications car il y a un risque que les hypothèses réalisées soient dans ce nouveau cas, erronées,
   - *Consommation excessive de mémoire* lorsque les données ne sont pas en configuration pire cas,
2. Allouer de manière dynamique l'espace mémoire nécessaire en fonction des besoins réels,
   - *Manipulation de pointeurs obligatoire*, risque d'erreur de conception…
   - Risque *d'oubli de désallocation* des espaces mémoires (fuites mémoires),

Allocation et libération - Fonctions de base

- Méthodes disponibles dans la bibliothèque standard « *stdlib* » :

  - ***pointeur = malloc(taille);***
    - « malloc » permet d'allouer un espace mémoire dont la taille est fourni en paramètre. L'espace mémoire n'est pas initialisé.

  - ***void free(pointeur);***
    - « free » permet de libérer une zone mémoire à partir du pointeur sur cette zone.

  - ***pointeur = calloc(nb_element, taille_element);***
    - « calloc » permet d'allouer de la mémoire comme le fait la fonction « malloc » et l'espace mémoire est initialisé à zéro.

  - ***Ptr = realloc(pointeur, new_size);***
    - La fonction « realloc » permet de redimensionner un espace mémoire déjà alloué à une taille différente. La recopie des données est automatique, ainsi que la désallocation de la première zone.

Allocation et libération - Fonctions de base

- Dans la partie proprement objets du cours nous utiliserons les opérateurs « ***new*** » et « ***delete*** » qui permettent d'allouer des objets en mémoire.

```
int main( void ){
    int *tableau;
    tableau = new int[2000];
    delete tableau;

    tableau = (int*)malloc(2000 * sizeof(int));
    free tableau;

    tableau = (int*)calloc(2000, sizeof(int));
    free tableau;

    return 0;
}
```

III. Rappels sur le langage C

- Les structures de données -

Les structures de données - Définition

- Afin de simplifier le stockage des informations, il est possible de définir des ensembles de données nommés « *structures* »
- Pour cela, on utilise le mot clé « *struct* ». Sa syntaxe est la suivante :

```
struct
nom_structure {
      type_1 champ_1;
      type_2 champ_2;
     … … … … … …
};
```

Les structures de données - Définition

- Dans la définition suivante nous déclarons une structure dont le nom est "***Chien***" et nous déclarons une variable nommée "***Cesar***" de type "***Chien***"

```
struct Chien
{
    unsigned char  Age;
    unsigned char  Taille;
    char*  Race;
} Cesar;
```

Les structures de données - Définition

```
/* Instanciation statique */
struct ma_struc{ … }
ma_struc variable;
variable.champ_1 = 10;
```

La durée de vie de la structure dépend de la prochaine accolade fermante

La durée de vie de la structure dépend de la position de sa libération (free)

```
/* Instanciation dynamique */
ma_struc *variable;
variable = (ma_struc*) calloc(1, sizeof(ma_struc));
variable→champ_1 = 10;
(*variable).champ_2 = 10;
free( variable );
```

Les structures de données – Les unions

- Les unions constituent un type de structure particulier,
- Elles sont déclarées avec le mot clé « union », qui a la même syntaxe que « struct »,
- La différence entre les structures et les unions est que les différents champs d'une union occupent le même espace mémoire,
  - On ne peut donc, à tout instant, n'utiliser qu'un des champs de l'union.

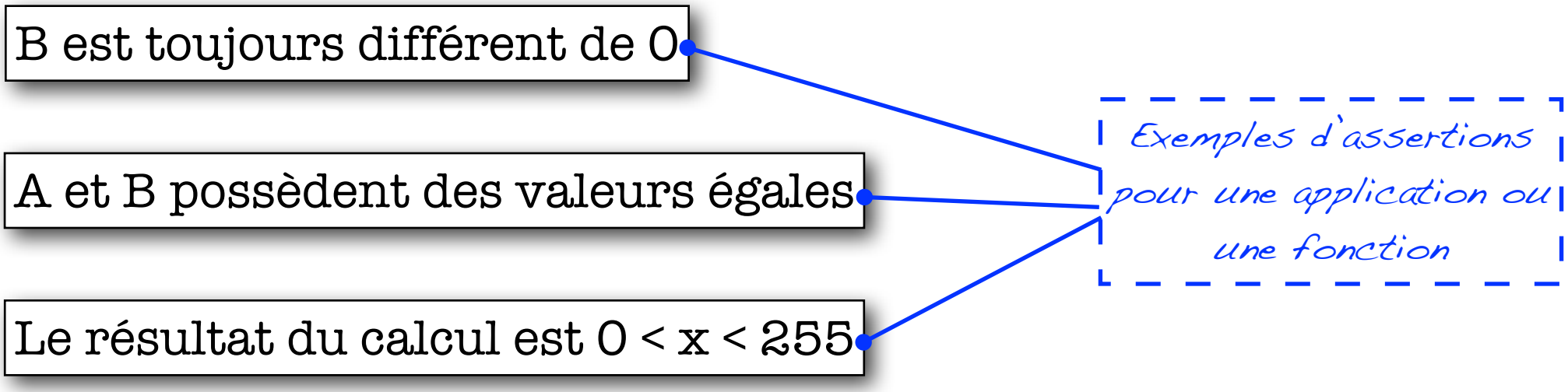
```
union A_ou_B
{
    int a;
    float b;
};

A_ou_B donnees;
donnees.a = 10;
… … … … … …
… … … … … …
donnees.b = 2.4;
// destruction de la
// valeur de "a"
```

III. Rappels sur le langage C

- Les assertions -

Les assertions - Qu'est ce que c'est ?

- Les mécanismes d'assertion permettent de vérifier que des hypothèses émises lors de la conception d'un système sont toujours respectées lors de son exécution,
  - Méthodes de vérification provenant du développement logiciel (assertions natives en C, C ++, Java, ...),
  - Traduction de assertion : affirmation ou hypothèse



Les assertions : hypothèses d'utilisation

- Les assertions permettent de valider le comportement de l'application lors de l'exécution et de gérer les erreurs dynamiques,
  - Une erreur stoppe l'exécution du programme avec indication de la cause de l'erreur + localisation,
  - Les assertions sont utilisées uniquement dans le cadre du développement et de la mise au point : l'utilisateur n'a pas besoin d'observer les problèmes qui ne le concernent pas,
- Elles disparaissent à la compilation lorsque l'on utilise les optimisations du compilateur,
- Outil indispensable lors des phases de conception (inadapté autrement).
- Des mécanismes de gestions d'assertions sont disponibles dans la majorité des langages de conception matériel et logiciel (*VHDL*, *SystemC*, *Java*, *C++*, *C*, etc.).

Mise en oeuvre des assertions en C/C++

```
#include <iostream>
#include <stdio.h>
#include <assert.h>

int ma_fonction(int a, int b){
    printf("Lancement de la division...\n");
    assert(b != 0);
    int c = a/b;
    printf("Fin de la division...\n");
   return c;
}

int main (int argc, char * const argv[]) {
    ma_fonction(2, 0);
    return 0;
}
```

Voici un exemple de code source dans lequel nous avons introduit une assertion afin de réaliser une hypothèse de fonctionnement.

Exemple d'assertions en C/C++

Compilation normale du programme à l'aide de GCC :

gcc mon_prog.c -o mon_prog

```
int ma_fonction(int a, int b){
    printf("Lancement de la division...\n");
    assert(b != 0);
    int c = a/b;
    printf("Fin de la division...\n");
  return c;
}
```

ma_fonction(2, 0);

```
The Debugger has exited with status 0.
[Session started at 2007-12-12 15:12:05 +0100.]
Lancement de la division...
Assertion failed: (b != 0), function ma_fonction, file
      /Users/legal/XCode/Essais_code/main.cpp, line 32.

The Debugger has exited due to signal 6 (SIGABRT).
```

Exécution du binaire :

./mon_prog

Exemple d'assertions en C/C++

Compilation normale du programme à l'aide de GCC :

```
gcc mon_prog.c -o mon_prog
```

```
int ma_fonction(int a, int b){
    printf("Lancement de la division...\n");
    assert(b != 0);
    int c = a/b;
    printf("Fin de la division...\n");
  return c;
}
```

ma_fonction(2, 2);

```
The Debugger has exited with status 0.
[Session started at 2007-12-12 15:12:04 +0100.]
Lancement de la division...
Fin de la division...
```

Exécution du binaire :

```
./mon_prog
```

Exemple d'assertions en C/C++

Compilation avec désactivation des assertions à l'aide de GCC :

```
gcc mon_prog.c -o mon_prog -NDEBUG
```

```
int ma_fonction(int a, int b){
    printf("Lancement de la division...\n");
    assert(b != 0);
    int c = a/b;
    printf("Fin de la division...\n");
  return c;
}
```

ma_fonction(2, 0);

```
[Session started at 2007-12-12 18:01:16 +0100.]
Lancement de la division...
Fin de la division...

The Debugger has exited due to signal 8 (SIGFPE).
```

Exécution du binaire :

./mon_prog

=> Plantage !

Les assertions d'implémentation

- Elles sont spécifiées par le concepteur (designer)
  - ➜ Permettent d'encoder formellement les hypothèses faites durant la phase de conception des composants (le concepteur précise par exemple que le diviseur doit toujours être différent de la valeur 0),
  - ➜ Elles ne permettent pas de détecter des divergences entre la spécification et l'implémentation (ne sont pas faites pour ça).
- Hypothèses de fonctionnement
  - ➜ La taille de la matrice dont on doit calculer le déterminant ne peut jamais être (0x0) ni (1x1),
  - ➜ La chaîne de caractères contenant le nom de l'utilisateur a une taille comprise entre 0 < 255 (utilisation d'une allocation statique),
  - ➜ La taille du fichier que l'application va charger en mémoire cache est toujours inférieure à 2 méga-octets.

Les assertions de spécification

- Elles sont spécifiées par l'équipe de vérification
  - ➜ Permettent d'encoder formellement les hypothèses et les contraintes exprimées dans les spécifications comme par exemple :
    - Latence maximum d'un circuit, cadence de production des sorties,
    - Ordre de validation des signaux lors d'un Hand-Shake, etc.
    - La dynamique des données présentes en entrée / sortie du système,
  - ➜ Elles visent à détecter les erreurs fonctionnelles présentes dans le circuit conçu,

- Hypothèses du cahier des charges
  - ➜ Le circuit doit produire une données tous les 1/24 de secondes afin de respecter la cadence du système,
  - ➜ Si le composant gérant l'ABS reçoit un front montant des freins, il doit au maximum 10 cycles après transmettre l'ordre adéquate aux commandes des disques.

4

“ Introduction aux Objets ”

IV. Introduction aux objets

~ Les apports du langage C++ ~

Les limites du langage C

✔ Avantages

- Langage assez simple à apprendre,
- Bonne relation entre le code « C » et le code assembleur généré,
- Flexibilité sémantique (on écrit un peu ce que l'on veut avec des parties optimisées en assembleur si on le désire),
- ***Vitesse d'exécution rapide*** et maîtrise du coût mémoire des données manipulées,

✘ Inconvénients

- Mode d'écriture ***peu rigoureux*****,**
- Gestion de l'imbrication des structures,
- ***Dissociation*** trop importante des données et des méthodes de traitement associées,
- ***Faible maintenabilité*** et évolutivité des projets complexes (environnement industriel),

Apports fonctionnels

- La gestion des flux d'entrée et de sortie a été revue dans le langage C++ afin de simplifier les opérations d'E/S,
- De nouveaux types de données compris nativement dans le langage (booléen).

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

int main ( ) {
    cout << "hello world !" << endl;
}
```

Apports fonctionnels – Les constantes

- Les habitués du C ont l'habitude d'utiliser la directive du préprocesseur *#define* pour définir des constantes.
  - L'utilisation du préprocesseur est une source d'erreurs difficiles à détecter.
- En C++, l'utilisation du préprocesseur se limite aux cas les plus sûrs :
  - Inclusion de fichiers,
  - Compilation conditionnelle.
- Le mot réservé "*const*" permet de définir une constante.
  - L'objet ainsi spécifié ne pourra pas être modifié durant toute sa durée de vie. Il est indispensable d'initialiser la constante au moment de sa définition.

Apports fonctionnels – Les constantes

- Toute *tentative d'écriture* dans une donnée déclarée comme étant une constante sera mentionnée comme une *erreur par le compilateur*.

```
const int N     = 10; // N est un entier constant.
const int MOIS  = 12;
const double Pi = 3.141596…;
int tab[2 * N];    // autorisé en C++ (interdit en C)
```

Les constantes dans les méthodes

- L'utilisation du mot clef "const" lors de la déclaration d'une fonction permet de spécifier que le paramètre *ne doit pas être modifié* dans cette dite fonction.

```
void MaFonction(const int variable){
  int data = variable // OK
  variable = 2        // ERREUR
}
```

Le compilateur vous signalera une erreur à la ligne ou vous réalisez l'affectation.

Apport fonctionnels - Les espaces de nommage

- Le “C” est un langage pour lequel il est impossible de dénombrer les bibliothèques existantes,
  - Parfois les auteurs de ces bibliothèques (puisqu'ils ne se consultent pas) appellent leurs fonctions de la même façon même si elles réalisent des traitements différents,
    - Cela pose problème lorsque quelqu'un essaie d'utiliser deux bibliothèques “incompatibles” en même temps,
    - A l'édition des liens, il est impossible pour le “linker” de distinguer une fonction de l'autre, ce qui pose problème,
  - Le « C » ne propose aucune solution à ce type de problème; il faut impérativement modifier le code de l'une des bibliothèque (perte de la compatibilité avec les versions supérieures).

Apport fonctionnels - Les espaces de nommage

- Les “espaces de nommage” ont été défini afin de permettre une surcharge des données et méthodes « *globales* »,

- **namespace** (signifiant espace de noms) est le mot clef **C++** qui permet de sectoriser les fonctions. Les noms des fonctions vont être assignés comme dans une boîte,

  - Et si on crée une deuxième boîte, on pourra y mettre une fonction sans ce soucier de sa présence dans une autre boîte.

```
namespace A{
     int b;
}

namespace B{
     int b;
}

int main( void ){
     A::b = 10;
     B::b = 20
     return 0;
}
```

Apport fonctionnels - Les espaces de nommage

- L'utilisation des "espaces de nommage" peut rapidement devenir fatiguant :
  - Il faut mettre « espace:: » avant toutes ses variables,
- Le langage **C++** fournit le mot clef ***using***,
- En faisant: ***« using namespace A »*** toutes les variables, dont on ne précise pas le ***namespace***, seront considérées comme faisant parti de A.
  - Simplification de l'écriture,

```
namespace A{
    int b;
}

namespace B{
    int b;
}

using namespace A
int main( void ){
    // équivalent a A::b=10
    b = 10;
    B::b = 20
    return 0;
}
```

Apport fonctionnels - Les espaces de nommage

- Un ***namespace*** *« anonyme »* est un espace de nommage qui ne possède pas d'identifiant.

- Un ***namespace*** *« anonyme »* a la particularité de n'être visible que par l'unité de compilation dans laquelle il se trouve (c'est à dire le fichier compilable).

  - Son utilité est de permettre la déclaration d'une variable, d'une fonction ou d'un type dont la portée doit être celle du fichier.

```
namespace {
      int b;
}

namespace B{
      int b;
}

int main( void ){
    // Espace anonyme
    b = 10;
    B::b = 20
    return 0;
}
```

IV. Introduction aux objets

- Autres différences -

Autres différences

- Quelques incompatibilités sémantiques liées à la définition et au nommage des structures,
- Typage des données plus important qu'avec certains vieux compilateurs C,
- Présence de nouveau mots-clef qui n'existaient pas dans le langage C,
- Nouvelles méthodes permettant d'allouer de la mémoire dynamiquement.

> ➢ *Un code C n'est pas nécessairement compilable avec un compilateur C++ récent sans modification de son code source !*

IV. Introduction aux objets

- Introduction aux objets -

Introduction à la notion d'objets

- Jusqu'à maintenant vous avez utilisés des approches orientées « structures & fonctions »
  - Découpage des applications en un ensemble de structures de données et de fonctions pour les manipuler,
- L'approche objet considère comme indissociable les données et les méthodes pour les manipuler,
- Un objet peut être imaginé / représenté comme une boite noire implémentant une fonctionnalité et rendant un service,
  - Décorrélation de la *notion de service rendu* vis-à-vis de la réalisation de cette fonction.

Introduction à la notion d'objets

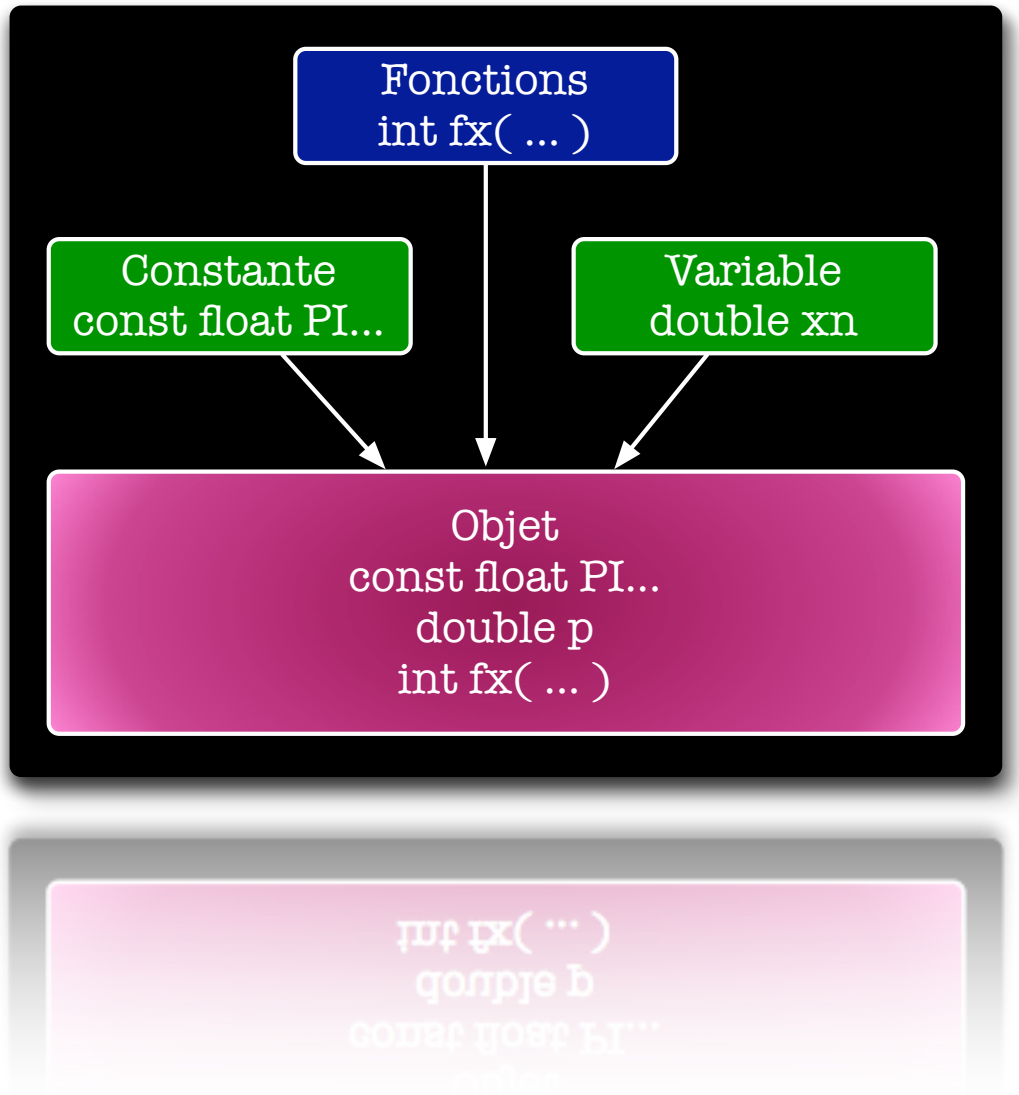
Développement fonctionnel

```
/* Données */
File *fichier;

/* Manipulateurs */
int fopen( … );
int read( … );
int write( … );
int close( … );

