# Remise á niveau en C++
# Partie 1

Emanuel Aldea <emanuel.aldea@u-psud.fr>
http://hebergement.u-psud.fr/emi

M2-E3A spécialisation SETI

# Premier chapitre

## Introduction

- Organisation du cours

# Organisation du cours

Les grandes lignes (1) :

- Introduction
- **Extensions** C++ : références, etc.
- **Données** en C++ : types, durée de vie, allocation
- Approche **objet**, construction et destruction
- **Propriétés** : attributs, méthodes, pointeur **this**
- **Encapsulation** : droits d'accès, accesseurs
- Surcharge d'opérateurs, flux d'entrée / sortie C++
- Instances et membres **statiques**

# Organisation du cours

Les grandes lignes (2) :

- **Héritage** simple, multiple, virtuel
- **Polymorphisme** statique et dynamique
- Bases abstraites, interfaces, gestion polymorphique
- **Templates** (patrons) de classes et de fonctions
- Bibliothèques C++ : STL, boost, OpenCV etc.
- **Exceptions** : fonctionnalité, gestion

# Origines et approche C++

## Apparition :

- **Extension syntactique** du langage C (un compilateur C++ compile du C)
- Travail entamé en 1979 par Bjarne Stroustrup
- Standardisation : ANSI/ISO '98, '03, '07, '11, '14, '17
- Principale extension : **les classes**
- Différence importante dans l'**architecture** du programme et dans l'**approche** de programmation
- Compréhension du langage :
  - $\Leftarrow$ compréhension de la réalité "physique" des données
  - $\Leftarrow$ compréhension du flot des données
  - $\Leftarrow$ essentiel en robotique ou en informatique industrielle

# Extensions C++

- Allocation dynamique (opérateurs **new** et **delete**)
- Nouveau type d'accès : la **référence**
- Fonctions : **décoration** et paramètres **par défaut**
- **Surcharge** des opérateurs
- Flots d'entrée / sortie C++ (**streams**)
- **Classes / objets** :
  - encapsulation
  - héritage
  - polymorphisme
- Gestion des **exceptions**

# Conception d'un programme C / C++

## Approche "classique" C

- Crée, écrit, lit et détruit des données
- Appelle des fonctions en passant et en recevant des paramètres

## Approche C++

- Crée et détruit des objets (toute donnée est un objet)
- Lit ou écrit les attributs d'objets
- Appelle les méthodes d'objets en passant et en recevant des paramètres