// Les fonctions ne sont pas
// associées directement a la
// structure "File"
```

Développement objets

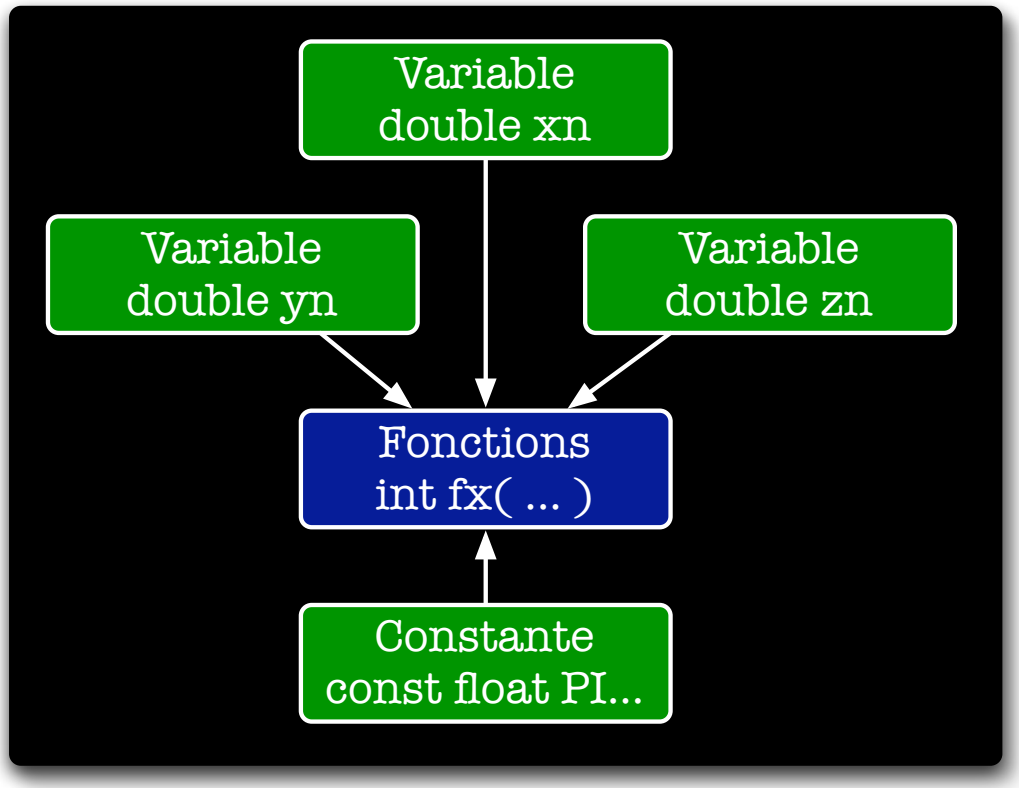
```
class fichier{
  /* services proposés */
  int open( … );
  int read( … );
  int write( … );
  int close( … );
}

// Les fonctions appartiennent
// directement a l'objet fichier, si
// vous créer un "fichier" ces
// services sont directement inclus.
```

Introduction à la notion d'objets



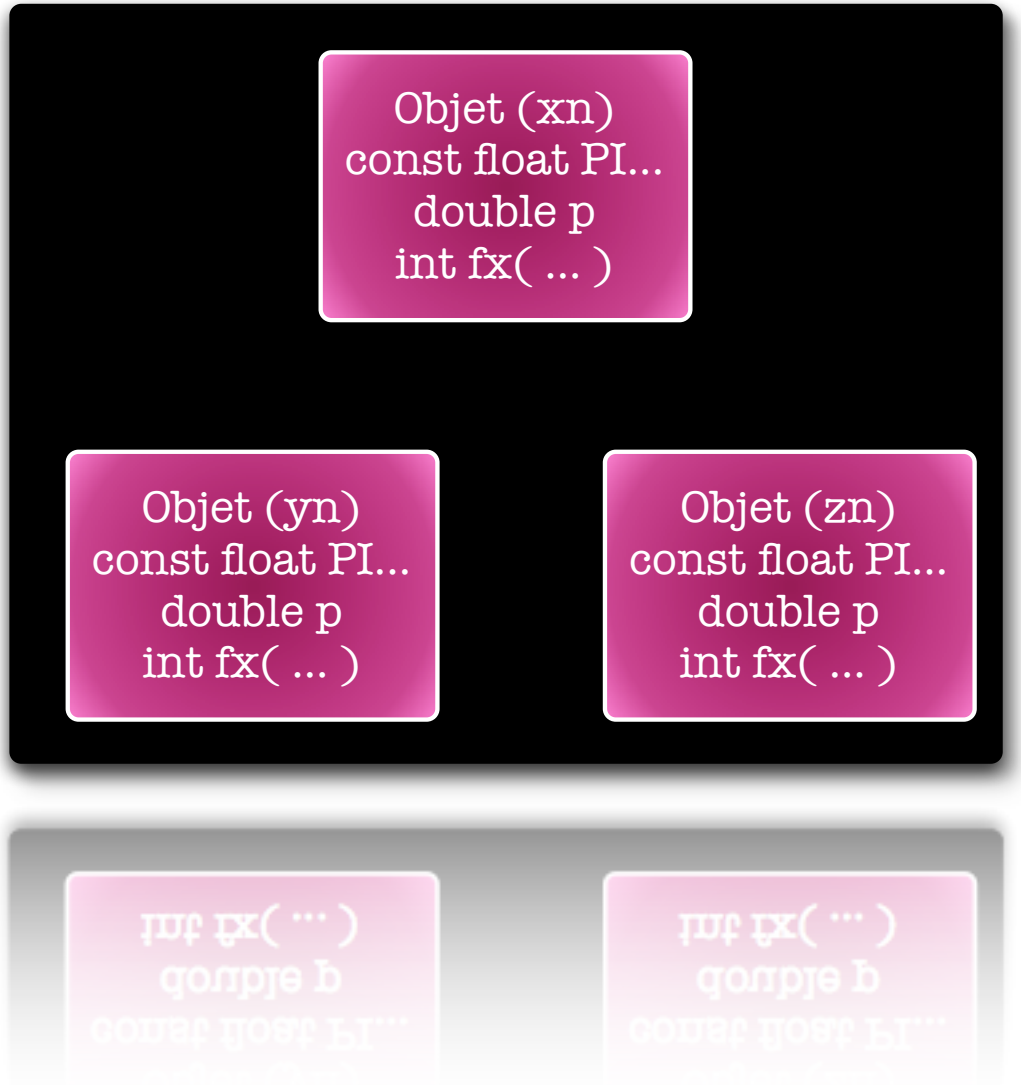
L'objet rend des services sur les données qui lui appartiennent !

Introduction à la notion d'objets



Dans l'approche fonctionnelle vous devez appeler la fonction avec les bon paramètres pour chaque variables

Introduction à la notion d'objets



Dans l'approche objets chaque objet (variable) possède les informations nécessaires à son traitement

=> Simplification du code source

Introduction à la notion d'objets

Appel de la «fonction» suivant les différentes données (C)

```
int var_1;
int var_2;
int var_3;

fonction( var_1 );
fonction( var_2 );
fonction( var_3 );
```

Appel de la «fonction» suivant les différentes données (C++)

```
Nombre var_1;
Nombre var_2;
Nombre var_3;

var_1.fonction( );
var_2.fonction( );
var_3.fonction( );
```

IV. Introduction aux objets

- Structure des classes -

Structure des classes

- La classe décrit le modèle structurel d'un objet :
  - Ensemble des attributs (ou champs ou données membres) décrivant sa structure,
  - Ensemble des méthodes (ou opérations ou fonctions membres) qui lui sont applicables.
- Une classe en C++ est une structure qui contient :
  - des *Fonctions*,
  - des *Données/Variables*.

```
class MaPremiereClasse {
    // Début des données
    // ... ... ... ...
    // ... ... ... ...
    // ... ... ... ...
    // Fin des données

    // Début des fonctions
    // ... ... ... ...
    // ... ... ... ...
    // ... ... ... ...
    // Fin des fonctions
};
// n'oubliez pas ce ;
// apres l'accolade
```

Structure des classes

- La classe décrit le modèle structurel d'un objet :
  - Ensemble des attributs (ou champs ou données membres) décrivant sa structure,
  - Ensemble des méthodes (ou opérations ou fonctions membres) qui lui sont applicables.
- Une classe en C++ est une structure qui contient :
  - *des Fonctions*,
  - *des Données/Variables*.

```
class Voiture {
public :
    // Méthodes publiques
    Voiture ( int _annee );
    void demarrer ( int code );
    int  lireVitesse ();
    void accelerer ();
    void freiner ();

private :
    // Attributs privées
    char  immatriculation[6];
    char  *type;
    int    annee;
    float poids;

    // Méthode privée
    void erreur(char *message);
};
```

Décomposition fichiers « .cpp » et « .h »

Fichier « Nombre.h »

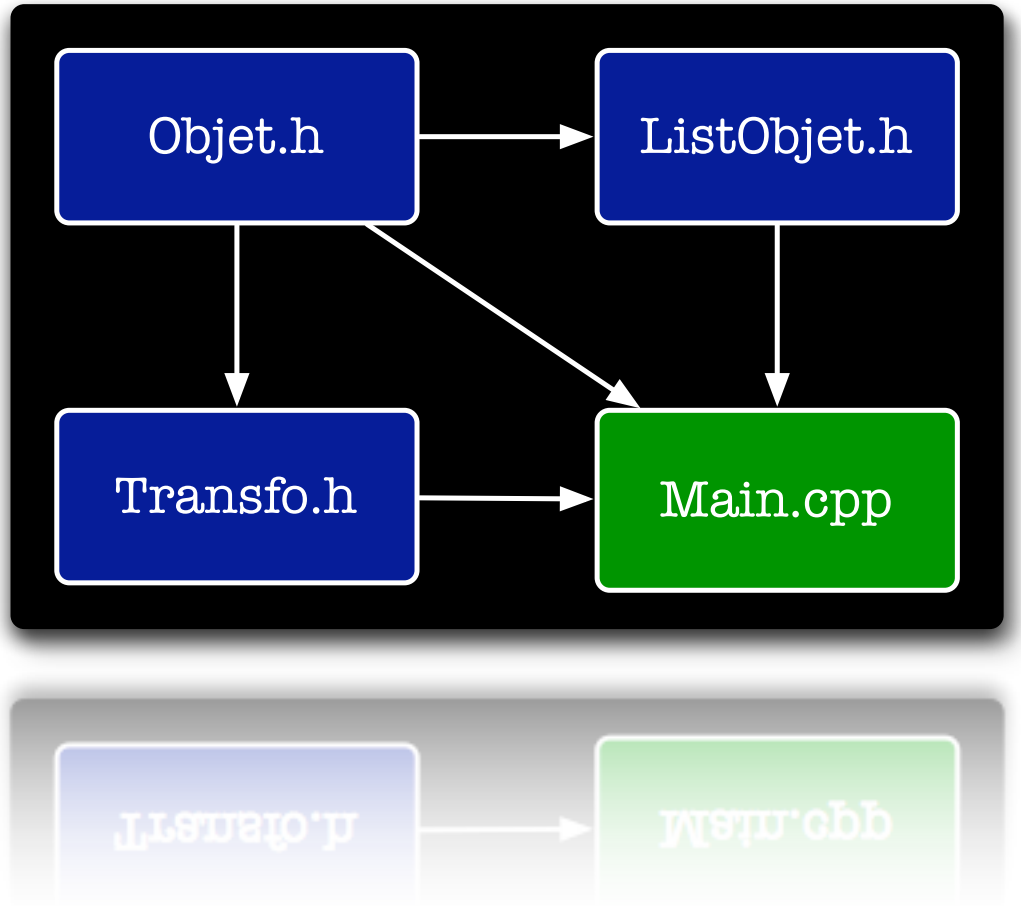
```
class Nombre{
    double valeur;

    Nombre(int _valeur) ;

    int getValue();

    void setValue(int _v);
};
```

Fichier « Nombre.cpp »

```
#include "Nombre.h"

Nombre::Nombre(int _valeur) {
    valeur = (double)_valeur;
}

int Nombre::getValue(){
    return (int)valeur;
}

void Nombre::setValue(int _v){
    valeur = (double)_v;
}
```

Faire comme les pros … « #define »

- Les dépendances entre les objets peuvent parfois provoquer des problèmes,
  - Un fichier « .h » peut être inclut plusieurs fois lors de la compilation d'un fichier « .cpp »
  - Le compilateur trouve plusieurs fois la même déclaration → Erreur.
- Afin d'éviter ce problème, on protège les fichiers « .h » des inclusions multiples
  - Utilisation du préprocesseur,



Faire comme les pros … « #define »

- Les dépendances entre les objets peuvent parfois provoquer des problèmes,
  - Un fichier « .h » peut être inclut plusieurs fois lors de la compilation d'un fichier « .cpp »
  - Le compilateur trouve plusieurs fois la même déclaration → Erreur.
- Afin d'éviter ce problème, on protège les fichiers « .h » des inclusions multiples
  - Utilisation du préprocesseur,

Fichier « Objet.h »

```
#ifndef OBJET_H_
#define OBJET_H_

class Objet{
    // Début des déclarations
    // … … … … … …
    // … … … … … …
    // Fin des déclarations
};

#endif /*OBJET_H_*/
```

Exercice de cours : la classe *ListData*

- Nous souhaitons développer une classe nommée *ListData* simplifiant les calculs liés aux vecteurs de données et assurant les services suivants :
  - Ajout de nouvelles données par l'utilisateur de la classe,
  - Lecture du nombre de données fournies par l'utilisateur,
  - Calcul de la somme des données fournies par l'utilisateur,
  - Calcul de la moyenne des données fournies par l'utilisateur,
  - Calcul de la valeur minimum contenue dans les données fournies,
  - Calcul de la valeur maximum contenue dans les données fournies,
  - Affichage des données présentes dans la liste à l'écran.
- Hypothèse :
  - L'utilisateur ne peut pas fournir plus de 1024 données. Chaque donnée est rentrée une par une dans le tableau (ajoutée à la fin).

Question 1 : Définissez la structure de la classe

```
class ListData
{
     // Attributs
     int liste[1024];
     int nombre;

     // Methodes
     void ajouter(int data);
     int somme();
     double moyenne();
     int min();
     int max();
     int variationMax();
     int taille();
     void afficher();
};
```

Définition de la classe ListData

Définition des attributs de la classe

Définition des méthodes (services) fournies par la classe ListData

Question 1 : Définissez la structure de la classe

```
class ListData
{
     // Attributs
     int *liste;
     int nombre;

     // Methodes
     void ajouter(int data);
     int somme();
     double moyenne();
     int min();
     int max();
     int variationMax();
     int taille();
     void afficher();
};
```

Définition de la classe ListData

Définition des attributs de la classe

Définition des méthodes (services) fournies par la classe ListData

Question 2 : Ajout de données dans la liste

On profite de cette méthode pour vérifier que l'on ne dépasse pas la taille du tableau où l'on mémorise les informations

```
void ListData::ajouter(int data)
{
     if(nombre > 1024){
          cout << "Erreur la liste est pleine !" << endl;
          return;
     }
     liste[nombre++] = data;
}
```

Question 3 : Lecture du nombre de données

Dans cette méthode nous ne retournons que la valeur de la variable contenue dans la classe

```
int ListData::taille()
{
    return nombre;
}
```

Question 4 : Calcul de la somme des données

```
int ListData::somme()
{
    int sum = 0;
    for(int i=0; i<nombre; i++){
        sum += liste[i];
    }
    return sum;
}
```

Le calcul de la somme des données se fait à l'aide d'une boucle parcours uniquement les données saisies par l'utilisateur de la classe

Question 5 : Calcul de la moyenne des données

```
double ListData::moyenne()
{
     double sum = 0;
     for(int i=0; i<nombre; i++){
          sum += liste[i];
     }
     return sum/(double)nombre;
}
```

Méthode permettant de calculer la moyenne des données contenues dans la liste des données

N'oubliez jamais qu'un concepteur objet est avare en nombre de lignes de code... De plus cela a bien d'autres avantages !

```
double ListData::moyenne()
{
     return (double)(somme())/(double)taille();
}
```

Question 6 : Calcul des données mini/maxi

```
int ListData::min()
{
     int _min = 10000000;
     for(int i=0; i<nombre; i++){
          _min = (_min<liste[i])?_min:liste[i];
     }
     return _min;
}
```

Méthode permettant de calculer la valeur minimum contenue dans la liste des données fournies

Méthode permettant de calculer la valeur maximum contenue dans la liste des données fournies

```
int ListData::max()
{
     int _max = -10000000;
     for(int i=0; i<nombre; i++){
          _max = (_max>liste[i])?_max:liste[i];
     }
     return _max;
}
```

Question 7 : Calcul de l'espace de variation

- Développez une nouvelle méthode permettant de connaître l'espace de variation maximum entre les données fournies par le concepteur.
  - ➜ Cette méthode sera nommée variationMax( ) et retournera un entier

```
int ListData::variationMax()
{
     int _min = 10000000;
     int _max = -10000000;
     for(int i=0; i<nombre; i++){
          _min = (_min<liste[i])?_min:liste[i];
          _max = (_max>liste[i])?_max:liste[i];
     }
     return (_max-_min);
}
```

Voici une version plus réaliste de ce que ferait un développeur objet

```
int ListData::variationMax()
{
     return max()-min();
}
```

IV. Introduction aux objets

- Instanciation des classes -

Instanciation d'objets

- Création d'instances à partir des classes
  - De façon similaire à une ***struct*** ou à une ***union***, le nom de la classe représente un nouveau type de donnée,
  - On peut donc définir des variables de ce nouveau type; on dit alors que l'on crée des ***objets*** ou des ***instances*** de cette classe.

- **Exemples**

```
Voiture car_1;                          // une instance simple (statique)
Voiture *car_2;                         // un pointeur (non initialisé)
car_2 = new Voiture(2005);               // création (dynamique) d'une
                                        // instance avec initialisation
Voiture compagnie_location[10];          // un tableau d'instances
```

Instanciation d'objets – Durée de vie

- Déclaration sous forme de variable locale (statique)
  - L'objet a une durée de vie associée à la structure dans laquelle il est déclaré (identique à une variable classique).

```
void MaFonction( … ){
    Objet obj(  );
    … … … … …
    … … … … …
}   // Mort de l'objet
```

- Déclaration sous forme de pointeur (dynamique)
  - Durée de vie illimitée, il faut donc penser à la détruire !
  - Dans le cas contraire, des fuites mémoire vont apparaître.

```
Objet *obj;
obj = new
Objet(parametres);
… … … … …
… … … … …
delete obj;  //Destruction
```

Instanciation d'objets – Durée de vie

- Déclaration sous forme de variable locale (statique)
  - Accès aux méthodes et aux attributs à l'aide du séparateur « . »

```
void MaFonction( … ){
    Objet obj(  );
    obj.Methode( valeur );
    obj.attribut = 10;
}
```

- Déclaration sous forme de pointeur (dynamique)
  - Accès aux méthodes et aux attributs à l'aide du séparateur « → »

```
Objet *obj;
obj = new
obj->Methode( valeur );
obj->attribut = 10;
delete obj;  //Destruction
```

Instanciation d'objets – Durée de vie

```
void MaFonction( … ){
    Objet obj(  );
    obj.Methode( valeur );
    obj.attribut = 10;
}

void Main( … ){
    // … … … … … …
    MaFonction( parametres );
    // … … … … … …
}
```

L'objet est déclaré de manière statique (comme une structure), il vit entre la parenthèse ouvrant et la prochaine fermante. Ensuite il est automatiquement détruit

Instanciation d'objets – Durée de vie

```
void MaFonction( … ){
    Objet *obj;
    obj = new Objet(  );
    obj->Methode( valeur );
    obj->attribut = 10;
}

void Main( … ){
    // … … … … … …
    MaFonction( parametres );
    // … … … … … …
}
```

L'objet est déclaré de manière dynamique (comme une structure), il vit jusqu'à ce qu'on le détruise

Ici nous avons une fuite mémoire car l'objet n'est jamais détruit...

Instanciation d'objets – Durée de vie

```
void MaFonction( … ){
    Objet obj;
    obj = new Objet(  );
    obj->Methode( valeur );
    obj->attribut = 10;
    delete obj;
}

void Main( … ){
    // … … … … … …
    MaFonction( parametres );
    // … … … … … …
}
```

L'objet est déclaré de manière dynamique (comme une structure), il vit jusqu'à ce qu'on le détruise.
C'est l'action que nous réalisons avec le delete.

Instanciation d'objets – Durée de vie

```
void MaFonction( … ){
    Objet *obj;
    obj = new Objet(  );
    obj->Methode( valeur );
    delete obj;
    obj->attribut = 10;
}

void Main( … ){
    // … … … … … …
    MaFonction( parametres );
    // … … … … … …
}
```

L'objet est déclaré de manière dynamique (comme une structure), il vit jusqu'à ce qu'on le détruise.
C'est l'action que nous réalisons avec le delete.

Un plantage a lieu car l'objet a déjà été détruit donc il n'existe plus !

Instanciation d'objets - Accès aux membres

- La manière de laquelle vous créez vos objets influe sur l'écriture des accès à ces derniers,
  - Risque de confusion / problème de fuite si vous n'êtes pas rigoureux dans l'écriture de votre code,
  - Le compilateur est tout de même là pour vous insulter si vous vous trompez…
- Comparaison avec JAVA
  - En Java, il n'existe pas de destructeur, la machine virtuelle réalise cette tache de manière autonome (consommation de temps CPU),
  - Pas de notion de pointeur (visible), impliquant une seule manière d'accéder aux membres d'une classe « . ».

Question 8 : Utilisation statique de la classe

- Ecrivez un programme "main" permettant d'instancier statiquement votre objet.
- Ensuite ajouter 4 données à la liste (3,9,13, -1).
- Calculer la moyenne des données.
- Ajouter la donnée (0).
- Récupérez la valeur minimum présente dans la liste.

```
int main() {
     ListData ld;
     ld.ajouter(  3 );
     ld.ajouter(  9 );
     ld.ajouter( 13 );
     ld.ajouter( -1 );
     cout << "Calcul de la moyenne      : " << ld.moyenne() << endl;
     ld.ajouter( 0 );
     cout << "Calcul de la valeur min   : " << ld.min()     << endl;
     return 0;
}
```

Question 9 : Utilisation dynamique de la classe

- Ecrivez un programme "main" permettant d'instancier dynamiquement votre objet.
- Ensuite ajouter 4 données à la liste (3,9,13, -1).
- Calculer la moyenne des données.
- Ajouter la donnée (0).
- Récupérez la valeur minimum présente dans la liste.

```
int main() {
     ListData *ld = new ListData();
     ld->ajouter(  3 );
     ld->ajouter(  9 );
     ld->ajouter( 13 );
     ld->ajouter( -1 );
     cout << "Calcul de la moyenne      : " << ld->moyenne() << endl;
     ld->ajouter( 0 );
     cout << "Calcul de la valeur min   : " << ld->min() << endl;
     delete ld;
     return 0;
}
```

Question 10 : Utilisation multiple de la classe

- Créez 2 objets de type ListeData (de manière dynamique)
- Ensuite ajouter 4 données à la liste "1" (3,9,13, -1).
- Ensuite ajouter 3 données à la liste "2" (-5,-2,-7).
- Calculer la moyenne des données des 2 listes.
- Comparez ces données et affichez à l'écran si elles sont identiques ou non ?

Question 10 : Utilisation multiple de la classe

- Créez 2 objets de type ListeData (de manière dynamique)
- Ensuite...
- Ensuite...
- Calculer...
- Comparez...

```
int main() {
     ListData ld;
     ld.ajouter(  3 );
     ld.ajouter(  9 );
     ld.ajouter( 13 );
     ld.ajouter( -1 );

     ListData *ld2 = new ListData();
     ld2->ajouter(  -5 );
     ld2->ajouter(  -2 );
     ld2->ajouter( -13 );

     int moy1 = ld.moyenne();
     int moy2 = ld2->moyenne();

     if( moy1 == moy2 )
          cout << "Les moyennes sont identiques" << endl;
     else
          cout << "Les moyennes ne sont pas identiques" << endl;

     delete ld2;
     return 0;
}
```

Question 11 : Utilisation multiple de la classe

➜ Utilisez des commandes ASSERT afin de vérifier que la classe fonctionne correctement.

```
int main() {
     ListData *ld = new ListData();
     ld->ajouter( 2 );
     ld->ajouter( 5 );
     ld->ajouter( -2 );
     assert( ld->somme()   == 5         );
     assert( ld->moyenne() == (5.0/3.0) );
     assert( ld->min()     == -2        );
     assert( ld->max()     == 5         );
     assert( ld->taille()  == 3         );
     delete ld;
     return 0;
}
```

IV. Introduction aux objets

- Adoptez les bonnes manières -

Adoptez de bonnes manières - Vocabulaire objet

- Les objets décrits en *C++*, en *Java* ou dans tout autre langage informatique sont composés de 2 éléments de base,

Les attributs

- Les *attributs correspondent aux données* qui sont mémorisées dans l'objet de manière à pouvoir assurer sa fonction. En C, les attributs sont nommés données ou variables.

Les méthodes

- Les *méthodes correspondent aux traitements* qui sont disponibles au travers de l'objet. En C, les méthodes sont généralement nommées fonctions.

Adoptez de bonnes manières – Style d'écriture

- Faire attention à son style d'écriture…
  - La *première lettre* du nom de la classe en majuscule,
  - La liste des *membres publics* en premier,
  - Les *noms des méthodes en minuscules* (tout au moins la première partie si le nom est composé),
  - Le caractère _ comme premier caractère du nom d'une donnée membre,
- Ces règles simples permettent de relire plus facilement un code source, rendant sa compréhension plus facile par des personnes extérieurs.

Adoptez de bonnes manières – Style d'écriture

- Essayer d'écrire du code intelligible pour tous car vous ne savez jamais qui va relire votre "œuvre d'art" !
  - Le compilateur sait éliminer les variables qui ne servent à rien (dans certains cas de figure),
  - Les optimisations peuvent améliorer certaines parties de votre code mieux que ce que vous feriez vous même...

```
int fonction(int a, int b, int c){
    return (a>b)?((a>c)?a:c):((b>c)?b:c);
}   // Mais que fait donc cette jolie fonction ?
```

Adoptez de bonnes manières – Style d'écriture

- La première lettre du nom de la classe en majuscule
  - *class* **..**
  - *class* ***N****ombre*
- Les noms des méthodes en minuscules (tout au moins la première partie si le nom est composé);
  - *int* ***get****Pixel(int x, int y)*
  - *char** ***lire****Nom( )*
- Le caractère "_"comme premier caractère du nom d'une donnée membre :
  - *void ecrireValeur(int _valeur, int _position)*
  - *double FIR(int _echantillons[32], int _donnees[32])*

IV. Introduction aux objets

- Les constructeurs -

Les constructeurs - Introduction

- Pourquoi un constructeur ?
  - Les données membres d'une classe ne peuvent pas être initialisées à l'instanciation de l'objet,
  - Il faut donc prévoir une méthode d'initialisation des données qui sera invoquée lors de la création de l'objet en mémoire,
- Que risque t'on a ne pas en avoir ?
  - Si l'on oublie d'appeler cette fonction d'initialisation, le reste n'a plus de sens et il se produira très certainement des surprises fâcheuses à l'exécution,
- Solution
  - Il est nécessaire d'écrire un ou plusieurs constructeurs afin de permettre une initialisation correcte de l'objet (on peut/doit considérer plusieurs cas, par exemple : avec et sans paramètres).

Les constructeurs - Exemple

```
class Somme{
    double valeur;

    Somme( ){
        valeur = 0;
    }

    void ajouter(double _valeur){
        valeur += _valeur;
    }

    double lireResultat( ){
        return valeur;
    }
};
```

Lors de la création de l'objet, le constructeur est automatiquement appelé.

```
int main( void ){
    Somme *s = new Somme();
    s->ajouter( 10 );
    delete s;
    return 1;
}
```

Les constructeurs - Exemple

```
#include "Somme.h"

Somme::Somme(){
     cout << "Construction" << endl;
     valeur = 0;
}

void Somme::ajouter(double _valeur){
     valeur += _valeur;
}

double Somme::lireResultat( ){
     return valeur;
}
```

```
> ./KeyNote
Construction
>
```

```
int main( void ){
     Somme *s = new Somme();
     // ... ... ... ... ...
     delete s;
     return 1;
}
```

Les constructeurs multiples

- Le langage C++ comme la majorité des langages objets ne limite pas le nombre de constructeurs que peu posséder un objet,

- Différents cas de figure sont possibles :

1. *Aucun constructeur,*
   - ➡ Ce cas de figure est peu recommandé car le constructeur en créra un vide ce qui n'initialise rien...

2. *Un constructeur*
   - ➡ Cas de figure assez fréquent ou l'on possède peu de cas de figure pour l'utilisation de l'objet.

3. *Plusieurs constructeurs*
   - ➡ C'est le meilleur choix dans une majorité de cas car cela augmente la flexibilité de l'objet pour ses utilisateurs et donc sa réutilisation dans plusieurs projets.Il est possible d'utiliser un constructeur avec des valeurs par défaut.

Les constructeurs multiples - Exemple

```
class Somme{
     double valeur;

     Somme( ){
          valeur = 0;
     }

     Somme(int _valeur ){
          valeur = (double)_valeur;
     }

     Somme(double _valeur ){
          valeur = _valeur;
     }

     void ajouter(double _valeur){
          valeur += _valeur;
     }

     double lireResultat( ){
          return valeur;
     }
};
```

Créer plusieurs constructeurs permet d'utiliser plus simplement la classe par la suite => ce n'est pas une perte de temps !

```
int main( void ){
     Somme *s1 = new Somme();
     Somme *s2 = new Somme( 2    );
     Somme *s3 = new Somme( -4,3 );
     // ... ... ... ... ...
     delete s1, s2, s3;
     return 1;
}
```

Les constructeurs multiples - Valeurs par défaut

L'utilisation de constructeur avec des valeurs par défaut réduit la taille du code en augmentant la flexibilité et la réutilisabilité

```
class Somme{
    double valeur;

    Somme(double a=0, double b=0 ){
        valeur = a + b;
    }

    void ajouter(double _valeur){
        valeur += _valeur;
    }

    double lireResultat( ){
        return valeur;
    }
};
```

```
class Somme{
    double valeur;

    Somme( ){
        valeur = 0;
    }

    Somme(double a){
        valeur = a;
    }

    Somme(double a=0, double b=0 ){
        valeur = a + b;
    }

    void ajouter(double _valeur){
        valeur += _valeur;
    }

    double lireResultat( ){
        return valeur;
    }
};
```

Question 12 : Ecrivez le constructeur de ListData

➜ Ecrivez le constructeur manquant de la classe ListData. Ce constructeur ne prend aucun paramètre (le tableau est statique : liste[1024])

```
ListData::ListData()
{
     cout << "Construction d'une liste" << endl;
     nombre = 0;
}
```

Question 12 : Ecrivez le constructeur de ListData

➜ Ecrivez le constructeur manquant de la classe ListData. Ce constructeur ne prend aucun paramètre (le tableau est dynamique : int* liste)

```
ListData::ListData()
{
     cout << "Construction d'une liste" << endl;
     nombre = 0;
     liste  = new int [1024];
}
```

Question 13 : Ecrivez le constructeur de ListData

➜ *Ecrivez un second constructeur à la classe, ce dernier prend comme paramètre la première valeur de la liste.*

```
ListData::ListData(int data)
{
     cout << "Construction d'une liste" << endl;
     nombre = 0;
     liste  = new int [1024];
     liste[nombre++] = data;
}
```

On réécrit le constructeur par défaut en modifiant légèrement son comportement.

Il est toutefois préférable de réemployer les méthodes déjà créées pour un plus grand confort / sécurité.

```
ListData::ListData(int data)
{
     cout << "Construction d'une liste" << endl;
     nombre = 0;
     liste  = new int [1024];
     ajouter( data );
}
```

Question 14 : Ecrivez le constructeur de ListData

➜ Reprenez la question 10 et adapter votre code pour prendre en considération cette "nouvelle" possibilité.

```
int main() {
     ListData *ld = new ListData( 3 );
     ld->ajouter(  9 );
     ld->ajouter( 13 );
     ld->ajouter( -1 );
     cout << "Calcul de la moyenne      : " << ld->moyenne() << endl;
     ld->ajouter( 0 );
     cout << "Calcul de la valeur min   : " << ld->min() << endl;
     delete ld;
     return 0;
}
```

IV. Introduction aux objets

- Les destructeurs -

Les objets C++ (Les destructeurs)

- **Problématique**
  - Pour créer des objets nous allouons de manière plus ou moins transparente de la mémoire,
  - Il est nécessaire de libérer cette mémoire pour l'objet courant mais aussi tout ceux qui « meurent » en même temps que lui,
  - Sinon lorsque le programme tourne pendant des heures / jours il peut arriver a saturer le système (prendre sans jamais rendre)...
- **Solution**
  - Il est nécessaire de mettre en œuvre une méthode permettant de faire le ménage lorsque l'on n'a plus besoin d'un objet.
  - Dans certains objets, le destructeur par défaut (généré par le compilateur) peu suffire, mais il est toujours préférable de créer le sien,
    - ⇨ Oblige à la réflexion sur les données manipulées.

Objets C++ (Les destructeurs)

- De la même façon que pour les constructeurs, le **destructeur** est une fonction membre spécifique de la classe qui est appelée implicitement à la destruction de l'objet.

- Le destructeur est une fonction
  - Son nom est identique au nom de la classe précédé du caractère (tilda)
  - Il ne retourne pas de valeur,
  - Il n'accepte aucun paramètre (pas de surcharge possible),

```
class Exemple {
    Exemple();
    ~Exemple();
};

Exemple::Exemple() {
    // Code du constructeur
}

Exemple::~Exemple() {
    // Code du destructeur
}
```

Objets C++ (Les destructeurs)

A la construction de l'objet on alloue dynamiquement de la mémoire, a la destruction on se doit de libérer la mémoire que l'on a précédemment réservé.

```
class Liste {
    double *liste;

    // Constructeur de la classe
    Liste( ) {
        liste = new double[32];
    }

    // Destructeur de la classe
    ~Liste( ) {
        delete liste;
    }

    double get( int pos ){
        return liste[pos]; }

    double set( int pos, double val ) {
        liste[pos] = val; }
};
```

Objets C++ (Les destructeurs) - Instance dynamique

```
class Exemple {
    Exemple() {
        cout << "Construction" << endl;
    }

    ~Exemple() {
        cout << "Destruction" << endl;
    }
};


int main( void ){
    cout << "Lancement" << endl;
    Exemple *e = new Exemple();
    cout << "Pendant" << endl;
    delete e;
    cout << "Apres" << endl;
}
```

Affichage dans le terminal :

```
> ./Mon_Programme
Lancement
Construction
Pendant
Destruction
Apres
>
```

Objets C++ (Les destructeurs) - Instance statique

```
class Exemple {
    Exemple() {
        cout << "Construction" << endl;
    }

    ~Exemple() {
        cout << "Destruction" << endl;
    }
};


int main( void ){
    cout << "Lancement" << endl;
    Exemple e();
    cout << "Pendant" << endl;
}
```

Affichage dans le terminal :

```
> ./Mon_Programme
Lancement
Construction
Pendant
Destruction
>
```

Question 15 : Ecrivez le destructeur de ListData

➜ Ecrivez le destructeur manquant de la classe ListData (le tableau est statique : liste[1024])

```
ListData::~ListData()
{
     cout << "Destruction d'une liste" << endl;
}
```

Question 16 : Ecrivez le destructeur de ListData

➜ Ecrivez le destructeur manquant de la classe ListData (le tableau est dynamique : int* liste)

```
ListData::~ListData()
{
     cout << "Destruction d'une liste" << endl;
     delete liste;
}
```

IV. Introduction aux objets

~ Le mot clef “static” ~

Le mot clef « static » - Introduction

- Les objets possèdent leurs attributs ainsi que leurs méthodes de manière “propre”,
  - Ce comportement peut devenir très coûteux en mémoire dans certains cas particuliers,
    - Une application manipulant beaucoup d’objets dont le ratio **sizeof(***données***)** / **sizeof(***méthodes***)** est très faible.
  - Impossibilité de partager des données entre les instances,
    - Par exemple pour déterminer à chaque instant le nombre d’objets instanciés, le nombre d’objets détruits, etc.

Le mot clef « static » - Introduction

- Pour contourner ces problèmes, le mot clef « *static* » a été introduit dans la langage C++
  - Attaché à un attribut, il implique son unicité dans l'application et son partage entre les instances de l'objet,
    - Il faut penser à initialiser l'objet *en dehors du constructeur* de la classe !
  - Dans le cas d'une méthode cela signifie que cette dernière est instanciée une unique fois en mémoire,
    - Cette méthode ne peut pas utiliser le mot clef « *this* » ni directement les attributs de l'objet,

Le mot clef « static » - Exemple (1)

```
class Nombre{
    int count;

    Nombre( ){
        count
    }

    ~Nomb
        c
    }

    int Li
        ret
    }
};
```

FAUX

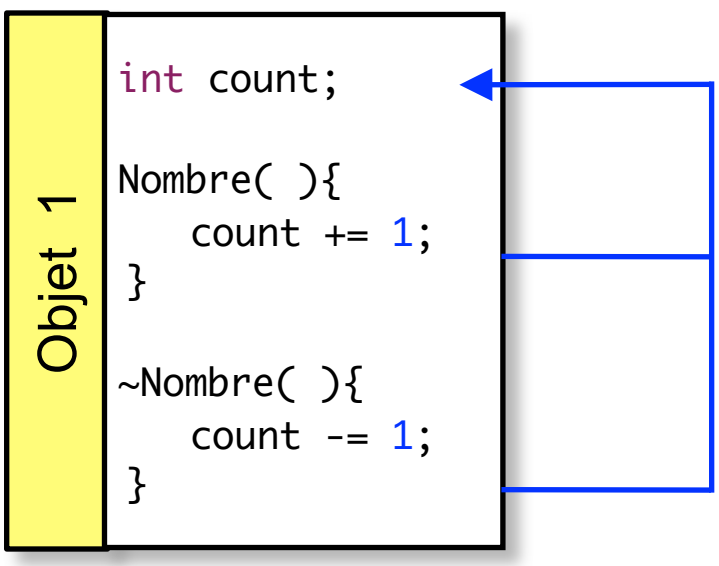
A la construction de
on incrémente un
r d'instance

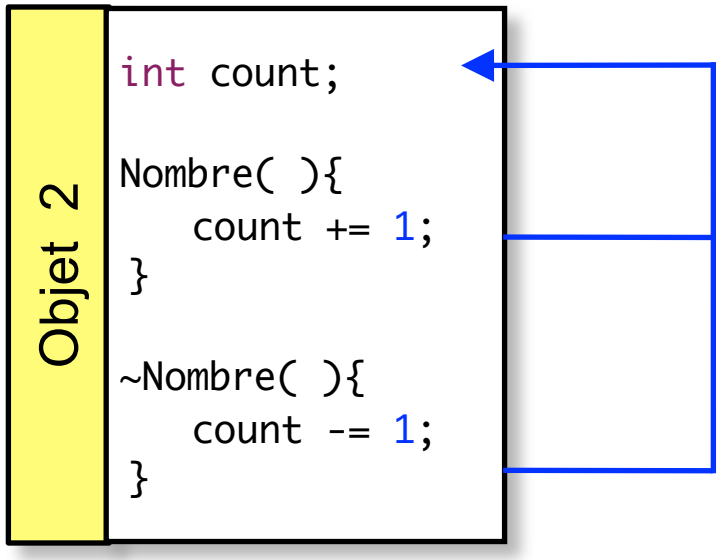
uction de
crémente le
d'instance

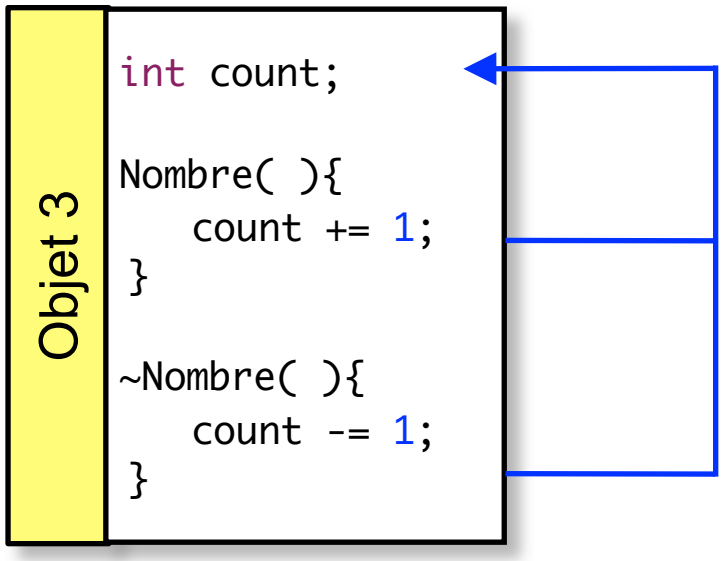
La méthode retourne la valeur du compteur d'instances

Le mot clef « static » - Exemple

A chaque objet instancié son attribut « count », dans ce cas, il est impossible d'interagir avec les autres objets et savoir combien d'instances sont en « activité ».







Le mot clef « static » - Exemple (2)

```
class Nombre{
    static int count;