# Que fait le programme (réellement) ?

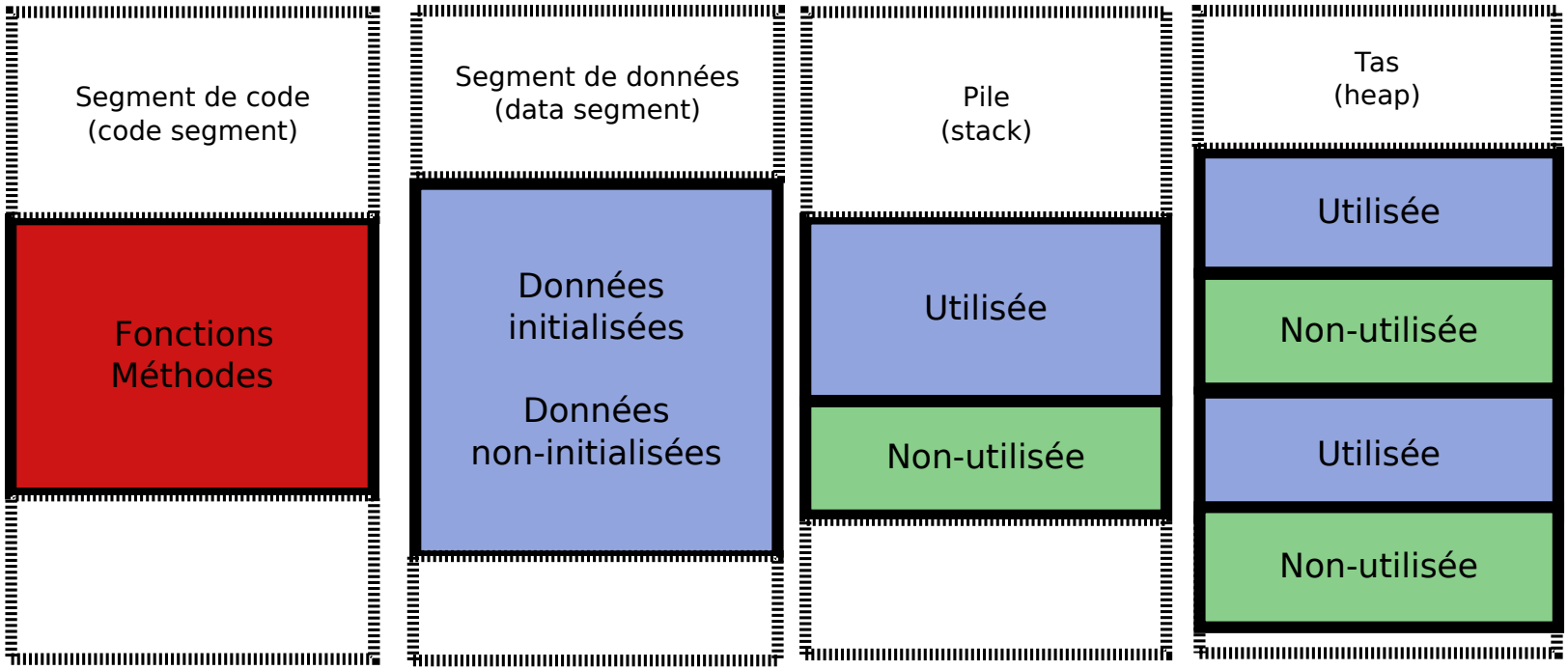
- Le programme effectue les actions désirées par le programmeur en s'exécutant dans un OS cible.
- Pour cela il faut avoir un **ficher exécutable** obtenu par compilation et édition de liens (link)
- Exécutable = **segment de code** + **segment de données**
- Segment de code = instructions assembleur (il n'y a pas d'interprétation ni de machine virtuelle)
- Exécution par un / plusieurs microprocesseurs des instructions assembleur en mode exclusif ou en même temps que d'autres programmes.

# Pour bien maîtriser les variables (objets)

Il faut savoir :

- C++ est un **langage typé** : toute variable et toute expression ont un **unique** type
- Faire la distinction entre **type** et **variable**
- Toute variable occupe une **place mémoire** de la taille du type pendant sa vie
- La **durée de vie** d'une variable
- L'**espace mémoire** dans lequel une variable vit
- **Où** et **comment** une variable est accessible
- Si l'évaluation d'une expression mène vers une variable (un conteneur) ou non : **l-value** et **r-value**

# Organisation mémoire proposée par C++




## Emplacement des données par défaut :

- Segment de données : variables **globales** et **statiques**
- Pile : variables **locales** et **temporaires**, **paramètres** d'appel et de retour des fonctions / méthodes
- Tas : variables dynamiques
- Registres micro-processeur :
  - les variables précédées par les mots clé **register** (interprété comme une suggestion)
  - toute variable locale, temporaire ou paramètre de fonction **inline** que le compilateur considère nécessaire, dès que l'optimisation est demandée (mode Release)

# Données : portée, accessibilité (lecture/écriture)

## Directe (par le nom) :

- **Globales** : dans le module courant, et dans les autres à partir de la redéclaration avec **extern**
- **Globales statiques** : dans le module courant
- **Locales** : jusqu'à la fin du bloc courant
- **Paramètres** : à l'intérieur de la fonction appelée
- **Dynamiques**, **de retour** ou **temporaires** : jamais (pas de nom !)
- Masquage : **locales** > **paramètres** > **attributs** > **globales**
- Espaces : **instruction** < **bloc** < **fonction/méthode** < **classe** < **namespace** < **module** < **programme**

# Données : portée, accessibilité (lecture/écriture)

## Indirecte :

- En évaluant une expression qui vaut la variable en question
- Par **adresse** : pointeur ou référence
- Dans une structure/classe/union : par `.` ou `->`
- Dans un tableau : arithmétique des pointeurs `*`, `+`, `-`, `[]`
- accès à un autre espace par l'opérateur de résolution de portée `::`

Le langage C++ ne garantit pas que l'évaluation d'une expression mène vers une location mémoire valide, ceci est la responsabilité du concepteur !