    Nombre( ){
        count += 1;
    }

    ~Nombre( ){
        count -= 1;
    }

    int LireNombreInstances( ){
        return count;
    }
};

int Nombre::count = 0;
```

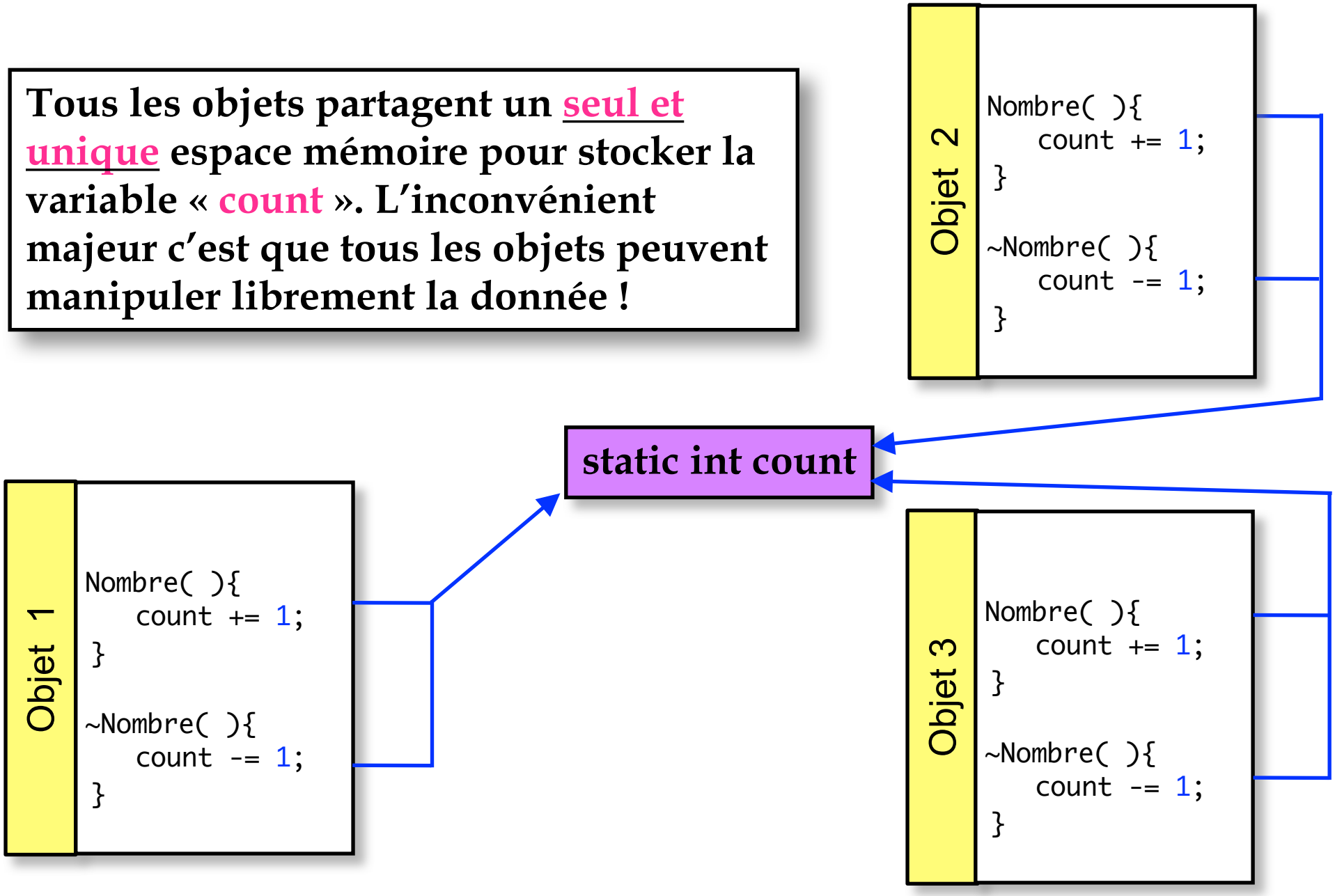
La variable est définie comme étant statique, cela implique qu'elle est partagée entre les différentes instances

La méthode retourne réellement le nombre des instances en mémoire à un instant T

Initialisation de la variable au lancement du programme (main) et non à l'instanciation de la classe.

Le mot clef « static » - Exemple

Tous les objets partagent un seul et unique espace mémoire pour stocker la variable « count ». L'inconvénient majeur c'est que tous les objets peuvent manipuler librement la donnée !



Question 17 : Dénombrement du nombre d'objets

➜ *Modifiez la classe ListData afin de permettre un dénombrement des objets à l'aide d'une méthode nommée count()*

```
class ListData
{
     // Attributs
     int liste[1024];
     int nombre;

     // Methodes
     void ajouter(int data);
     int somme();
     double moyenne();
     int min();
     int max();
     int variationMax();
     int taille();
     void afficher();

     // Instances
     static int instance;
     int count();
};
```

```
ListData::~ListData()
{
     instance -= 1;
     delete liste;
}
```

```
ListData::ListData()
{
     nombre = 0;
     instance += 1;
     liste  = new int [1024];
}
```

```
int ListData:: instance = 0;

ListData::count(){
     return instance;
}
```

IV. Introduction aux objets

- Le polymorphisme -

Abscence du Polymorphisme en langage C

- Dans le langage C, l'identification de la fonction (ou méthode) appelée par le concepteur dans son programme est basé uniquement sur le nom de la fonction déclarée.
- Cette identification des fonctions oblige à définir des fonctions aux noms différents pour peu que l'on réalise la même opération sur des types de données non compatibles entre eux.

Inconvénient typique du langage C

```
int valeur;

void LoadInteger(int v){
     value = v;
}

void LoadDouble(double v){
     value = (int)v;
}

void LoadString(char *v){
     value = atoi(v);
}
```

La notion de Polymorphisme en C++

- Dans le langage C++ cette restriction sémantique est levée grâce à une identification différente des méthodes définies et appelées par le concepteur,
- Une méthode est identifiée par une « clef » basée sur son nom et le type et l'ordre des paramètres qu'elle accepte,
- C'est le compilateur qui se charge d'analyser les appels réalisés et de les lier aux fonctions déclarées,
- **Attention :** le type de la donnée renvoyée n'est pas utilisé lors de la génération de l'identifiant.

Utilisation du concept de polymorphisme

```
int valeur;

void Load( int v) {
    value = v;
}

void Load( double v ) {
    value = (int)v;
}

void Load( char *v ) {
    value = atoi( v );
}
```

La notion de Polymorphisme en C++

- Cette possibilité de définir plusieurs fonctions portant le même nom et acceptant des paramètres de type et de nombre différent est appelée le polymorphisme.

- **Attention**, certains paramètres peuvent sembler différents sans pour autant l'être !

  ↳ Sur les architectures 32 bits : (int = long)

Ces exemples sont ils valides vis-a-vis des règles ?

```
int  ma_fonction();
void ma_fonction(int _a);
```

Code source valide

```
bool ma_fonction(int _a, int b);
bool ma_fonction(int _a);
bool ma_fonction(int _b);
```

Code source non valide

```
void ma_fonction(int  _a);
void ma_fonction(long _a);
```

Code source non valide

```
void ma_fonction(char   _a);
void ma_fonction(short  _a);
void ma_fonction(double _a);
```

Code source valide

```
int    ma_fonction(int _a, int b);
double ma_fonction(int _a, int b);
```

Code source non valide

Polymorphisme – Exemple (1)

```
class Nombre{
     int valeur;

     Nombre( int _valeur) {
          valeur = _valeur;
  }

     Nombre( char * _valeur ) {
          valeur = atoi(_valeur );
  }

     Nombre( Nombre *objet ) {
          valeur = objet->valeur;
  }
}
```

Cet objet possède de multiples constructeurs permettant à ses utilisateurs d'utiliser le plus adéquate en fonction des besoins.

Polymorphisme – Exemple (2)

```
class Personne{
public:
    char *nom_famille;

    void  nom(char *_nom);
    char* nom( void );
};


Personne *p = new Personne();
p->nom("LE GAL");
cout << "Nom = " << p->nom() << endl;
delete p;
```

On utilise un seul et unique nom de méthode afin d'accéder en lecture et en écriture à une variable de classe

IV. Introduction aux objets

- La surcharge des opérateurs -

Polymorphisme – Surcharge d'opérateurs

- La *surcharge* d'opérateurs ou de méthodes est la capacité à pouvoir redéfinir une méthode déjà existante dans un objet,
  - Utilisation du même identifiant et des même paramètres que la fonction à « surcharger »,
  - Cette particularité sera détaillée dans la section consacrée à l'héritage et à la déviation de classes,
- La surcharge peut aussi être employée pour redéfinir les opérateurs de base entre les classes (+, -, *, /, &, etc.) et les opérateurs de conversion, on peut aussi utiliser les mécanismes de surcharge des opérateurs existants.

Surcharge des méthodes – Opérateurs de base

- *Surcharge des opérateurs arithmétiques classiques*

```
Nombre a(2), b(3);
a+= b;
a -= b;
a *= b;
```

```
Nombre &Nombre::operator+=(Nombre &n)
{
   valeur += n. valeur;
   return *this;
}

Nombre &Nombre::operator-=(Nombre &n)
{
   valeur -= n.valeur;
   return *this;
}

Nombre &Nombre::operator*=(Nombre &n)
{
   valeur *= n.valeur;
   return *this;
}
```

Surcharge des méthodes – Opérateurs de base

- Surcharge des opérateurs de comparaison

```
Nombre a(2), b(3);
bool c = (a == b);
bool d = (a > b);
bool e = (d < c);
```

```
bool Nombre::operator==(Nombre &n)
{
    return (valeur == n→valeur);
}

bool Nombre ::operator<(Nombre &n)
{
    return (valeur < n→valeur);
}

bool Nombre ::operator>(Nombre &n)
{
    return (valeur > n→valeur);
}
```

Surcharge des méthodes – Opérateurs de base

- Surcharge des opérateurs d'affectation.

```
Nombre a(2);
Nombre b = a;
```

- Cas spéciaux

```
b = b;
Nombre c = b = a;
```

On gère le cas particulier ou l'on fait (obj = obj) car cela n'a pas de sens...

```
Nombre Nombre::operator=
          ( const Nombre &n )
{
   if ( &n != this ) {
      n.valeur = valeur;
   }
   return *this;
}
```

Surcharge des méthodes – Opérateurs de base

- Surcharge des opérateurs de pré & post-incrémentation

```
Nombre a(2);
a++;
a--;
```

```
// Opérateur suffixe : retourne
// la valeur et incrémente la
// variable.
Nombre Nombre::operator++(int)
{
   Nombre temp(valeur);
   ++valeur;
   return *temp;
}

// Opérateur préfixe : incrémente
// la variable et la retourne.
Nombre Nombre::operator++(void)
{
    valeur += 1;
    return *this;
}
```

IV. Introduction aux objets

- Optimisation de la classe ListData -

Besoin d'optimisation de la classe

- Suite à la livraison de la classe au client, ce dernier nous a fait remarquer ses faibles performances (mémoire, vitesse) !
  - ✦ Nous allons maintenant améliorer ces caractéristiques en modifiant sa structure interne. Dans un même temps vous supprimerez en même temps la limitation sur le nombre de données maximum.
- Proposez une nouvelle structure de classe permettant d'améliorer la vitesse de calcul ainsi que l'occupation mémoire.
  - ➜ Définissez la nouvelle classe tout en conservant les fonctionnalités déjà offertes (les méthodes doivent rester compatibles),
- Modifiez les méthodes afin de les adapter aux changements que vous avez réalisé dans la question précédente.

Question 18 : Nouvelle structure de la classe

➜ *Réfléchissez à une nouvelle manière de proposer les mêmes services sans utiliser de tableaux. Ecrivez la structure de la nouvelle classe nommée ListData_v2*

```
class ListData_v2{
public:
     int sum;
     int nombre;
     int minimum;
     int maximum;

     ListData_v2();
     ListData_v2(int data);
     ~ListData_v2();
     void ajouter(int data);
     int somme();
     double moyenne();
     int min();
     int max();
     int variationMax();
     int taille();
     void afficher();
};
```

Définition de la classe ListData

Définition des attributs de la classe

Définition des constructeurs et du destructeur

Définition des méthodes (services) fournies par la classe ListData

Question 19 : Nouveau constructeur / destruct.

```
ListData_v2::ListData_v2()
{
     cout << "Construction d'une liste_v2" << endl;
     sum     = 0;
     nombre  = 0;
     minimum = INT_MAX;
     maximum = INT_MIN;
}
```

```
ListData_v2::ListData_v2(int data)
{
     cout << "Construction d'une liste_v2" << endl;
     sum     = 0;
     nombre  = 0;
     minimum = INT_MAX;
     maximum = INT_MIN;
     ajouter( data );
}
```

Mise à jour des constructeurs et du destructeur en fonction des modifications réalisées sur la structure de la classe

```
ListData_v2::~ListData_v2()
{
     cout << "Destruction..." << endl;
}
```

Question 20 : Nouvelle méthodes...

```
void ListData_v2::ajouter(int data)
{
    sum    += data;
    nombre += 1;
    minimum = (minimum<data)?minimum:data;
    maximum = (maximum>data)?maximum:data;
}
```

La méthode d'ajout d'une donnée se "complexifie" mais va simplifier le reste de la classe (vitesse, mémoire) en supprimant la limitation à 1024 données

Question 21 : Nouvelle méthodes...

```
int ListData_v2::somme()
{
    return sum;
}
```

```
int ListData_v2::max()
{
    return maximum;
}
```

```
int ListData_v2::min()
{
    return minimum;
}
```

```
int ListData_v2::variationMax()
{
    return max()-min();
}
```

```
double ListData_v2::moyenne()
{
    return (double)(somme())/(double)taille();
}
```

```
int ListData_v2::taille()
{
    return nombre;
}
```

```
void ListData_v2::afficher()
{
    cout << "[ Affichage des donnees impossible ]" << endl;
}
```

Question subsidiaire

- Vous venez de réécrire la classe initiale afin d'améliorer ses caractéristiques. Pour cela vous avez :
  - ➜Modification des structures de données,
  - ➜Modification des méthodes de calcul,
- *Quel est l'impact de ces modifications sur la fonction main écrite dans la partie précédente ?*

Nouvelles fonctionnalités

- La classe que nous venons de créer possède déjà bien des fonctionnalités mais pour la rendre encore plus interessante, nous souhaitons en rajouter.
- Voici la liste des fonctionnalités qui pourrait être utiles :
  - ➜ Ajout d'un constructeur par recopie,
  - ➜ Ajout d'une méthode permettant d'additionner 2 objets de type liste,
  - ➜ Ajout d'une méthode permettant se soustraire 2 objets de type liste,
  - ➜ Ajout d'une méthode permettant de comparer 2 objets de type liste (égal et différent),
- Etape 6 : définissez les prototypes des méthodes et écrivez leur code source (attention à être un bon développeur objet !). Modifiez le programme main permettant de tester la classe.

Question 22 : Constructeur par recopie

```
ListData_v2 *ld2 = new ListData_v2( -2 );
ld2->ajouter( 5 );
ld2->ajouter( 2 );
ListData_v2 *ld3 = new ListData_v2( ld2 );
```

On souhaite pouvoir recréer une nouvelle instance de classe identique à celle fournie en paramètre

```
ListData_v2::ListData_v2(ListData_v2 *liste)
{
    cout << "Construction d'une liste_v2" << endl;
    sum     = liste->somme();
    nombre  = liste->taille();
    minimum = liste->min();
    maximum = liste->max();
}
```

Ce constructeur prend en paramètre un objet, en récupère les informations et les utilise pour son initialisation => clonage

Question 23 : Addition de 2 listes

```
ld3->addition( ld2 );
ld3->soustraction( ld2 );
```

On veut pouvoir ajouter les données contenues dans la liste "ld2" dans la liste "ld3"

```
void ListData_v2::addition(ListData_v2 *liste)
{
     sum    += liste->somme();
     nombre += liste->taille();
     minimum = (minimum<liste->min())?minimum:liste->min();
     maximum = (maximum<liste->max())?maximum:liste->max();
}
```

Question 24 : Soustraction de 2 listes

```
ld3->addition( ld2 );
ld3->soustraction( ld2 );
```

On veut pouvoir soustraire les données contenues dans la liste "ld2" dans la liste "ld3"

```
void ListData_v2::soustraction(ListData_v2 *liste)
{
    sum    -= liste->somme();
    nombre += liste->taille();
    minimum = (minimum<liste->min())?minimum:liste->min();
    maximum = (maximum<liste->max())?maximum:liste->max();
}
```

Question 25 : Fonction d'égalité et de différence

```
bool ListData_v2::egalite(ListData_v2 *liste)
{
    if( sum !=liste->somme()     ) return false;
    if( nombre != liste->taille()) return false;
    if( minimum != liste->min()  ) return false;
    if( maximum != liste->max()  ) return false;
    return true;
}
```

Méthode permettant de vérifier si la liste passée en paramètre est équivalente à l'instance courante

```
bool ListData_v2::difference(ListData_v2 *liste)
{
    return ! egalite(liste);
}
```

Méthode permettant de vérifier si la liste passée en paramètre est différente de l'instance courante

Question 26 : Mise en oeuvre des fonctionnalités

```
ListData_v2 *ld3 = new ListData_v2( ld2 );
cout << "Calcul de la somme        : " << ld2->somme()         << endl;
cout << "Calcul de la moyenne      : " << ld2->moyenne()       << endl;
cout << "Calcul de la valeur min   : " << ld2->min()           << endl;
cout << "Calcul de la valeur max   : " << ld2->max()           << endl;
cout << "Taille de la liste        : " << ld2->taille()        << endl;
cout << "Test d'egalite des listes : " << ld3->egalite   (ld2) << endl;
cout << "Test de diff.  des listes : " << ld3->difference(ld2) << endl;

ld3->addition( ld2 );
cout << "Calcul de la somme        : " << ld2->somme()   << endl;
cout << "Calcul de la moyenne      : " << ld2->moyenne() << endl;
cout << "Calcul de la valeur min   : " << ld2->min()     << endl;
cout << "Calcul de la valeur max   : " << ld2->max()     << endl;
cout << "Taille de la liste        : " << ld2->taille()  << endl;

ld3->soustraction( ld2 );
cout << "Calcul de la somme        : " << ld2->somme()   << endl;
cout << "Calcul de la moyenne      : " << ld2->moyenne() << endl;
cout << "Calcul de la valeur min   : " << ld2->min()     << endl;
cout << "Calcul de la valeur max   : " << ld2->max()     << endl;
cout << "Taille de la liste        : " << ld2->taille()  << endl;

delete ld2;
delete ld3;
```

5

“ Objets, Notions Avancées ”

V. Objets, notions avancées

- Encapsulation des données -

Encapsulation des données - Introduction

- L'encapsulation des attributs contenus dans un objet à plusieurs avantages vis-à-vis des accès traditionnels aux structures de données (C) :
  - *Découplage* des services rendus (traitement) et du stockage des données au sein de l'objet,
  - Rendre invisible les données inutiles à l'utilisation de l'objet (*protection des secrets de fabrication*),
  - *Contrôle sur les affectations* des données, restriction de leurs lectures (ce qui est caché doit le rester),
  - Maintenabilité et *évolutivité* plus facile,

Encapsulation des données - Problème

- L'accès direct aux attributs d'une classe ne permet pas de s'assurer de la cohérence de leurs valeurs,
  - Besoin de *vérifier la validité* des données saisies,
  - *Normalisation des transformations* (arrondis lorsque float→int par ex.),
  - Dissociation de l'accès des données / méthode de stockage,
- Pas de protection du contenu des attributs:
  - Ecriture possible à tout moment,
  - Impossibilité de gérer une affectation unique (configuration),
  - Collision possible avec les phases de débogage,

```
class
Compte_Bancaire{
    char solde;
    // … … … … … …
}

int main( void )
    Compte_Bancaire
cb;
    cb.solde = 200;
    // … … … … … …
}
```

```
}
```

Le solde affecté est supérieur au solde max. autorisé par le programme (dynamique de « solde »). Les char sont codés sur 8 bits, cette borne max. est 127.

Encapsulation des données - Solution

- Utilisation de méthodes afin d'assurer la validité des affectations,
- Grâce à cela, il devient possible de debugger simplement le programme,
  - Poser un point d'arrêt sur chaque affectation de la donnée,
- L'utilisateur ne voit plus *que les services* proposés par la classe,
  - *Indépendance* du traitement vis-à-vis de la mémorisation,
- On peut changer le coeur de l'objet sans toucher aux interfaces (évolutivité).

```
class Compte_Bancaire{
    char solde;
    // … … … …

    void setSolde(int v){
      solde = (v>127)?127:v);
  }
}

int main( void )
    Compte_Bancaire cb;
    //cb.solde = 200;
    cb.setSolde( 200 );
    // … … … …
}
```

Résumé sur l'encapsulation

L'encapsulation des données

L'encapsulation des données dans un développement objet vise a restreindre ou interdire directement l'accès aux attributs des objets à l'extérieur de la classe (voir aussi à l'intérieur). Pour cela l'objet doit proposer des interfaces (méthodes) pour lire ou écrire les attributs.
Cette technique permet de sécurisé l'accès aux données en vérifiant par exemple des conditions de valeur ou d'accès et permettant une plus grande flexibilité du coeur des objets (on peut modifier les attributs de la classe sans pour autant devoir modifier toutes les utilisations...).

Le processus d'encapsulation des données est un élément essentiel de la programmation par objets dans un milieu collaboratif !

V. Objets, notions avancées

- Les droits d'accès -

Droits d'accès - Introduction

- Pour éviter que l'ensemble des attributs et des méthodes ne soit visible et modifiable depuis l'extérieur,
  - Méthodes de restriction de la visibilité des données / méthodes existent,
  - La gestion de la visibilité des méthodes et des attributs est du *rôle du concepteur uniquement*,
- 3 niveaux de protection des attributs sont disponibles dans le langage C++ :
  1. Visibilité publique (**public**),
  2. Visibilité protégée (**protected**),
  3. Visibilité privée (**private**),

Droits d'accès – Accès « private »

- **Accès privé**
  - les membres privés ne sont accessibles que par les méthodes appartenant à la classe,
  - Les parties privées sont aussi appelées **réalisations**,
- Droits d'accès minimums que l'on peut accorder à un attribut ou à une méthode,
  - Cela permet de cacher les détails d'implémentation vis-à-vis de l'extérieur,

A utiliser pour 99% des attributs !

```
class Acces{
public:
    // Constructeur(s)
    // Destructeur
    // Attributs …
    // Méthodes …

protected:
    // Attributs …
    // Méthodes …

private:
    // Attributs …
    // Méthodes …
}
```

Droits d'accès – Accès « protected »

- **Accès protégé**
  - Les membres protégés sont équivalents aux membres privés,
  - Ils sont accessibles par les méthodes des classes dérivées (voir héritage),
- Un compromis entre visibilité publique et privée,
  - Cette alternative est plutôt peu utilisée sauf dans les cas l'on rencontre des *problèmes de performance*.

```
class Acces{
public:
    // Constructeur(s)
    // Destructeur
    // Attributs …
    // Méthodes …

protected:
    // Attributs …
    // Méthodes …

private:
    // Attributs …
    // Méthodes …
}
```

Droits d'accès – Accès « public »

- **Accès publique**
  - Les membres publics sont accessibles par tous,
  - La partie publique est appelée **interface** car tout le monde peut y accéder,
  - De plus ce sont les méthodes (attributs) que toutes classes dérivées possèdent obligatoirement,
- Permet d'utiliser la classe (méthodes et attributs) dans un programme.

```
class Acces{
public:
    // Constructeur(s)
    // Destructeur
    // Attributs …
    // Méthodes …

protected:
    // Attributs …
    // Méthodes …

private:
    // Attributs …
    // Méthodes …
}
```

Droits d'accès – Exemple de Classe

- Nous allons nous attacher à une classe que nous nommerons Pile_FIFO,
- Modélisation équivalente à une FIFO en électronique,
  - FIFO: First-In, First-Out
- L'exemple propose un objet permettant de répondre au comportement de la FIFO.

```
class Pile_FIFO{
    double tableau[256]
    int taille_max;

    bool ajouter(int v, int pos);
    bool supprimer(int v, int pos);

    int read( );
    int write( int value );
    int write( double value );
    bool empty( );
    bool full( );
}
```

Droits d'accès – Exemple de Classe

- L'utilisateur peut accéder à l'ensemble des méthodes et des attributs que vous avez déclaré dans votre code source,
  - Il peut commettre des erreurs par méconnaissance de la classe !
- On va restreindre les droits d'accès aux méthodes et aux attributs,
- Il est préférable d'être trop stricte au départ (quasiment tout en « private »), puis de desserrer la bride plutôt que l'inverse.

```
class Pile_FIFO{
    double tableau[256]
    int taille_max;

    bool ajouter(int v, int pos);
    bool supprimer(int v, int pos);

    int read( );
    int write( int value );
    int write( double value );
    bool empty( );
    bool full( );
}
```

Droits d'accès – Problématique liée

- Le concepteur doit alors se poser plusieurs questions afin de déterminer au mieux sa politique de droits d'accès,
  - A quoi doit réellement servir la classe que je développe ?
  - Que doit réellement voir l'utilisateur ?
  - A quoi doit'il avoir accès ?
  - Y a-t-il des redondances dans mes méthodes et mes attributs ?
- *Cette liste est bien sur loin d'être exhaustive…*

Droits d'accès – Exemple de Classe

```
class Pile_FIFO{
    double tableau[256]
    int taille_max;

    bool ajouter(int v, int pos);
    bool supprimer(int v, int pos);

    int read( );
    int write( int value );
    int write( double value );
    bool empty( );
    bool full( );
}
```

*Données **privées** car elles appartiennent à l'implémentation de la classe et non a son interface.*

*Méthodes **privées** car elles appartiennent à l'implémentation de la classe et non a son interface.*

*Méthodes **publiques** car elles appartiennent aux fonctionnalités élémentaires que l'utilisateur doit utiliser dans la classe.*

Droits d'accès – Encapsulation des données

- Afin de protéger et sécuriser l'accès aux données contenues dans la classe, nous allons toutes les déclarer en « private »
- *Comment alors peut' on y accéder depuis l'extérieur ?*
- Mise en œuvre de méthodes d'accès en lecture et/ou écriture uniquement en fonction des besoins réels.
  - *Accesseurs*: pour lire la valeur des données privées,
  - *Mutateurs*: pour modifier la valeur des données privées,

```
class Acces{
private:
   int read_write;

public:
  void write( int _value ){
     read_write = _value;
  }

  int read( ){
     return read_write;
  }
}
```

Droits d'accès – Exemple d'initialisation

```
class Configuration{
private:
    int parametre;
    bool writed;

public:
    Configuration( ){
        writed = false;
        parametre = 0;
    }

    int read(  ){
        return parametre;
    }
};
```

```
bool write( int _value ){
    if( writed == false ){
        parametre = _value;
        writed = true;
        return true;
    }
    return false;
}

bool readable(){
    return writed;
}
```

Exemples d'utilisation (1)

```
class Compte_Bancaire{
private:
    char solde;

public:
    bool setSolde( int _solde ){
        if( (_solde < -128) || (_solde > 127) ){
            return false;
        }
        solde = _solde;
        return true;
    }
}
```

Exemples d'utilisation (2)

```
bool setSolde( int _solde, int _password ){
    if( password != ????? ){
        return false;
    }
    else if( (_solde < -128) || (_solde > 127) ){
        return false;
    }
    solde = _solde;
    return true;
}
```

Question 27 : Droits d'accès dans ListData

➜ *Récrivez le prototype de la classe ListData en spécifiant les droits d'accès des méthodes et des attributs*

```
class ListData_v2
{
private:
     int sum;
     int nombre;
     int minimum;
     int maximum;

public:
     ListData_v2();
     ListData_v2(int data);
     ListData_v2(ListData_v2 *liste);
     ~ListData_v2();

     void ajouter(int data);
     int somme();
     double moyenne();
     int min();
     int max();
     int variationMax();
     int taille();
     void afficher();
};
```

L'utilisateur ne doit pas pouvoir modifier les données => private

Toutes les méthodes et les constructeurs sont dans le cas présent accessibles => public

V. Objets, notions avancées

- Héritage et dérivation -

Dérivation et héritage

- L'*héritage*, également appelé *dérivation*, permet de définir une *nouvelle classe* en se basant sur une classe déjà existante nommée aussi la « *classe de base* »,
  - La classe de base à partir de laquelle on hérite est aussi appelée aussi « *classe mère* » dans la littérature,
  - Réutilisation totale ou partielle des classes déjà développées,
    - Gain de temps et d'énergie pour le développement et la maintenance des codes sources (les méthodes identiques ne sont codées qu'une seule fois dans la classe mère).

Dérivation et héritage - Exemple

```
class A {
    int dataA;          // l'attribut dataA et la méthode fA()
    int fA();           // appartiennent à la classe "A"
};

class B: public A {
   int fB( ){                              // Dans la classe "B" nous utilisons
       return (dataA / 2) + fA();          // nativement l'attributs et la méthode
   }                                       // "A" et "B" dont nous héritons
};

int main( void ){
    B *mon_objet = new B();          // On crée une instance de "B", puis
    int val_1 = mon_objet->fA();     // on accède aux méthodes définies
    int val_2 = mon_objet->fB();     // dans la classe "A" et "B" sans
}                                    // faire de distinction.
```

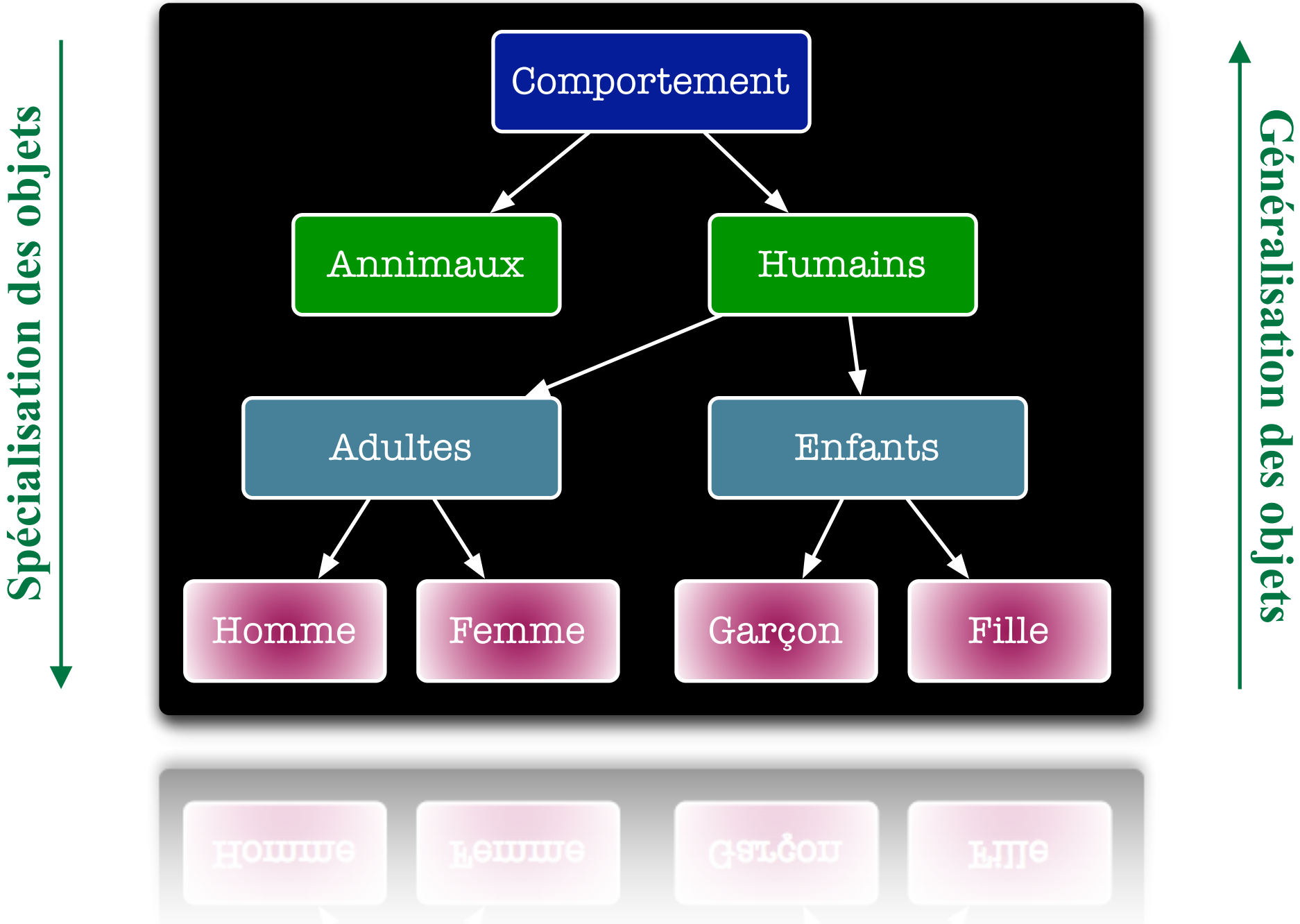
Dérivation et héritage

- L'*héritage* permet de donner à une classe toutes les caractéristiques d'une ou de plusieurs autres classes,
  - Les classes dont elle hérite sont appelées *classes mères*, *classes de base* ou *classes antécédentes*,
  - La classe elle-même est appelée *classe fille*, *classe dérivée* ou *classe descendante*,
- Les *propriétés héritées* héritées par la classe fille sont : l'ensemble des attributs et des méthodes de la classe de base.

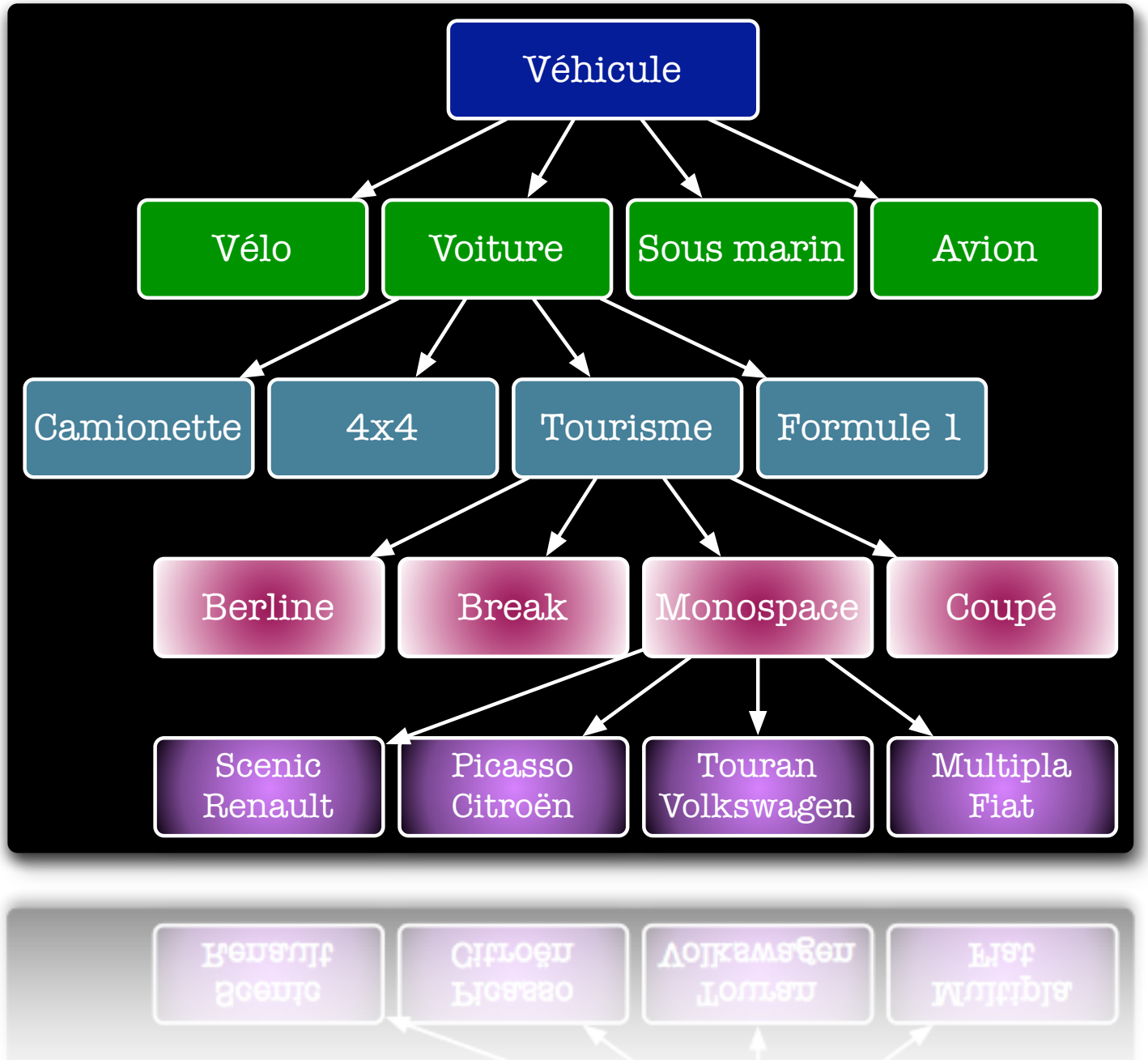
Dérivation et héritage

- Possibilité après un héritage d'ajouter de nouveaux membres à la classe ou de redéfinir des méthodes,
  - On s'assure ainsi une compatibilité avec l'interface de la classe dont nous héritons,
    - Compatibilité des objets entre eux,
    - Standardisation des interfaces entre développeurs,
  - Cela permet aussi de réaliser un développement progressif en fonction de la complexité de l'objet final,
    - On peut ainsi modifier les objets de bas niveau (optimiser l'implémentation par exemple) sans avoir à modifier les objets reposants dessus.

Dérivation et héritage – Exemple (1)



Dérivation et héritage – Exemple (2)



Dérivation et héritage - Sémantique

- **Syntaxe**
  - La classe “B” hérite de façon publique de la classe “A”.
  - Tous les membres publics ou protégés de la classe “A” font partis de l'interface de la classe “B”.
- **Exemple**

```
Extension *e;
e = new Extension(2, 1);
int somme = e->sum( );
int diff  = e->sub( );
```

```
class Couple{
protected:
    int a,b;

public:
    Couple(int _a, int _b)
        a = _a;
        b = _b;
    }

    int sum( ){
        return (a+b);
    }
}

class Extension : public Couple{
    int sub( ){
        return (a-b);
    }
}
```

Encapsulation ou Dérivation ?

- Comment savoir si je dois dériver une classe ou l'encapsuler ?
  - Une question à se poser : *« est-ce que X est un genre de Y, ou est-ce que X utilise un Y ? »*

1. Si la réponse est *« X est un genre de Y »*, il s'agit d'un cas où je dérive une classe.
2. Si la réponse est *« X utilise Y »*, il s'agit d'un cas où je vais encapsuler une classe.

Dérivation et héritage - Droits d'accès

- Lors de la définition d'un héritage, le concepteur spécifie explicitement ou intrinsèquement le mode de dérivation :
  - ***Public***, ***Protected*** ou ***Private***.
- Le mode d'héritage détermine quels sont les méthodes et les attributs de la classe de base qui seront accessibles dans la classe dérivée,
- Au cas où aucun mode d'héritage n'est spécifié, le compilateur C++ prend par défaut : *private*,
- Les *membres privés* de la classe de base *ne sont jamais accessibles* par les membres des classes dérivées.

Dérivation et héritage - Droits d'accès

- Héritage public
  - Il donne aux membres publics et protégés de la classe de base le même statut dans la classe dérivée.
- Cas d'Utilisation
  - C'est la forme la plus courante d'héritage, car il permet de modéliser les relations *"Y est une sorte de X"* ou *"Y est une spécialisation de la classe de base X"*.

```
class Vehicule {
    public:      void pub1();
    protected:   void prot1();
    private:     void priv1();
};

class Voiture : public Vehicule
{
  public:
    int pub2() {
      pub1();       // Possible
      prot1();      // Possible
      priv1();      // Impossible
    }
};

Voiture safrane;
safrane.pub1();  // Possible
safrane.pub2();  // Possible
```

Dérivation et héritage - Droits d'accès

- Héritage privé
  - Il donne aux membres publics et protégés de la classe de base le statut de membres privés dans la classe dérivée.
- Cas d'Utilisation
  - Il permet de modéliser les relations *"Y est composé de un ou plusieurs X"* .
  - Type d'héritage peu employé
    - Plutôt que d'hériter de façon privée de la classe de base X, on peut faire de la classe de base une donnée membre.

```
class String {
public:

   int length();
    // ...
};

class Telephone_number:
                  private String {
   void f1() {
     // ...
     l = length();      // OK
    // ...
   }
};

Telephone_number tn;
cout << tn.length();   // ERREUR
```

Dérivation et héritage - Droits d'accès

- Héritage protégé
  - Il donne aux membres publics et protégés de la classe de base le statut de membres protégés dans la classe dérivée,
  - L'héritage fait partie de l'interface mais n'est pas accessible aux utilisateurs,
- Cas d'Utilisation
  - Lorsque l'on souhaite que des méthodes soient accessibles aux futures déviations de la classe (développeurs).

```
class String {
protected:
    int n;
};

class Telephone_number:
                protected String {
protected:
    void f2() { n++; }  // OK
};

class Local_number:
          public Telephone_number {
protected:
    void f3() { n++; } // OK
};
```

Dérivation et héritage - Surcharge de méthodes

- Surcharger une méthode
  - Cela correspond à une redéfinition une fonction dans une classe dérivée.
  - Il est nécessaire que la méthode possède le même nom et les même attributs que dans la classe de base.
- Objectif
  - Redéfinir le comportement d'une méthode pour l'objet courant tout en respectant l'interface de l'objet dont on hérite.

```
class Constant{
public:
    int getValue()
    { return 0; }    // retourne 0
};

class Constant_1 : Constant{
public:
    int getValue(){
        return 1;    // retourne 1
    }
};

class Constant_2 : Constant{
public:
    int getValue(){
        return 2;    // retourne 2
    }
};
```

Dérivation et héritage - Surcharge de méthodes

- Il est possible de choisir quelle méthode on désire exécuter à l'aide de l'opérateur de portée « :: »
  - Exécuter la méthode surchargée,
  - Exécuter la méthode de la classe de base (dont on hérite),

```
class Constant{
public:
     int getValue()
     { return 0; }
}

class Constant_1 : Constant{
public:
     int getValue(){
          return 1;
     }

     int Compute(){
          int a = getValue()
          // retourne 1
          int b = Constant::getValue()
          // retourne 0
     }
}
```

Dérivation et héritage - Ajustement des droits

- Lors d'un héritage protégé ou privé, nous pouvons spécifier que certains membres de la classe ancêtre conservent leur mode d'accès dans la classe dérivée.
- Ce mécanisme, appelé déclaration d'accès, ne permet en aucun cas d'augmenter ou de diminuer la visibilité d'un membre de la classe de base.