# Fonctions/méthodes : portée, accessibilité (appel)

## Directe :

- Par le nom de la fonction/méthode et ()
- Le nom seul d'une fonction est une adresse dont le type est déterminé par le prototype de la fonction

```
int f(int x){
  return x;
}
int main(){
  int (*fp)(int);
  fp = &f;
  (*fp)(5); //utilisation canonique
  fp(6);    //permis en C aussi, utilisation courante
  return 0;
}
```

## Indirecte :

- Expression dont l'évaluation mène à l'adresse d'une fonction, puis appel avec ()
- Pour une méthode il est impératif d'avoir comme expression un objet (ou adresse d'objet) valide, puis l'opérateur d'accès . ou -> et appel avec ()

# Données : accessibilité

```
static int i1=30;
double d2=3.14;
float f ( float f1){
  char c='d' ;
  static int st=7;
  add (&i1,( int)c );
  st += 5;
  return f1;
}
int main(){
  f(3);f(4);
  return 0;
}
```

main.cpp

```
extern double d2;



void add ( int* pa,int b){
  long var2=32145;
  for( int i=0;i<10;i++){
    double d2=i;
    d2+=* pa ;
  }
  var2=d2;
}
```

part2.cpp

# Données : accessibilité

```
static int i1=30;
double d2=3.14;
float f ( float f1){
  char c='d' ;
  static int st=7;
  add (&i1,( int)c );
  st += 5;
  return f1;
}
int main(){
  f(3);f(4);
  return 0;
}
```

main.cpp

```
extern double d2;



void add ( int* pa,int b){
  long var2=32145;
  for( int i=0;i<10;i++){
    double d2=i;
    d2+=* pa ;
  }
  var2=d2;
}
```

part2.cpp

La variable statique `i1` :

- variable globale, mais visible uniquement dans son module

# Données : accessibilité

```
static int i1=30;
double d2=3.14;
float f ( float f1){
  char c='d' ;
  static int st=7;
  add (&i1,( int)c );
  st += 5;
  return f1;
}
int main(){
  f(3);f(4);
  return 0;
}
```

main.cpp

```
extern double d2;



void add ( int* pa,int b){
  long var2=32145;
  for( int i=0;i<10;i++){
    double d2=i;
    d2+=* pa ;
  }
  var2=d2;
}
```

part2.cpp

La variable **statique** `i1` :

- variable **globale**, mais visible uniquement dans son module
- accessible par son adresse dans les autres modules

# Données : accessibilité

```
static int i1=30;
double d2=3.14;
float f ( float f1){
  char c='d' ;
  static int st=7;
  add (&i1,( int)c );
  st += 5;
  return f1;
}
int main(){
  f(3);f(4);
  return 0;
}
```

main.cpp

```
extern double d2;



void add ( int* pa,int b){
  long var2=32145;
  for( int i=0;i<10;i++){
    double d2=i;
    d2+=* pa ;
  }
  var2=d2;
}
```

part2.cpp

La variable statique `st` :

- variable locale, visible uniquement au niveau du bloc

# Données : accessibilité

```
static int i1=30;
double d2=3.14;
float f ( float f1){
  char c='d' ;
  static int st=7;
  add (&i1,( int)c );
  st += 5;
  return f1;
}
int main(){
  f(3);f(4);
  return 0;
}
```

main.cpp

```
extern double d2;



void add ( int* pa,int b){
  long var2=32145;
  for( int i=0;i<10;i++){
    double d2=i;
    d2+=* pa ;
  }
  var2=d2;
}
```

part2.cpp

La variable statique `st` :

- variable locale, visible uniquement au niveau du bloc
- valeur persistante entre les parcours du bloc. Où réside t elle en mémoire ?

# Données : accessibilité

```
static int i1=30;
double d2=3.14;
float f ( float f1){
  char c='d' ;
  static int st=7;
  add (&i1,( int)c );
  st += 5;
  return f1;
}
int main(){
  f(3);f(4);
  return 0;
}
```

main.cpp

```
extern double d2;



void add ( int* pa,int b){
  long var2=32145;
  for( int i=0;i<10;i++){
    double d2=i;
    d2+=* pa ;
  }
  var2=d2;
}
```

part2.cpp

La variable statique `st` :

- variable locale, visible uniquement au niveau du bloc
- valeur persistante entre les parcours du bloc. Où réside t elle en mémoire ?
- R : en data segment, soit dans la partie de données initialisées explicitement (i.e. `st`), soit dans la partie de données non-initialisées explicitement

# Données : accessibilité

```
static int i1=30;
double d2=3.14;
float f ( float f1){
  char c='d' ;
  static int st=7;
  add (&i1,( int)c );
  st += 5;
  return f1;
}
int main(){
  f(3);f(4);
  return 0;
}
```

main.cpp

```
extern double d2;



void add ( int* pa,int b){
  long var2=32145;
  for( int i=0;i<10;i++){
    double d2=i;
    d2+=* pa ;
  }
  var2=d2;
}
```

part2.cpp

La variable statique `st` :

- variable locale, visible uniquement au niveau du bloc
- valeur persistante entre les parcours du bloc. Où réside t elle en mémoire ?
- R : en data segment, soit dans la partie de données initialisées explicitement (i.e. `st`), soit dans la partie de données non-initialisées explicitement
- initialisation effectuée une seule fois

# Données : accessibilité

```
static int i1=30;
double d2=3.14;
float f ( float f1){
  char c='d' ;
  static int st=7;
  add (&i1,( int)c );
  st += 5;
  return f1;
}
int main(){
  f(3);f(4);
  return 0;
}
```

main.cpp

```
extern double d2;



void add ( int* pa,int b){
  long var2=32145;
  for( int i=0;i<10;i++){
    double d2=i;
    d2+=* pa ;
  }
  var2=d2;
}
```

part2.cpp

La variable statique `st` :

- variable locale, visible uniquement au niveau du bloc
- valeur persistante entre les parcours du bloc. Où réside t elle en mémoire ?
- R : en data segment, soit dans la partie de données initialisées explicitement (i.e. `st`), soit dans la partie de données non-initialisées explicitement
- initialisation effectuée une seule fois
- limiter l'utilisation des variables statiques locales

# Données : accessibilité

```
static int i1=30;
double d2=3.14;
float f ( float f1){
  char c='d' ;
  static int st=7;
  add (&i1,( int)c );
  st += 5;
  return f1;
}
int main(){
  f(3);f(4);
  return 0;
}
```

main.cpp

```
extern double d2;



void add ( int* pa,int b){
  long var2=32145;
  for( int i=0;i<10;i++){
    double d2=i;
    d2+=* pa ;
  }
  var2=d2;
}
```

part2.cpp

La variable globale `d2` :

- visible au niveau du module

# Données : accessibilité

```
static int i1=30;
double d2=3.14;
float f ( float f1){
  char c='d' ;
  static int st=7;
  add (&i1,( int)c );
  st += 5;
  return f1;
}
int main(){
  f(3);f(4);
  return 0;
}
```

main.cpp

```
extern double d2;



void add ( int* pa,int b){
  long var2=32145;
  for( int i=0;i<10;i++){
    double d2=i;
    d2+=* pa ;
  }
  var2=d2;
}
```

part2.cpp

La variable globale `d2` :

- visible au niveau du module
- visible également à l'extérieur du module avec une déclaration extern ...

# Données : accessibilité

```
static int i1=30;
double d2=3.14;
float f ( float f1){
  char c='d' ;
  static int st=7;
  add (&i1,( int)c );
  st += 5;
  return f1;
}
int main(){
  f(3);f(4);
  return 0;
}
```

main.cpp

```
extern double d2;




void add ( int* pa,int b){
  long var2=32145;
  for( int i=0;i<10;i++){
    double d2=i;
    d2+=* pa ;
  }
  var2=d2;
}
```

part2.cpp

La variable globale `d2` :

- visible au niveau du module
- visible également à l'extérieur du module avec une déclaration extern ...
- ...mais attention aux masquages

# Données : accessibilité

```
static int i1=30;
double d2=3.14;
float f ( float f1){
  char c='d' ;
  static int st=7;
  add (&i1,( int)c );
  st += 5;
  return f1;
}
int main(){
  f(3);f(4);
  return 0;
}
```

main.cpp

```
extern double d2;



void add ( int* pa,int b){
  long var2=32145;
  for( int i=0;i<10;i++){
    double d2=i;
    d2+=* pa ;
  }
  var2=d2;
}
```

part2.cpp

Le paramètre de fonction/méthode `f1` :