```
class X {
public:
    void f1();
    void f2();

protected:
    void f3();
    void f4();
};

class Y : private X {
public:
    X::f1;   // f1() reste public dans Y
    X::f3;   // ERREUR: un membre protégé
             // ne peut pas devenir public
protected:
    X::f4;   // f3() reste protégé dans Y
    X::f2;   // ERREUR: un membre public
             // ne peut pas devenir protégé
};
```

Dérivation et héritage – Construc. et Destruc.

- Les constructeurs, constructeur de copie, destructeurs et opérateurs d'affectation ne sont jamais hérités.
- Les constructeurs par défaut des classes de bases sont *automatiquement appelés* avant le constructeur de la classe dérivée.
- Pour ne pas appeler les constructeurs par défaut, mais des constructeurs avec des paramètres, vous devez employer une *liste d'initialisation*.
- L'appel des destructeurs se fera dans l'ordre inverse des constructeurs.

```
class Animal{
public:
     Animal()
          { cout<< " Animal " << endl; }
     ~Animal()
          { cout<< "~ Animal " << endl; }
};

class Chien : public Animal{
public:
     Chien()
          { cout<< " Chien " << endl; }
     ~Chien()
          { cout<< "~ Chien " << endl; }
};

void main(){
     Chien *york = new Chien ();
     // ... ... ... ...
     delete york;
}
```

Dérivation et héritage – Construc. et Destruc.

```
class Animal{
public:
     Animal()
          { cout<< " Animal " << endl; }
     ~Animal()
          { cout<< "~ Animal " << endl; }
};

class Chien : public Animal{
public:
     Chien()
          { cout<< " Chien " << endl; }
     ~Chien()
          { cout<< "~ Chien " << endl; }
};

void main(){
     Chien *york = new Chien ();
     // ... ... ... ...
     delete york;
}
```

Voici ci-dessous la sortie qui apparaîtrait dans le terminal lors de l'exécution du programme défini à gauche.

Exemple de sortie dans un terminal lors de l'utilisation du programme :

```
./Mon_Programme
Animal
Chien
~Chien
~Animal
```

Question 28 : Héritage de la classe ListData

➜ Ecrivez la classe ListData_Positive qui n'accepte que des nombre entiers positifs comme valeurs

```
class ListData_Positive : public ListData_v2
{
public:
     ListData_Positive(int data);
     ListData_Positive(ListData_Positive *liste);
     ~ListData_Positive();

     void ajouter(int data);
};
```

Question 28 : Héritage de la classe ListData

➜ Ecrivez la classe ListData_Positive qui n'accepte que des nombre entiers positifs comme valeurs

```
ListData_Positive::ListData_Positive(int data)
{
     cout << "Construction d'une ListData_Positive" << endl;
     ajouter( data );
}

ListData_Positive::~ListData_Positive()
{
     cout << "Destruction d'une ListData_Positive" << endl;
}

void ListData_Positive::ajouter(int data)
{
     if( data < 0 ){
          sum    += data;
          nombre += 1;
          minimum = (minimum<data)?minimum:data;
          maximum = (maximum>data)?maximum:data;
     }else{
          cout << "ERREUR : La donnee est negative" << endl;
     }
}
```

Question 29 : Héritage de la classe ListData

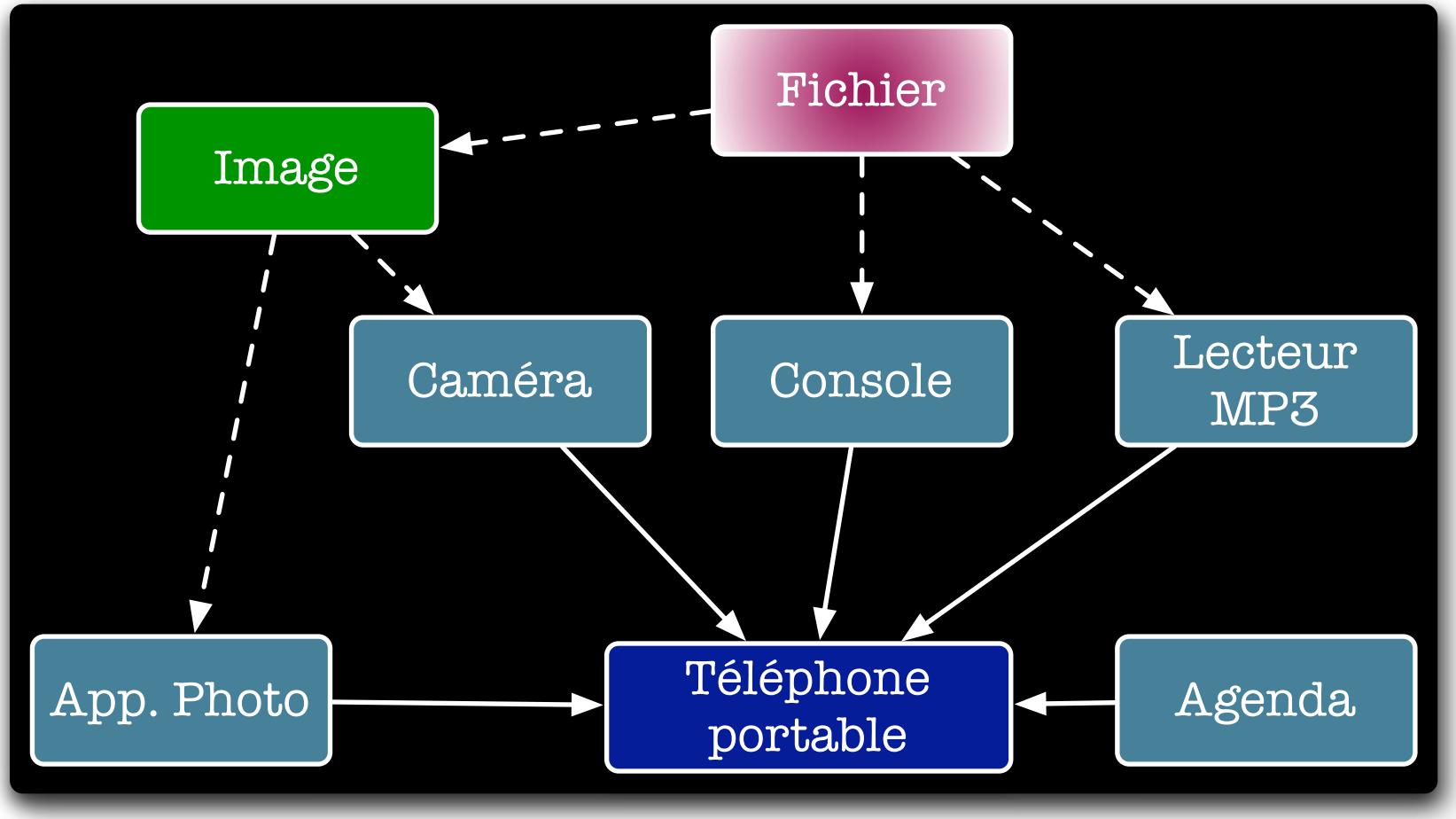
- Ecrivez un programme "main" permettant d'instancier dynamiquement votre objet.
- Ensuite ajouter 4 données à la liste (3,9,13, 0).
- Calculer la moyenne des données.
- Ajouter la donnée (0).
- Récupérez la valeur minimum présente dans la liste.

```
int main() {
     ListData_Positive *ld = new ListData_Positive();
     ld->ajouter(  3 );
     ld->ajouter(  9 );
     ld->ajouter( 13 );
     ld->ajouter(  0 );
     cout << "Calcul de la moyenne      : " << ld->moyenne() << endl;
     ld->ajouter( 0 );
     cout << "Calcul de la valeur min   : " << ld->min() << endl;
     delete ld;
     return 0;
}
```

V. Objets, notions avancées

- Héritage multiple -

Héritage multiple - Introduction



Héritage multiple - Introduction

```
class A {
    int dataA;                  // l'attribut dataA appartient a la
};                              // classe "A"

class B {
    int dataB;                  // l'attribut dataB appartient a la
};                              // classe "B"

class C: public B, public A {
    int somme( ){                   // Dans la classe "C" nous utilisons
        return dataA + dataB;       // nativement les attributs des classes
    }                               // "A" et "B" dont nous héritons
};
```

Héritage multiple - Introduction

- Le langage C++ permet de réaliser des héritages multiples,
  - Pour chaque classe de base, on peut définir le mode d'héritage,
- Complexification de la compréhension du code source,
  - Nécessité de lever les ambiguïtés s'il existe un recouvrement entre les classes (sur le nom des méthodes et des attributs).

```
class A {
public:
    void fa(){ /* ... */ }
protected:
    int x;
};

class B {
public:
    void fb(){ /* ... */ }
protected:
    int x;
};

class C: public B, public A{
public:
    void fc(){
        int i;
        fa();
        i = A::x + B::x;
    }
};
```

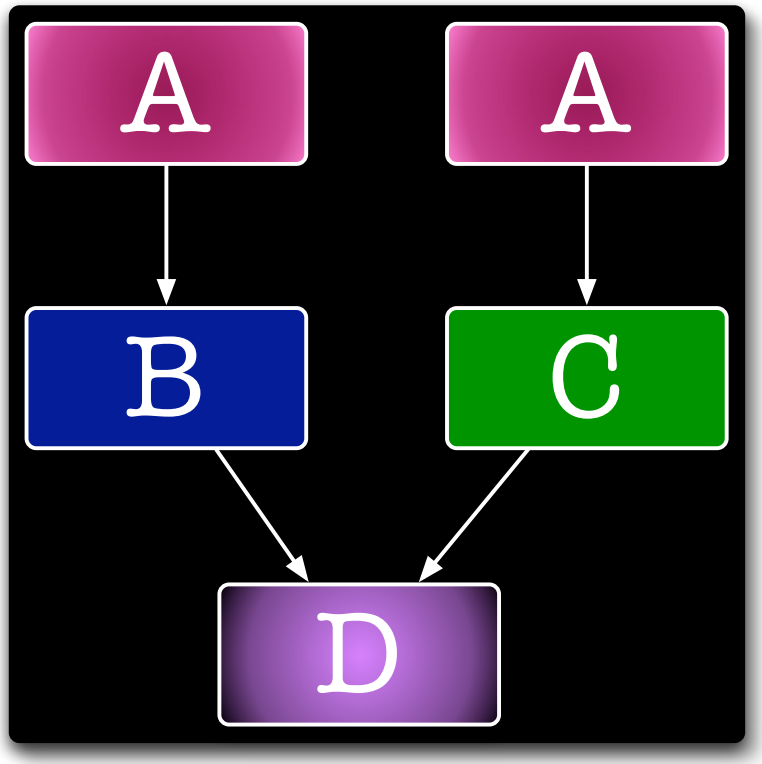
Héritage multiple - Définition

- Appel des constructeurs lors d'un héritage multiple,
  - Les constructeurs sont appelés dans l'ordre de déclaration de l'héritage,
  - Les destructeurs sont appelés dans l'ordre inverse de celui des constructeurs,
- Exemple ci-contre
  - Le constructeur par défaut de la classe ***C*** appelle le constructeur par défaut de la classe ***B***, puis celui de la classe ***A*** et en dernier lieu le constructeur de la *« classe dérivée »*,
  - Cela est identique, même si une liste d'initialisation existe.

```
class A {
public:
    A(int n=0) { /* ... */ }
};

class B {
public:
    B(int n=0) { /* ... */ }
};

class C: public B, public A {
public:
    C(int i, int j): A(i), B(j) { /*...*/ }
};

void main() {
    C objet_c;
    // appel des constructeurs
    // B(), A() et C()
}
```

Héritage multiple – Problème !

- Un objet de la classe ***D*** contiendra deux fois les données héritées de ***A***,
  - une fois par héritage de la classe ***B***,
  - une autre fois par héritage de ***C***.
- Il y a donc deux fois le membre *attr* dans la classe ***D***.
  - Pour accéder à *attr* de la classe mère ***A*** il faut lever l'ambiguïté.

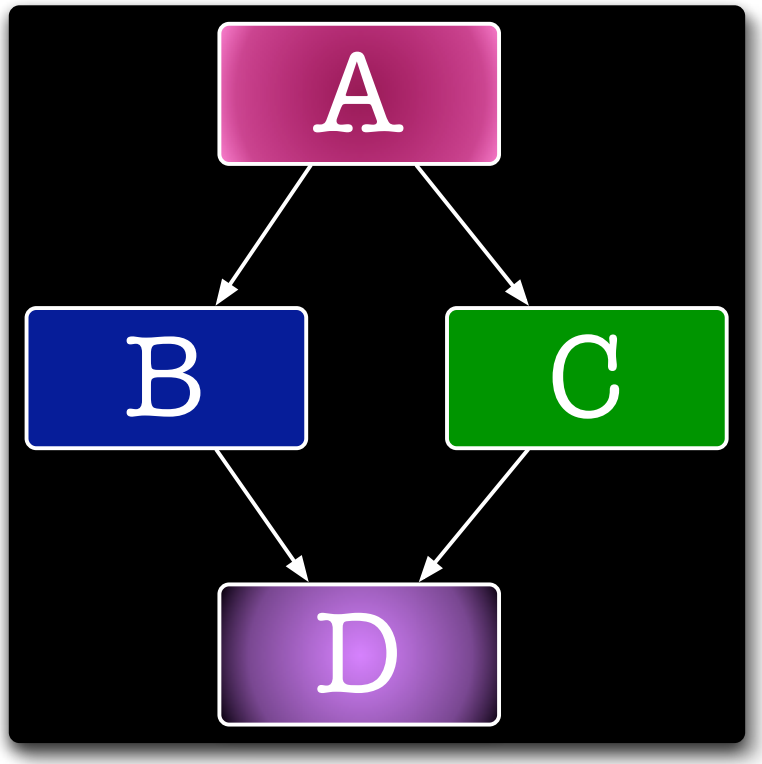



```
void main() {
    D obj;
    obj.attr = 0;     // Ambiguité (erreur)
    obj.B::attr = 1; // OK
    obj.C::attr = 2; // OK
}
```

Héritage multiple - Héritage virtuel

- On souhaite une unique occurrence des membres de la classe mère
  - Pour que la classe "*D*" n'hérite qu'une seule fois de la classe "*A*", il faut que les classes "*B*" et "*C*" **héritent virtuellement** de "*A*".

```
class A
    { int attr; }
class B : virtual public A
    { /* … */ }
class C : virtual public A
    { /* … */ }
class D : public B : public C
    { /* … */ }
```



Une seule occurrence de « A » composera la classe « D », supprimant l'ambiguïté.

V. Objets, notions avancées

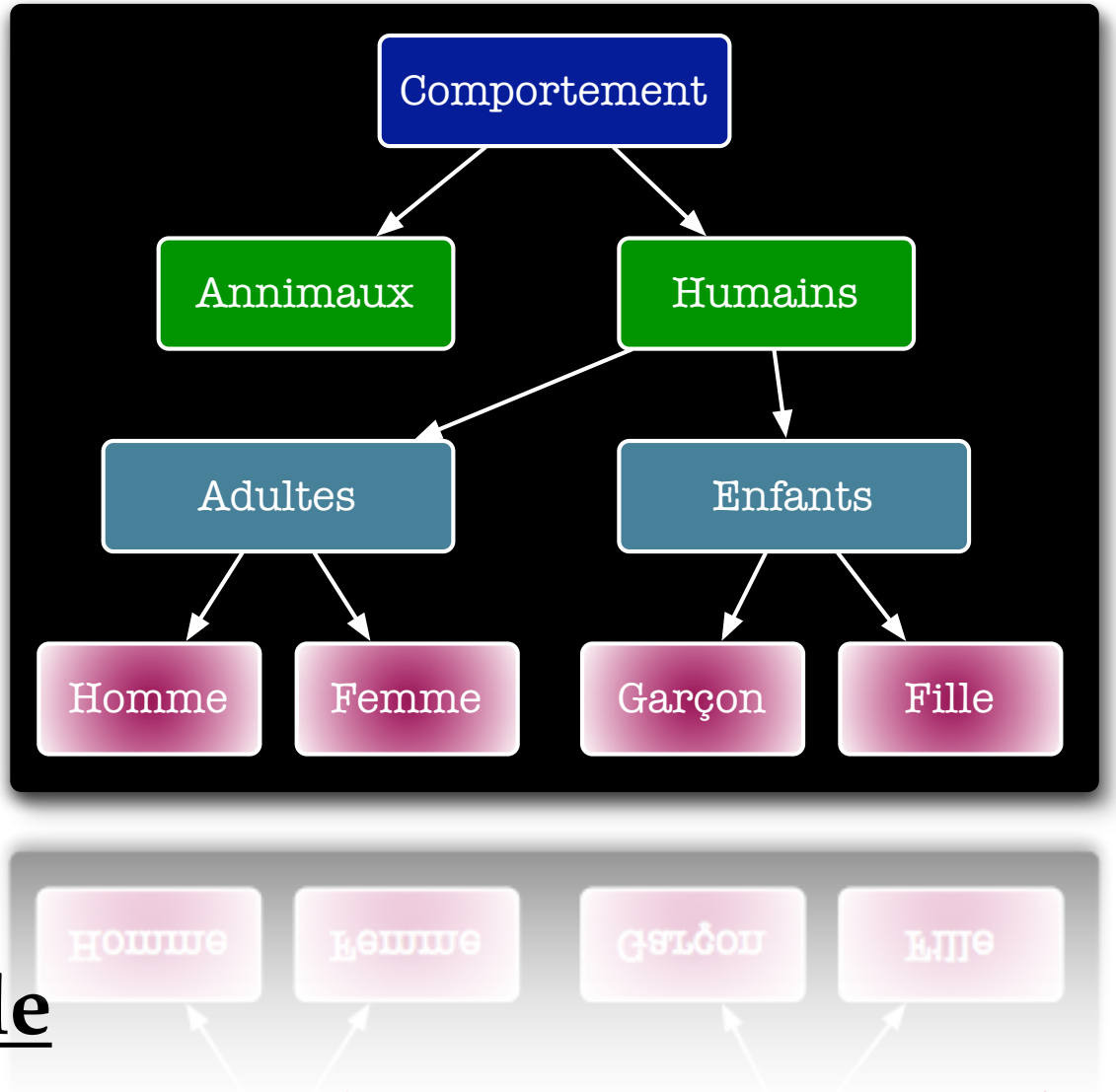
- Compatibilité entre classes -

Compatibilité entre objets

- Une conversion implicite d'une instance d'une classe dérivée en une instance de la classe de base est possible si l'héritage est **public**.

- L'inverse est interdit car le compilateur ne saurait pas comment initialiser les membres supplémentaires de la classe dérivée

Compatibilité entre objets



Exemple

1. Un *enfant* est un être *humain*, un *garçon* est un *enfant*,
2. Un être *humain* n'est pas nécessairement un *enfant* et n'est pas nécessairement un *garçon*.

Compatibilité entre objets

```
class Vehicule {
public:
    void accelerer();
};

class Voiture : public Vehicule {
public:
    int accelerer();
};

void demarrer(Vehicule v) {
    v.accelerer();    // OK;
}

void main( ){
    Voiture porche;
    demarrer(porche);
}
```

Une voiture est un type particulier de véhicule. Un véhicule possède une méthode lui permettant de démarrer donc une voiture possède obligatoirement une méthode démarrer (surcharger ou non)

Compatibilité entre objets

```
int main( void ){
    Voiture **liste = (Voiture**)new Voiture[10];

    liste[0] = new Voiture();
    liste[1] = new Renault();
    liste[2] = new Porche();
    liste[3] = new Peugeot();
    liste[4] = new Jaguar();

    for(int i=0; i<5; i++){
        liste[i]->demarrer();
    }

    for(int i=0; i<5; i++){
        delete liste[i];
    }

    delete liste;
    return 0;
}
```

Dans un tableau de voiture nous pouvons rentrer tous types de classes filles par contre la méthode exécuter par "démarrer" dépend du type de la méthode : virtuelle ou non.

Question 30 : Compatibilité de la classe ListData

➜ Ecrivez un programme permettant de sommer un ListData avec une ListeData_Positive

```
int main() {
     ListData_v2 *l1 = new ListData_v2( 3 );
     l1->ajouter( -1 );