- visible uniquement au niveau de la fonction/méthode

# Données : accessibilité

```
static int i1=30;
double d2=3.14;
float f ( float f1){
  char c='d' ;
  static int st=7;
  add (&i1,( int)c );
  st += 5;
  return f1;
}
int main(){
  f(3);f(4);
  return 0;
}
```

main.cpp

```
extern double d2;



void add ( int* pa,int b){
  long var2=32145;
  for( int i=0;i<10;i++){
    double d2=i;
    d2+=* pa ;
  }
  var2=d2;
}
```

part2.cpp

Le paramètre de fonction/méthode `f1` :

- visible uniquement au niveau de la fonction/méthode
- durée de vie : destruction à la fin de la fonction/méthode

# Données : accessibilité

```cpp
static int i1=30;
double d2=3.14;
float f ( float f1){
  char c='d' ;
  static int st=7;
  add (&i1,( int)c );
  st += 5;
  return f1;
}
int main(){
  f(3);f(4);
  return 0;
}
```

main.cpp

```cpp
extern double d2;



void add ( int* pa,int b){
  long var2=32145;
  for( int i=0;i<10;i++){
    double d2=i;
    d2+=* pa ;
  }
  var2=d2;
}
```

part2.cpp

La variable compteur `i` :

- définie à l'intérieur de la boucle (for, while)

# Données : accessibilité

```
static int i1=30;
double d2=3.14;
float f ( float f1){
  char c='d' ;
  static int st=7;
  add (&i1,( int)c );
  st += 5;
  return f1;
}
int main(){
  f(3);f(4);
  return 0;
}
```

main.cpp

```
extern double d2;



void add ( int* pa,int b){
  long var2=32145;
  for( int i=0;i<10;i++){
    double d2=i;
    d2+=* pa ;
  }
  var2=d2;
}
```

part2.cpp

La variable compteur `i` :

- définie à l'intérieur de la boucle (for, while)
- attention : les compilateurs Microsoft considèrent la déclaration d'un compteur comme une déclaration de variable locale, et donc le compteur persistera au delà de la boucle respective

# Données : accessibilité

```
static int i1=30;
double d2=3.14;
float f ( float f1){
  char c='d' ;
  static int st=7;
  add (&i1,( int)c );
  st += 5;
  return f1;
}
int main(){
  f(3);f(4);
  return 0;
}
```

main.cpp

```
extern double d2;



void add ( int* pa,int b){
  long var2=32145;
  for( int i=0;i<10;i++){
    double d2=i;
    d2+=* pa ;
  }
  var2=d2;
}
```

part2.cpp

La variable compteur `i` :

- définie à l'intérieur de la boucle (for, while)
- attention : les compilateurs Microsoft considèrent la déclaration d'un compteur comme une déclaration de variable locale, et donc le compteur persistera au delà de la boucle respective
- l'option /Zc :forScope force le comportement standard (contexte restreint à l'intérieur de la boucle)

# Réflexion : variables globales, paramètres

- **Q** : envoyer fréquemment plusieurs variables d'intérêt en paramètre est encombrant. Comment faire mieux en C++ ?

# Réflexion : variables globales, paramètres

- **Q** : envoyer fréquemment plusieurs variables d'intérêt en paramètre est encombrant. Comment faire mieux en C++ ?
- **R** : regrouper les fonctions qui travaillent avec les mêmes variables en tant que méthodes au sein d'une classe avec une fonctionnalité bien définie.

# Réflexion : variables globales, paramètres

- **Q** : envoyer fréquemment plusieurs variables d'intérêt en paramètre est encombrant. Comment faire mieux en C++ ?
- **R** : regrouper les fonctions qui travaillent avec les mêmes variables en tant que méthodes au sein d'une classe avec une fonctionnalité bien définie.
- **R** : les variables communes seront déclarées comme membres privés de classe

# Réflexion : variables globales, paramètres

- **Q** : envoyer fréquemment plusieurs variables d'intérêt en paramètre est encombrant. Comment faire mieux en C++ ?
- **R** : regrouper les fonctions qui travaillent avec les mêmes variables en tant que méthodes au sein d'une classe avec une fonctionnalité bien définie.
- **R** : les variables communes seront déclarées comme membres privés de classe
- **Q** : pour quoi ne pas déclarer toutes les variables comme des variables globales ?

# Réflexion : variables globales, paramètres

- **Q** : envoyer fréquemment plusieurs variables d'intérêt en paramètre est encombrant. Comment faire mieux en C++ ?
- **R** : regrouper les fonctions qui travaillent avec les mêmes variables en tant que méthodes au sein d'une classe avec une fonctionnalité bien définie.
- **R** : les variables communes seront déclarées comme membres privés de classe
- **Q** : pour quoi ne pas déclarer toutes les variables comme des variables globales ?
- **R** : il y a un certain temps, c'était justement la pratique usuelle. Au fur et à mesure que les programmes sont devenus plus complexes, cela est devenu impossible de les déboguer car les données pouvaient être modifiées par toute fonction présente dans le programme. Des années d'expérience ont convaincu la communauté que l'accès aux données doit être le plus local possible.

# Réflexion : variables globales, paramètres

- **Q** : envoyer fréquemment plusieurs variables d'intérêt en paramètre est encombrant. Comment faire mieux en C++ ?
- **R** : regrouper les fonctions qui travaillent avec les mêmes variables en tant que méthodes au sein d'une classe avec une fonctionnalité bien définie.
- **R** : les variables communes seront déclarées comme membres privés de classe
- **Q** : pour quoi ne pas déclarer toutes les variables comme des variables globales ?
- **R** : il y a un certain temps, c'était justement la pratique usuelle. Au fur et à mesure que les programmes sont devenus plus complexes, cela est devenu impossible de les déboguer car les données pouvaient être modifiées par toute fonction présente dans le programme. Des années d'expérience ont convaincu la communauté que l'accès aux données doit être le plus local possible.
- **Q** : Les variables globales vont contre l'approche orientée objet. Comment faire pour partager de manière sécurisée l'accès à certaines variables entre plusieurs instances de la même classe ?

# Réflexion : variables globales, paramètres

- **Q** : envoyer fréquemment plusieurs variables d'intérêt en paramètre est encombrant. Comment faire mieux en C++ ?
- **R** : regrouper les fonctions qui travaillent avec les mêmes variables en tant que méthodes au sein d'une classe avec une fonctionnalité bien définie.
- **R** : les variables communes seront déclarées comme membres privés de classe
- **Q** : pour quoi ne pas déclarer toutes les variables comme des variables globales ?
- **R** : il y a un certain temps, c'était justement la pratique usuelle. Au fur et à mesure que les programmes sont devenus plus complexes, cela est devenu impossible de les déboguer car les données pouvaient être modifiées par toute fonction présente dans le programme. Des années d'expérience ont convaincu la communauté que l'accès aux données doit être le plus local possible.
- **Q** : Les variables globales vont contre l'approche orientée objet. Comment faire pour partager de manière sécurisée l'accès à certaines variables entre plusieurs instances de la même classe ?
- **R** : Membre statique de classe

# Accessibilité : espace de noms

- en C++ on peut cloisonner les variables et les fonctions globales à l'aide de l'espace de nom (namespace)
- déclarer dans un espace de noms

```
namespace Espace1
{
  // declarations de type (et de classes)
  // declarations de variables
  // declarations et definitions de fonctions
}
```

- y faire référence

```
Espace1::variable
Espace1::fonction()
```

méthode 1

```
using namespace Espace1;
variable
```

méthode 2

# Exemple : espace de noms

```
namespace Anglais
{
  char * color []={"White"," Yellow "," Red ","Blue"};
  void Couleurs ( int i)
  {
    printf (" Color number %d is %s \ n", i,color [i]);
  }
}

namespace Francais
{
  char * color []={"Blanc","Jaune","Rouge","Bleu"};
  void Couleurs ( int i)
  {
    printf ("Couleur numero %d est %s \ n", i,color [i]);
  }
}
```

```
...
EspaceCouleurs::Francais::Couleurs(2);

...
```

accès indirect

```
using namespace EspaceCouleurs ;
using namespace Francais;
...
Couleurs(2);
```

accès direct

# Données : durée de vie

- entre la **création** (allocation de la mémoire) et la **destruction** (la libération de la mémoire)

# Données : durée de vie

- entre la **création** (allocation de la mémoire) et la **destruction** (la libération de la mémoire)
- **globales** et/ou **statiques** : toute la durée d'exécution du programme