     ListData_Positive *l2 = new ListData_Positive( 6 );
     l2->ajouter( 4 );

     l1->addition( l2 );

     delete l1, l2;
     return 0;
}
```

V. Objets, notions avancées

- Les classes abstraites -

Les classes Abstraites - Introduction

- Il arrive souvent que la méthode virtuelle définie dans la classe de base serve de *cadre générique* pour les méthodes virtuelles des classes dérivées,
  - Ceci permet de garantir une *bonne homogénéité* de votre architecture de classes,
- Une classe est dite *abstraite* si elle contient au moins une *méthode virtuelle pure*,
  - On ne peut pas créer d'instance d'une classe abstraite,
  - Une classe abstraite ne peut pas être utilisée comme argument ou type de retour d'une fonction.

Les classes Abstraites - Introduction

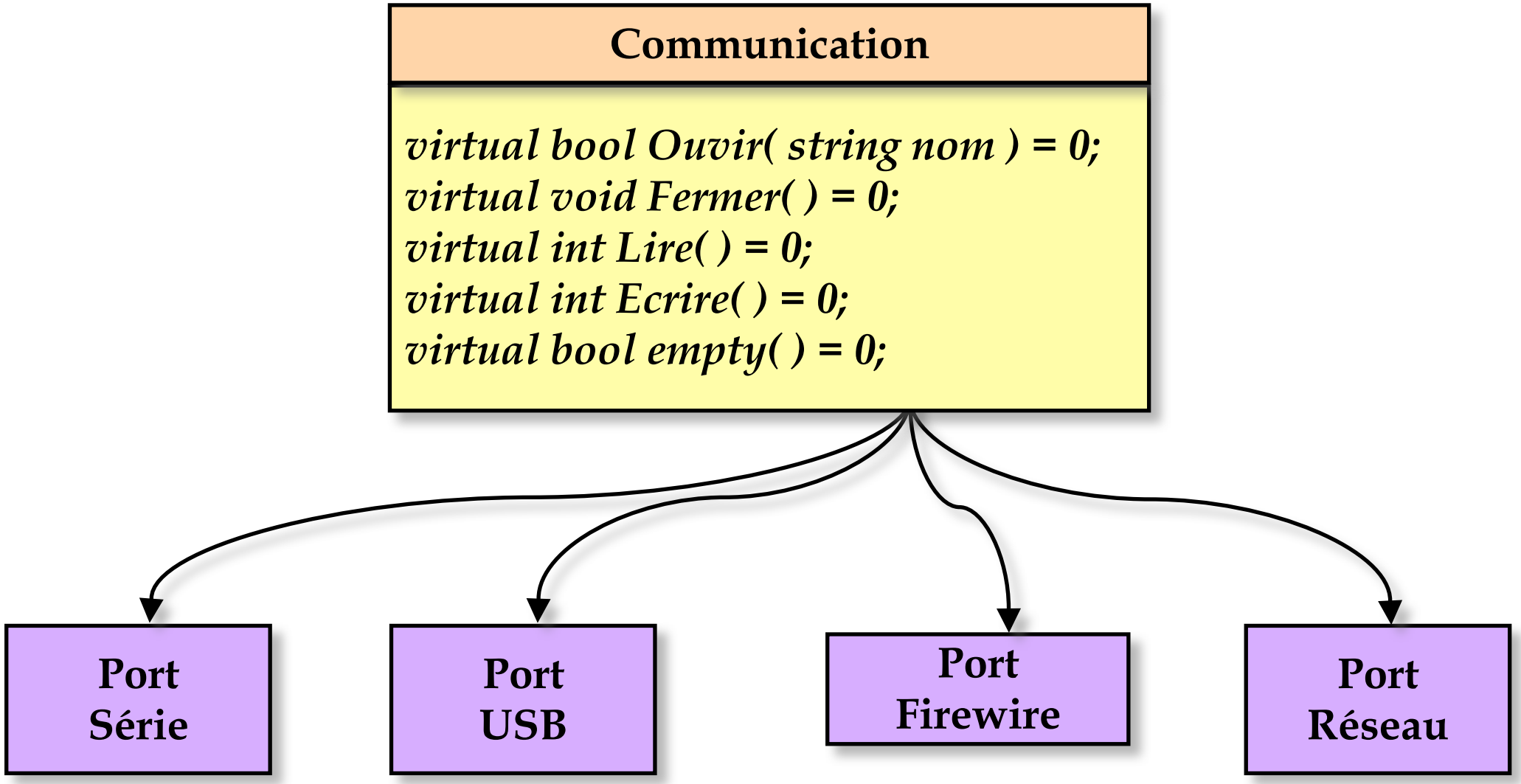
- Une *méthode virtuelle pure* est une méthode qui est déclarée mais *non définie* dans une classe. Elle est définie dans une des classes dérivées de cette classe.

- Pour déclarer une *méthode virtuelle pure* dans une classe, il suffit de faire suivre sa déclaration de *« =0 »*. La fonction doit également être déclarée virtuelle :

  ↳ ***virtual type*** nom(paramètres) ***=0***;

- L'information « =0 » signifie ici simplement qu'il n'y a pas d'instance de cette méthode dans cette classe.

Les classes Abstraites - Exemple



Les classes Abstraites - Exemple

```
Communication *com;

if( mode = "IEEE" ){
    com = new IEEE( "..." );
}else if( mode = "Serial" ){
    com = new SerialPort( "..." );
}else{
    // ......
}

com->open( );
com->write( 10 );
```

La classe communication est un "modèle" définissant ce que ses filles doivent posséder au minimum

Ici le choix de la classe peut dépendre d'un choix de l'utilisateur mais le programme est générique

Résumé (Fonctions virtuelles / classes abstraites)

- Comme nous venons de le voir, le mot clef « virtuel » peut avoir plus significations en fonction de son contexte d'utilisation :

1. Dans le cadre d'un *héritage de classe*
   - ⇨ Cela indique au compilateur que les données de la classe mère dont on hérite de manière virtuelle ne doit pas être dupliquée dans les classes filles,
2. Dans le cadre de la *définition d'une méthode*
   - ⇨ Cela implique que la classe ne peut pas être instanciée, et que les classes en dérivant doivent surcharger cette méthode sinon elles seront-elles aussi virtuelles,

V. Objets, notions avancées

- Gestion des exceptions -

Les exceptions - Problématique

- Les exceptions sont la pour gérer les erreurs liées à l'exécution du programme :
  - *Erreurs matérielles* : saturation mémoire, disque plein, rupture d'une connexion réseau, etc.
  - *Erreurs logicielles* : division par zéro, débordement dans un tableau, etc.
- La solution habituellement pratiquée est *l'affichage d'un message d'erreur* et l'arrêt du programme avec le renvoi d'un code d'erreur au programme appelant.

Les exceptions – Gestion classique des erreurs

- Gestion classique des erreurs à l'exécution dans un programme développé en langage C.

```
1  void lire_fichier(const char *nom) {
2    FILE f = fopen( nom ); // ouverture en lecture
3    if ( f == NULL ) {
4      cerr << "Problème à l'ouverture" << nom << endl;
5      exit(1);
6    }
7    // lecture du fichier ...
8  }
```

Les exceptions - Implémentation C++

- Une *exception* est une interruption de l'exécution du programme à la suite d'un événement particulier. Le but des exceptions est de réaliser des traitements spécifiques aux événements qui en sont la cause :
  - Ces traitements peuvent rétablir le programme dans son mode de fonctionnement normal, auquel cas son exécution reprend,
  - Il se peut aussi que le programme se termine, si aucun traitement n'est approprié,
- Lorsqu'une erreur survient, le programme doit *lancer une exception*. L'exécution normale du programme s'arrête et le contrôle est passé à un *gestionnaire d'exception*.
- *Dissociation de l'écriture des méthodes permettant de traiter les données des gestionnaires d'erreurs,*

Les exceptions - Implémentation C++

- Nouvelle structure de contrôle permettant la gestion des erreurs d'exécution, la structure ***try ... Catch***

```
try {
   // ...
 throw objet
   // lancement de l'exception
} catch ( type ) {
   // traitement de l'erreur
}
```

1. Construction d'un objet (d'un type quelconque) qui représente l'erreur
2. Lancement de l'exception *(throw)*
3. Propagation de l'exception dans la structure de contrôle *try ... catch* englobante
4. Cette structure essaye attraper *(catch)* l'objet
5. Si elle n'y parvient pas, la fonction *terminate()* est appelée.

Les exceptions - Syntaxe de la structure

- **catch (TYPE)** : intercepte les exceptions du type TYPE, ainsi que celles de ses classes dérivées.
- **catch (TYPE e)** : intercepte les exceptions du type TYPE, ainsi que celles de ses classes dérivées.
  - Dans le catch un objet « e » est utilisable pour extraire d'autres informations sur la nature de l'exception.
- **catch (...)** : intercepte les exceptions de tous types, non traitées par les blocs ***catch*** précédents.

```
try {
   // ...
   throw objet
   // lancement de l'exception
} catch ( type ) {
   // traitement de l'erreur
}
```

Types d'exception :

- **Xalloc** : allocation mémoire,
- **bad_alloc** : idem,
- **Bad_cast** : gestion des transtypage de classe,
- … … …

Les exceptions – Exemple

- L'exemple présente une allocation mémoire importante qui a de fortes chances d'être impossible !

```
#include <exception.h>
// ...
try {
  char *ptr = new char[1000000000];

}catch ( bad_alloc ) {
  // en cas d'échec d'allocation mémoire par new
  // une exception bad_alloc est lancée par new
  // traitement de l'erreur d'allocation}
  cout << "Erreur lors de l'allocation mémoire !" << endl;
}
```

Les exceptions - Exception personnalisée

```
#include <iostream>
#include <sstream>
#include <exception>

class NombreInvalide : public std::exception{
public:
  NombreInvalide ( ){
  this->msg = “Le nombre est invalide”;
}

virtual ~NombreInvalide () throw(){ }

virtual const char * what() const throw{
  return this->msg.c_str();
}

private:
  string msg;
};
```

Les exceptions - Exception personnalisée

```
class Generique : public std::exception{
public:
    Generique ( const char * Msg, int Line ){
        std::ostringstream oss;
        oss << "Erreur ligne " << Line << " : " << Msg;
        this->msg = oss.str();
    }

    virtual ~ Generique () throw(){ }

    virtual const char * what() const throw{
        return this->msg.c_str();
    }

private:
    std::string msg;
};
```

Les exceptions - Exception personnalisée

```
// Dans cette fonction nous allons nous meme provoquer
// une exception afin de terminer le traitement en cours si
// la valeur du parametre est incorrect (different des hypotheses)

bool setAngleDegre(int angle){
  try {
    if( (angle < 0) || (angle >360)){
        throw NombreInvalide();
    }
  }catch ( const exception & e ) {
    std::cerr << e.what() << "\n";
  }
}
```

Les exceptions - Exception générique

```
// Dans cette fonction nous allons nous même provoquer
// une exception "personnalisee" lorsque nous rencontrons un probleme
// le message passe au constructeur s'affichera en même temps que le
// numero de la ligne ou le probleme est apparu.

bool setValue(int a){
  try {
    if( a < 0 ){
      throw my_exception( "exception test", __LINE__ );
    }
  }catch ( const std::exception & e ) {
    std::cerr << e.what() << "\n";
  }
}
```

Déclarer les exceptions « autorisées » pour une f(x)

- Il est possible de spécifier les exceptions qui peuvent apparaître dans une fonction. Pour cela, il faut faire suivre son en-tête du mot-clé “throw”, avec entre parenthèses, et séparées par des virgules, les classes des exceptions envisageables.

- Exemple

```
int fonction_sensible(void) throw (int, double, erreur){
    /* Code contenu dans le cœur de la fonction qui ne
        devrait generer que les exceptions prevues … */
}
```

- Cette fonction n'a le droit de lancer que des exceptions du type “int”, “double” ou “erreur”. Si une autre exception est lancée, par exemple une exception du type “char *”, il se produit encore une fois une erreur à l'exécution.

Déclarer les exceptions « autorisées » pour une f(x)

- La liste des exceptions autorisées dans une fonction ne fait pas partie de sa signature,
  - Elles n'intervient donc pas dans les mécanismes de surcharge des fonctions,

- La déclaration de la liste des exceptions autorisées dans la méthode doit se faire :
  - Après le mot-clé « *const* » dans les déclarations de fonctions membres « *const* »,
  - Et avant « *=0* » dans le cas des déclarations des *fonctions virtuelles pures*.

V. Objets, notions avancées

~ Les flux de données ~

Les flux de données - Généralités

- Un *flux* (ou *stream*) est une abstraction logicielle représentant un flot de données entre :
  - Une source produisant de l'information,
  - Une cible consommant cette information,
- Un flux peut être vu comme un "buffer" et les mécanismes associés à celui-ci. Il prend en charge, quand le flux est créé, l'acheminement de ces données,
  - Comme pour le langage C, les instructions entrées/sorties ne font pas partie des instructions du langage,
  - Elles appartiennent à une librairie standardisée qui implémente les flux à partir de classes.

Les flux de données - Généralités

- Par défaut, chaque programme C++ peut utiliser 3 flots de données distincts :
  - « *cout* » qui correspond à la sortie standard
  - « *cin* » qui correspond à l'entrée standard
  - « *cerr* » qui correspond à la sortie standard d'erreur.
- Pour utiliser d'autres flots, il est nécessaire de les créer et de les attacher des fichiers :
  - Fichiers normaux sur le disque dur,
  - Fichier spéciaux (drivers matériels, sockets réseaux, …),
  - Ou à des tableaux de caractères.

Les flux de données – Intérêt des flux

- Vitesse d'exécution plus rapide :
  - la fonction *printf* doit analyser à l'exécution la chaîne de formatage, tandis qu'avec les flots, la traduction est faite à la compilation.
- On peut utiliser les flux avec des types d'objet définis par l'utilisateur (si surcharge des opérateurs >> et <<).
- Vérification de type : pas d'affichage erroné car le compilateur choisit automatiquement la conversion à réaliser.

```
#include <stdio.h>
#include <iostream.h>

void main() {
  int i=1234;
  double d=567.89;
  printf("i= %d d= %d\n", i, d);
  cout << "i= " << i << " d= " << d;
}
```

```
/* Résultat de l'exécution **/
i= 1234      d= -5243
i= 1234      d= 567.89
/*****************************/
```

Les flux de données – les classes de base

- Les classes d'entrées sorties standards :
  - « **istream** » : classe dérivée de *ios* pour les flots en entrée,
  - « **ostream** » : classe dérivée de *ios* pour les flots en sortie,
  - « **iostream** » : classe dérivée de *istream* et de *ostream* pour les flots bidirectionnels,
- Les classes pour la manipulation de fichiers
  - « **ifstream** » : classe permettant d'effectuer des entrées à partir des fichiers,
  - « **ofstream** » : classe permettant d'effectuer des sorties sur des fichiers,
  - « **fstream** » : classe permettant d'effectuer des entrées/sorties à partir des fichiers.

Les flux – Le flot de sortie « ostream »

- Il fournit des sorties formatées et non formatées (dans un *streambuf*)
- Il gère les types prédéfinis du langage C++
- On peut (doit) le surcharger pour ses propres types afin de simplifier l'utilisation de l'objet par la suite :
  - Pour afficher les informations (débogage),
  - Sauvegarder les données dans un fichier par exemple.

```
class Client {
public :
      friend ostream &operator<<(ostream
                        &os, const Client &ex);

private :
  char _nom[20];
  int  _age;
};

ostream &operator<<(ostream &os,
                        const Client &x) {
  return os << x._nom << " " << x.age;
}

void main() {
  Exemple e1;
  cout << "Exemple = " << e1 << endl;
}
```

Les flux – Le flot d'entrée « istream »

- Il fournit des entrées formatées et non formatées (dans un *streambuf*),
- Il gère les types prédéfinis du langage C++,
- On peut (doit) le surcharger pour ses propres types afin de simplifier l'utilisation de l'objet par la suite :
  - Pour lire des données à partir du clavier (débogage),
  - Lire des données sauvegardées dans un fichier par exemple,
- Par défaut >> ignore tous les espaces (voir ios::skipws).

```
class Client {
public :
  friend istream &operator>>(istream
                             &is, Client &ex);
private :
  char _nom[20];
  int  _valeur;
};

istream &operator>>(istream &is,
                                Client &x){
  return is >> x.nom >> x.valeur;
}

void main() {
  Client y;
  cout << "Nom et une valeur : ";
  cin >> y;
}
```

Les flux - Formatage de l'information

- Chaque flot conserve en permanence un ensemble d'indicateurs spécifiant l'état du formatage,
  - Ceci permet de donner un comportement par défaut au flot, contrairement aux fonctions *printf()* et *scanf()* du langage C, dans lesquelles on fournissait pour chaque opération d'entrée/sortie l'indicateur de format.

```
class ios {
  enum {
    left,       // justifie les sorties a gauche
    right,      // justifie les sorties a  droite
    dec,     // conversion en décimal
    … … … … … … … … … … … … …
    hex,        // conversion en hexadécimal
    scientific, // notation 1.234000E2 avec les réels
  }
};
```

Les flux - Formatage de l'information

- Ces valeurs peuvent se combiner avec l'opérateur | comme dans l'exemple :
  - ios::showbase | ios::showpoint | ios::showpos
- Des constantes sont aussi définies dans la classe *ios* pour accéder à un groupe d'indicateurs :
  - **static const long basefield;** : permet de choisir la base (*dec*, *oct* ou *hex*)
  - **static const long adjustfield;** permet de choisir son alignement (*left*, *right* ou *internal*)
  - **static const long floatfield;** permet de choisir sa notation (*scientific* ou *fixed*)
- Les méthodes suivantes (définies dans la classe *ios*) permettent de lire ou de modifier la valeur des indicateurs de format :
  - **long flags()** : retourne la valeur de l'indicateur de format
  - **long flags(long f)** : modifie l'ensemble des indicateurs en fonction de la valeur de *f*.
- Exemple

```
cout << setf(ios::dec, ios::basefield) << i;
cout << setf(ios::left, ios::adjustfield)<< hex << 0xFF;
// affiche : 000xFF
```

Les flux - Contrôler de l'état d'un flux

- La classe *ios* décrit les aspects communs des flots d'entrée et de sortie. C'est une classe de base virtuelle pour tous les objets flux. Vous utiliserez ses méthodes pour tester l'état d'un flot ou pour contrôler le formatage des informations.

- **Méthodes de la classe de base *ios* :**

  - **int good( )** : retourne une valeur différente de zéro si la dernière opération d'entrée/sortie s'est effectuée avec succès et une valeur nulle en cas d'échec.

  - **int fail( )** : fait l'inverse de la méthode précédente

  - **int eof( )** : retourne une valeur différente de zéro si la fin de fichier est atteinte et une valeur nulle dans le cas contraire.

  - **int bad( )** : retourne une valeur différente de zéro si vous avez tenté une opération interdite et une valeur nulle dans le cas contraire.

  - **int rdstate( )** : retourne la valeur de la variable d'état du flux. Retourne une valeur nulle si tout va bien.

  - **void clear( )** : remet à zéro l'indicateur d'erreur du flux. C'est une opération obligatoirement à faire après qu'une erreur se soit produite sur un flux.

Les flux – Ouvrir des fichiers

- Il est possible de créer un objet flot associé à un fichier autre que les fichiers d'entrées/sorties standards.
- Pour cela on emploi les 3 classes de base permettant l'accès aux données : « **ifstream** », « **ofstream** » et « **fstream** ».
- L'ouverture d'un fichier est réalisé à l'aide de la méthode « **open** » commune aux 3 classes,

```
#include <fstream.h>

int main( void ){
    ifstream f1;                    // instanciation de l'objet
    f1.open("essai1.tmp");          // ouverture du flux de données
    ofstream f2("essai2.tmp");      // combinaison des opérations
    // ... ... ... ... ...
}
```

V. Objets, notions avancées

- La généricité (les templates) -

La généricité - Besoin et concept

- Nous avons vu précédemment comment réaliser des structures de données relativement indépendantes de la classe de leurs données avec les classes abstraites,
- La surcharge permet d'avoir des méthodes gérant plusieurs types de données,
  - Emploi des classes abstraites est assez fastidieux,
  - La surcharge n'est pas généralisable pour tous les types de données,
- Paramètres génériques, que l'on appelle encore paramètres *template*,
  - Un paramètre *template* est soit un type générique, soit une constante dont le type est assimilable à un type intégral.
  - Comme leur nom l'indique, les paramètres *template* permettent de paramètrer la définition des fonctions et des classes.

La généricité - Besoin et concept

- Les fonctions *template* sont donc des fonctions qui peuvent travailler sur des objets dont le type est un type générique (c'est-à-dire un type quelconque), ou qui peuvent êtres paramétrés par une constante de type intégral.
  - Les classes *template* sont des classes qui contiennent des membres dont le type est générique ou qui dépendent d'un paramètre intégral.

- La génération du code a lieu lors de la compilation du code source. Les types génériques sont remplacés par les vrais types donnés en argument. Cette opération s'appelle l'instanciation des *templates*.

- *Les mécanismes de gestion des templates sont gérés à la compilation et non à l'exécution ou tous les types de données sont connus et statiques.*

La généricité - Les Templates

- Les paramètres *« template »* sont :
  - Soit des types génériques
  - Soit des constantes dont le type peut être assimilé à un type intégral.