# Données : durée de vie

- entre la **création** (allocation de la mémoire) et la **destruction** (la libération de la mémoire)
- **globales** et/ou **statiques** : toute la durée d'exécution du programme
- **locales** : de la déclaration jusqu'à la fin du bloc (sauf **statiques**)

# Données : durée de vie

- entre la **création** (allocation de la mémoire) et la **destruction** (la libération de la mémoire)
- **globales** et/ou **statiques** : toute la durée d'exécution du programme
- **locales** : de la déclaration jusqu'à la fin du bloc (sauf **statiques**)
- **temporaires** (sans nom) : jusqu'à la fin de l'instruction courante

# Données : durée de vie

- entre la **création** (allocation de la mémoire) et la **destruction** (la libération de la mémoire)
- **globales** et/ou **statiques** : toute la durée d'exécution du programme
- **locales** : de la déclaration jusqu'à la fin du bloc (sauf **statiques**)
- **temporaires** (sans nom) : jusqu'à la fin de l'instruction courante
- **paramètres d'appel et de retour** : pendant la durée de l'appel de la fonction

# Données : durée de vie

- entre la **création** (allocation de la mémoire) et la **destruction** (la libération de la mémoire)
- **globales** et/ou **statiques** : toute la durée d'exécution du programme
- **locales** : de la déclaration jusqu'à la fin du bloc (sauf **statiques**)
- **temporaires** (sans nom) : jusqu'à la fin de l'instruction courante
- **paramètres d'appel et de retour** : pendant la durée de l'appel de la fonction
- **dynamiques** : entre création par **new** et destruction par **delete**

# Types scalaires disponibles

- Types scalaires **numériques** natifs
  - **taille** en mémoire : 1, 2, 4, 8, 10, 16 octets
  - présence d'un **signe** : signé / non-signé
  - **granularité** : booléen, entier, réel

# Types scalaires disponibles

- Types scalaires **numériques** natifs
  - **taille** en mémoire : 1, 2, 4, 8, 10, 16 octets
  - présence d'un **signe** : signé / non-signé
  - **granularité** : booléen, entier, réel
- types scalaires d'**adressage** (permettent d'accéder à une variable en mémorisant son adresse)
  - **pointeur** : une adresse, un type et une taille
  - **référence** : une adresse, un type, une variable à "cloner" et une taille

# Références : déclaration

- La référence est un alias pour la variable à laquelle elle est associée

# Références : déclaration

- La référence est un alias pour la variable à laquelle elle est associée
- On ne peut jamais déclarer une référence sans l'associer à une variable (par l'attribution à la déclaration)

```
type_t & reference = < var_ou_ref_de_type_t > ;
```

# Références : déclaration

- La référence est un alias pour la variable à laquelle elle est associée
- On ne peut jamais déclarer une référence sans l'associer à une variable (par l'attribution à la déclaration)

```
type_t & reference = < var_ou_ref_de_type_t > ;
```

- Exemple :

```
int i=5, j=10;
int & ri=i; int &ri2 = ri;
cout << i   << " " << j    << " "
     << ri  << " " << ri2  << endl;
cout << &i  << " " << &j   << " "
     << &ri << " " << &ri2 << endl;
return 0;
```

# Références : déclaration

- La référence est un alias pour la variable à laquelle elle est associée
- On ne peut jamais déclarer une référence sans l'associer à une variable (par l'attribution à la déclaration)

```
type_t & reference = < var_ou_ref_de_type_t > ;
```

- Exemple :

```
int i=5, j=10;
int & ri=i; int &ri2 = ri;
cout << i   << " " << j    << " "
     << ri  << " " << ri2  << endl;
cout << &i  << " " << &j   << " "
     << &ri << " " << &ri2 << endl;
return 0;
```

- On peut utiliser partout une référence à la place de sa variable associée (le type est le même à l'évaluation !)