```
template <class T = int>
class Pile_FIFO{
…
};
```

- Les paramètres *« template »* qui sont des types peuvent prendre des valeurs par défaut, en faisant suivre le nom du paramètre d'un signe égal et de la valeur par défaut.
  - Ici, la valeur par défaut doit évidemment être un type déjà déclaré.

La généricité - Les Templates, Exemple

- Déclaration d'une fonction avec « template »

```
template <class T>
T Min(T x, T y){
  return x<y ? x : y;
}

template <class T>
T Max(T x, T y){
  return x>y ? x : y;
}
```

- La définition de la fonction est précédée par la déclaration des paramètres « templates »

Classe générique (données internes)

```
template <class Type>
Class Complexe {
  Type reel;
  Type imag;

  Complexe(Type _reel,Type imag){
     reel = _reel;
     imag = _imag;
  }

 // ... ... ... ... ...
}

int main( void ){
    Complexe<float> f(0.2, -3.14)
    Complexe<int>   s* new Complexe<int>(0, 0);
    // ... ... ... ... ...
    return 0;
}
```

La généricité – Mise en œuvre

- Exemples d'instanciations une seule et unique classe vecteur utilisant le mécanisme des *templates* :

```
Tableau<double> liste;
liste = new Tableau<double>(10);
liste->set(3.1415, 0);
double d = liste->get(0);

Tableau<Objet> liste(10);
liste.set(new Objet(), 1);
Objet o = liste.get(1);

Tableau<Tableau> liste;
liste = new Tableau<Tableau>(10);
... ... ... ... ... ... ... ...
```

```
template <class Type = int>
class Tableau{
private:
    int taille;
    Type data;

public:
    Tableau( int __taille)
        taille = __taille;
        data = new Type[taille];
    }

    Type get(int position){
        return data[position];
    }

    Type set(Type obj, int position){
        data[position] = obj;
    }
};
```

La généricité - Conclusion

- Avantages
  - ✔ Permet un taux de réutilisation important,
  - ✔ Réduction du code à maintenir,
  - ✔ Uniformisation des objets (on utilise un nombre réduit d'objets différents pour réaliser les mêmes taches),
- Inconvénients
  - ✘ La définition des « templates » peut être différentes en fonction des compilateurs C++ utilisés (Gcc, Visual C++),
  - ✘ Obligation de regrouper les fichiers "h" et "cpp".

Question 31 : Généricité de la classe ListData

➜ *Ecrivez la classe ListDataX qui peut travailler sur tout type de données à l'aide des mécanismes de templates. Adaptez aussi les méthodes.*

```
template <class T>
class ListDataX
{
public:
     T sum;
     T nombre;
     T minimum;
     T maximum;

     ListDataX();

     ListDataX(T data);
     ListDataX(ListDataX<T> *liste);
     ~ListDataX();

     void ajouter(T data);
     T somme();
     double moyenne();
     T min();
     T max();
     T variationMax();
     int  taille();
     void afficher();
};
```

Question 31 : Généricité de la classe ListData

```
template<typename T>
ListDataX<T>::ListDataX()
{
     sum     = (T)0;
     nombre  = (T)0;
     minimum = (T)INT_MAX;
     maximum = (T)INT_MIN;
}

template<typename T>
ListDataX<T>::ListDataX(ListDataX<T> *liste)
{
     sum     = liste->somme();
     nombre  = liste->taille();
     minimum = liste->min();
     maximum = liste->max();
}

template<typename T>
ListDataX<T>::ListDataX(T data)
{
     sum     = (T)0;
     nombre  = (T)0;
     minimum = (T)INT_MAX;
     maximum = (T)INT_MIN;
     ajouter( data );
}
```

```
template<typename T>
void ListDataX<T>::ajouter(T data)
{
     sum    += data;
     nombre += 1;
     minimum = (minimum<data)?minimum:data;
     maximum = (maximum>data)?maximum:data;
}
```

```
template<typename T>
T ListDataX<T>::somme()
{
     return sum;
}
```

```
template<typename T>
double ListDataX<T>::moyenne()
{
     return (double)(somme())/(double)taille();
}
```

Question 31 : Généricité de la classe ListData

```
template<typename T>
ListDataX<T>::~ListDataX()
{
}
```

```
template<typename T>
T ListDataX<T>::max()
{
     return maximum;
}
```

```
template<typename T>
T ListDataX<T>::variationMax()
{
     T var = max()-min();
     return var;
}
```

```
template<typename T>
T ListDataX<T>::min()
{
     return minimum;
}
```

Question 32 : Utilisation de la généricité

➜ *Mettez en oeuvre cette nouvelle classe.*

```
int main() {
     ListDataX<long>   l1(   2);
     ListDataX<float>  l2( 2.4);
     ListDataX<double> l3(3.14);

     // ... ... ... ...
     l1->ajouter( 4    );
     l2->ajouter( -1.0 );
     l3->ajouter( ...  );
     // ... ... ... ...

     delete l1;
     delete l2;
     delete l3;
     return 0;
}
```

6

“ Conclusion sur le développement objets ”

Possibilités et limites

- La programmation objet permet comme vous avez pu l'apprécier permet de simplifier le développement d'applications logicielles :
  - Regroupant données et méthodes de traitement,
  - Vérification et sécurisation des affectation des données aux ayants droits uniquement,
  - Réutilisation facile des objets déjà développés et testés,
  - Définition d'interfaces de classe afin de normaliser les méthodes nécessaires pour une classe,
- *Alors est-ce la panacée pour autant ?*

Possibilités et limites

- La programmation orientée objets (POO) a aussi ses inconvénients :
  - ✘ Temps d'exécution supérieur à une solution équivalente écrite en C (allocation mémoire + temps de copie de l'instance en mémoire + temps d'exécution des constructeur),
  - ✘ Coût mémoire important, car en plus des données en mémoire pour chaque objet, nous avons aussi la zone mémoire dédiées aux méthodes associées à l'objet !

- Exemple
  - Si un objet (un vecteur) à 64 octets de données et 2ko de méthodes associées pour manipuler les données,
  - Quelle est sa rentabilité en mémoire ?

7

“ Les bibliothèques d’objets ”

VII. Les bibliothèques d'objets

- Standard Template Library -

Standard Template Library

- Tout comme pour le langage C, pour lequel un certain nombre de fonctions ont été définies et standardisées et constituent la bibliothèque C, une bibliothèque de classes et de fonctions a été spécifiée pour le langage C++.

- Cette bibliothèque est le résultat de l'évolution de plusieurs bibliothèques, parfois développées indépendamment par plusieurs fournisseurs d'environnements C++, qui ont été fusionnées et normalisées afin de garantir la portabilité des programmes qui les utilisent.

- Une des principales briques de cette bibliothèque est sans aucun doute la *STL* (abréviation de « *Standard Template Library* »), à tel point qu'il y a souvent confusion entre les deux.

Standard Template Library

- Les algorithmes et toutes les classes fournies par la bibliothèque sont susceptibles de travailler sur des données de type arbitraire.
  - La bibliothèque utilise donc complètement la notion de *template*, et se base sur plusieurs abstractions des données manipulées et de leurs types afin de rendre générique l'implémentation des fonctionnalités.
- De plus, la bibliothèque utilise le mécanisme des exceptions afin de signaler les erreurs qui peuvent se produire lors de l'exécution des méthodes de ses classes et de ses fonctions.
- Enfin, un certain nombre de notions algorithmiques avancées sont utilisées dans toute la bibliothèque.

Standard Template Library - String

- La classe *« string »* a été développée afin de faciliter la manipulation des chaînes de caractères,
- Cette classe permet :
  - De bénéficier de toutes les fonctions développées dans la *STL* sur les fonctions de base.
    - Il est possible de lire une ligne d'un fichier sans avoir à se préoccuper de sa taille.
- Un travail important de gestion et de vérification est fait en toute transparence, ce qui rend le code plus maintenable.
  - *« string »* est donc beaucoup plus simple et sûre d'utilisation que les nombreuses fonctions utilisant les *« char* »*.

Standard Template Library - String

- Liste non exhaustives des méthodes utiles :
  - ***size( )*** : renvoie le nombre de caractères composant la « string »,
  - ***c_str( )*** : retourne un pointeur sur une chaîne de caractères de type (char*),
  - ***find( )*** : recherche la première occurrence du paramètre (chaine ou caractère) dans l'objet et retourne sa position,
  - ***insert( )*** : insertion de caractère(s) dans une chaîne déjà existante,
  - ***erase( )*** : permet de supprimer un caractère ou une sous chaîne de l'objet.

Standard Template Library - String

```cpp
#include <stdlib.h>
#include <iostream>

using namespace std;

int main( ){
    // création de la chaine "Hello"
    string message = "Hello";

    // concaténation de "Word !"
    message += " World !";

    // affichage de "Hello World !"
    cout << message << endl;
}
```

Standard Template Library - String

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

int main( ) {
     istringstream iss( "mot1;mot2;mot3;mot4" );
     string mot;
          while ( std::getline( iss, mot, ';' ) ){
     cout << mot << endl;
     }
}
```

Standard Template Library - Vector

- La classe *« Vector »* permet d'implémenter de manière générique un tableau à taille variable dans le temps.
  - Cette structure de base peut être utilisée afin de gérer l'ensemble des structures itératives dont vous avez besoin.
  - Attention tout de même à la complexité algorithmique des opérations que vous allez opérer dessus !
- *Afin de choisir le conteneur le plus adapté à vos besoin référez vous à la partie suivante.*

Standard Template Library - Vector

```
// créer un tableau d'entiers vide
std::vector<int> v;

// ajouter l'entier 10 a la fin
v.push_back( 10 );

// afficher le premier élément (10)
cout << v.front() << '\n';

// afficher le dernier élément (10)
cout << v.back() << '\n';

// enlever le dernier élément
v.pop_back(); // supprime '10'

// le tableau est vide
if ( v.empty() )
{
    cout << "Tout est normal\n";
}
```

```
// redimensionner le tableau resize()
// initialise tous les nouveaux entiers
v.resize( 10 );

// quelle est sa nouvelle taille ?
cout << v.size() << '\n'; // affiche 10

// on peut accéder directement
// aux 10 premiers éléments
v[ 9 ] = 5;

// intitialiser tous les éléments a 100
std::fill( v.begin(), v.end(), 100 );

// vider le tableau
v.clear(); // size() == 0
```

Standard Template Library - Vector

```
// allouer au moins 50 éléments
v.reserve( 50 );

// vérifier la taille du vecteur
cout << v.size() << '\n';

// nombre d'éléments qu'il peut
// stocker sans devoir réallouer
cout << v.capacity() << endl;

for ( int i = 0; i < 50; ++i )
{
    // grace a reserve() on économise
    // de multiples réallocations
    // successives
    v.push_back( i );
}

// afficher la nouvelle taille
cout << v.size() << '\n';
```

```
// tronquer le tableau a 5 élments
v.resize( 5 );

// les trier par ordre croissant
std::sort( v.begin(), v.end() );

// parcourir le tableau
for (size_t i=0, i<v.size(), ++i )
{
    // attention : avec l'operateur []
    // les acces ne sont pas verifies !
    cout << v[ i ] << '\t';
}
cout << '\n';

 // at() : les acces sont vérifiés
try{
    // acces en dehors des limites
    v.at( v.size() ) = 10;
}catch ( const std::out_of_range & ){
    cout << "Exception out_of_range\n";
}
```

Standard Template Library - Itérateurs

- Les itérateurs (*iterator*) sont une généralisation des pointeurs,
  - Les itérateurs sont des objets qui pointent sur d'autres objets,
  - Comme leur nom l'indique, les itérateurs sont utilisés pour parcourir une série d'objets de telle façon que si on incrémente l'itérateur, il désignera l'objet suivant de la série,
  - La majorité des structures de données de la STL possèdent des itérateurs,
- Possibilité de définir ses propres itérateurs dans vos classes afin de simplifier l'écriture du code par la suite :
  - Le code source C++ ensuite écrit tendra vers la description algorithmique...

Standard Template Library - Itérateurs

- Ces objets permettent de se rapprocher du niveau de description algorithmique :

  Pour tous *les objets « i » de la liste « v »* ***faire***
  Afficher « i »
  Fin de Pour

- Récupération d'un itérateur à partir d'un objet de type « vector »

```
vector<int>::iterator i;
for (i= v.begin( ); i != v.end(); ++i ){
    cout << *i << endl;
}
```

Standard Template Library - Algorithm

- Des algorithmes génériques sont aussi inclus dans la STL afin d'effectuer des actions usuelles sur les objets standardisés :
  - Trier (selon une fonction de coût),
  - Somme de "n" éléments,
  - Somme conditionnelle,
  - Inversion / permutation
  - Fusionner
  - Rechercher un élément
  - Rechercher les valeurs Min et Max,
  - Etc …

VII. Les bibliothèques d'objets

- Structures de données -

Les structures de données

- Toutes les applications que vous développerez effectueront toujours les mêmes taches :
  - Mémoriser des informations
  - Traiter ces informations
  - Stocker les résultats, etc…
- Afin d'accélérer les traitements il est nécessaire d'employer des structures de données adaptées !
  - Permet des optimisations de 90% du temps d'exécution dans certains cas …

Les structures de données

- **Structures de données séquencielles**
  - collection rangée linéairement (accès par indices)
    - ***vector<T>*** optimisé pour insertion et suppression à la queue, temps d'accès uniforme à tous les éléments,
    - ***deque<T>*** optimisé pour insertion et suppression à la tête et à la queue (double ended queue), temps d'accès uniforme à tous les éléments,
    - ***list<T>*** adapté à des insertions et suppressions n'importe où.

http://www.pottsoft.com/home/stl/stl.htmlx

Les structures de données

- **Structures de données associatives**
  - : collection rangée par clé (accès par clé),
    - ***set <K,comparer>*** recherche rapide, pas de doublon,
    - ***multiset <K,comparer>*** recherche rapide, doublons autorisés,
    - ***map <K,T,comparer>*** table (bijection), recherche rapide avec clé, pas de doublon,
    - ***multimap <K,T,comparer>*** table, recherche rapide avec clé, doublons autorisés.

http://www.pottsoft.com/home/stl/stl.htmlx

- **Adaptateurs de conteneurs**
  - : adaptation de conteneurs
    - ***stack<T,S>*** restriction LIFO d'une séquence (Last In First Out),
    - ***queue<T,S>*** restriction FIFO d'une séquence (First In First Out),
    - ***priority_queue<T,S,K>*** sortie d'éléments basée sur la priorité.

http://www.pottsoft.com/home/stl/stl.htmlx

Exemple d'utilisation de la classe *STACK*

```
#include <vector>
#include <stack>

int main( int argc, char *argv[] ){
    stack< const char *, vector<const char *> > s;
    //  Push on stack in reverse order
    s.push("order");
    s.push("correct");
    s.push("the");
    s.push("in");
    s.push("is");
    s.push("This");

    //  Pop off stack which reverses the order
    while ( !s.empty() ) {
      cout  << s.top() << " ";
      s.pop();
    } cout << endl;
    return( EXIT_SUCCESS );
}
```

http://www.pottsoft.com/home/stl/stl.htmlx#Stack

Exemple d'utilisation de la classe *QUEUE*

```
#include <iostream>
#include <queue>

int main( int argc, char *argv[] ){
    queue< const char * > s;
    //  Push on stack in correct order
    s.push("First");
    s.push("come");
    s.push("first");
    s.push("served");
    s.push("- why");
    s.push("don't");
    s.push("bars");
    s.push("do");
    s.push("this ?");

    //  Pop off front of queue which preserves the order
    while ( !s.empty() ) {
      cout  << s.front() << " "; s.pop();
    } cout << endl;
    return( EXIT_SUCCESS );
}
```

http://www.pottsoft.com/home/stl/stl.htmlx

Exemple d'utilisation de la classe *PRIORITY_QUEUE*

```
class TaskObject {
public:
    friend class PrioritizeTasks;
    friend ostream &amp;amp; operator<<( ostream &amp;amp;os, TaskObject &task);
    TaskObject( const char *pname = "", unsigned int prio = 4 ){
      process_name = pname;
      priority = prio;
    }
private:
    unsigned int priority;
    string process_name;
};

// Friend function for "printing" TaskObject to an output stream
ostream & operator<<( ostream &os, TaskObject &task ){
    os << "Process: " << task.process_name << " Priority: " << task.priority;
    return ( os );
}
```

```
// Friend class with function object for comparison of TaskObjects
class PrioritizeTasks {
public :
    int operator()( const TaskObject &x, const TaskObject &y )
    {
      return x.priority < y.priority;
    }
};
```

Exemple d'utilisation de la classe *PRIORITY_QUEUE*

```
int main( int argc, char *argv[] )
{
    int i;
    priority_queue<TaskObject, vector<TaskObject>, PrioritizeTasks> task_queue;

    TaskObject tasks[] = { "JAF", "ROB", "PHIL", "JOHN"
                          ,TaskObject("OPCOM",6)  , TaskObject("Swapper",16)
                          ,TaskObject("NETACP",8) , TaskObject("REMACP",8) };

    for ( i = 0; i < sizeof(tasks)/sizeof(tasks[0]) ; i++ )
      task_queue.push( tasks[i] );

    while ( !task_queue.empty() ) {
        cout << task_queue.top() << endl; task_queue.pop();
    }
    cout << endl;

    return( EXIT_SUCCESS );
}
```

http://www.pottsoft.com/home/stl/stl.htmlx

VII. Les bibliothèques d'objets

- La bibliothèque QT -

La bibliothèque QT par Trolltech

- **Qt** est une bibliothèque logicielle offrant essentiellement des composants d'interface graphique (widgets), mais également des composants d'accès aux données, de connexions réseaux, de gestion des fils d'exécution, d'analyse XML, etc.

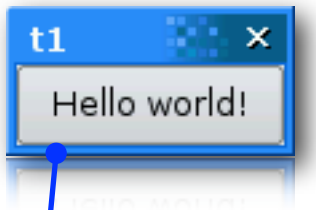
- Elle a été développée en C++ par la société Trolltech et est disponible pour les environnements Unix utilisant X11 (dont Linux), Windows et Mac OS.

- Qt utilise une version étendue du langage C++. Des "bindings" existent afin de pouvoir utiliser Qt à partir d'autres langages : Python, Ruby, langage C, langage Perl et Pascal.

- Cette bibliothèque sert de base à l'environnement de bureau KDE sous Linux. L'ensemble des programmes de cet environnement sont développés à l'aide des capacités de la bibliothèque Qt.

Utilisation d'objets existants - "Hello World"

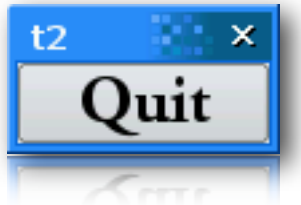
```
#include <QApplication>
#include <QPushButton>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    QPushButton hello("Hello world!");
    hello.resize(100, 30);

    hello.show();
    return app.exec();
}
```



Programme objets créant une fenêtre affichant le message "Hello World"

Utilisation d'objets existants - Fenêtre avec bouton

```
#include <QApplication>
#include <QFont>
#include <QPushButton>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    QPushButton quit("Quit");
    quit.resize(75, 30);
    quit.setFont(QFont("Times", 18, QFont::Bold));

    QObject::connect(&quit, SIGNAL(clicked()), &app, SLOT(quit()));

    quit.show();
    return app.exec();
}
```



Programme objets créant une fenêtre intégrant un bouton permettant de fermer l'application

Utilisation d'objets existants - Serveur HTTP

Petit programme objets implantant un serveur HTTP à l'aide des classes QT

```
class HttpDaemon : public QServerSocket{
      Q_OBJECT
public:
      HttpDaemon( QObject* parent=0 ) :
      QServerSocket(8080,1,parent), disabled(FALSE)
      { /* Appel au constructeur de la classe mere */}

      void newConnection( int socket ) {
            QSocket* s = new QSocket( this );
            connect( s, SIGNAL(readyRead()), this, SLOT(readClient()) );
            connect( s, SIGNAL(delayedCloseFinished()), this, SLOT(discardClient()) );
            s->setSocket( socket );
      }

private slots:
      void readClient() {
            QSocket* socket = (QSocket*)sender();
            if ( socket->canReadLine() ) {
                  QStringList tokens = QStringList::split( QRegExp("[ \r\n][ \r\n]*"), socket->readLine() );
                  if ( tokens[0] == "GET" ) {
                        QTextStream os( socket );
                        os.setEncoding( QTextStream::UnicodeUTF8 );
                        os << "HTTP/1.0 200 Ok\r\n"
                        "Content-Type: text/html; charset=\"utf-8\"\r\n"
                        "\r\n"
                        "<h1>Nothing to see here</h1>\n";
                        socket->close();
                        qService->reportEvent( "Wrote to client" );
                  }
            }
      }
};
```

http://doc.trolltech.com/solutions/4/qtservice/qtservice-example-server.html

Contrôler le développement & l'exécution

- Les différents types de problèmes que l'on peut rencontrer :
  - *Problèmes de conception,*
    - Problème dans le développement de votre code C++ qui ne correspond pas au fonctionnement prévu,
  - *Problèmes liés à l'utilisation,*
    - Vous avez récupéré une classe, une bibliothèque développé par une autre entité et cela ne fonctionne pas comme prévu,
  - *Problèmes liés à l'exécution,*
    - Le programme est correctement pensé et développé mais la machine sur lequel il fonctionne ne peut répondre aux besoins de l'application (mémoire libre insuffisante, etc.)

Validation fonctionnelle

- « ça marche ?! »
  - Qu'est ce que cette affirmation implique réellement, comment s'en assurer ?
- « Tester un objet »
  - Qu'est ce que cela implique ?
- « Validation d'une application »
  - Cela revient il a tester tous les objets ?
- Les méthodes et les outils associés à cette problématique.