# Références : déclaration

- La référence est un alias pour la variable à laquelle elle est associée
- On ne peut jamais déclarer une référence sans l'associer à une variable (par l'attribution à la déclaration)

```
type_t & reference = < var_ou_ref_de_type_t > ;
```

- Exemple :

```
int i=5, j=10;
int & ri=i; int &ri2 = ri;
cout << i   << " " << j    << " "
     << ri  << " " << ri2  << endl;
cout << &i  << " " << &j   << " "
     << &ri << " " << &ri2 << endl;
return 0;
```

- On peut utiliser partout une référence à la place de sa variable associée (le type est le même à l'évaluation !)
- La référence cache en réalité un pointeur vers la variable associée. La référence d'une variable var vaut toujours *(&var)

# Références : déclaration

- La référence est un alias pour la variable à laquelle elle est associée
- On ne peut jamais déclarer une référence sans l'associer à une variable (par l'attribution à la déclaration)

```
type_t & reference = < var_ou_ref_de_type_t > ;
```

- Exemple :

```
int i=5, j=10;
int & ri=i; int &ri2 = ri;
cout << i   << " " << j    << " "
     << ri  << " " << ri2  << endl;
cout << &i  << " " << &j   << " "
     << &ri << " " << &ri2 << endl;
return 0;
```

- On peut utiliser partout une référence à la place de sa variable associée (le type est le même à l'évaluation !)
- La référence cache en réalité un pointeur vers la variable associée. La référence d'une variable var vaut toujours *(&var)
- Q : peut-on avoir l'adresse d'une référence ?

# Références : déclaration

- La référence est un alias pour la variable à laquelle elle est associée
- On ne peut jamais déclarer une référence sans l'associer à une variable (par l'attribution à la déclaration)

```
type_t & reference = < var_ou_ref_de_type_t > ;
```

- Exemple :

```
int i=5, j=10;
int & ri=i; int &ri2 = ri;
cout << i   << " " << j    << " "
     << ri  << " " << ri2  << endl;
cout << &i  << " " << &j   << " "
     << &ri << " " << &ri2 << endl;
return 0;
```

- On peut utiliser partout une référence à la place de sa variable associée (le type est le même à l'évaluation !)
- La référence cache en réalité un pointeur vers la variable associée. La référence d'une variable var vaut toujours *(&var)
- Q : peut-on avoir l'adresse d'une référence ?
- R : non. La référence en elle même n'est pas un objet, la référence est son référent.

# Références : déclaration

- La référence est un alias pour la variable à laquelle elle est associée
- On ne peut jamais déclarer une référence sans l'associer à une variable (par l'attribution à la déclaration)

```
type_t & reference = < var_ou_ref_de_type_t > ;
```

- Exemple :

```
int i=5, j=10;
int & ri=i; int &ri2 = ri;
cout << i   << " " << j    << " "
     << ri  << " " << ri2  << endl;
cout << &i  << " " << &j   << " "
     << &ri << " " << &ri2 << endl;
return 0;
```

- On peut utiliser partout une référence à la place de sa variable associée (le type est le même à l'évaluation !)
- La référence cache en réalité un pointeur vers la variable associée. La référence d'une variable var vaut toujours *(&var)
- Q : peut-on avoir l'adresse d'une référence ?
- R : non. La référence en elle même n'est pas un objet, la référence est son référent.
- Q : peut-on modifier la variable associée ?

# Références : déclaration

- La référence est un alias pour la variable à laquelle elle est associée
- On ne peut jamais déclarer une référence sans l'associer à une variable (par l'attribution à la déclaration)

```
type_t & reference = < var_ou_ref_de_type_t > ;
```

- Exemple :

```
int i=5, j=10;
int & ri=i; int &ri2 = ri;
cout << i   << " " << j    << " "
     << ri  << " " << ri2  << endl;
cout << &i  << " " << &j   << " "
     << &ri << " " << &ri2 << endl;
return 0;
```

- On peut utiliser partout une référence à la place de sa variable associée (le type est le même à l'évaluation !)
- La référence cache en réalité un pointeur vers la variable associée. La référence d'une variable var vaut toujours *(&var)
- Q : peut-on avoir l'adresse d'une référence ?
- R : non. La référence en elle même n'est pas un objet, la référence est son référent.
- Q : peut-on modifier la variable associée ?
- R : non. Cela est spécifié dans le standard ISO.

# Références vs. pointeurs : à retenir

## Pour quoi les références sont utiles :

- La référence pointe toujours vers un objet, alors que l'utilisation d'un pointeur nécessite une vérification (pas NULL)

# Références vs. pointeurs : à retenir

## Pour quoi les références sont utiles :

- La référence pointe toujours vers un objet, alors que l'utilisation d'un pointeur nécessite une vérification (pas NULL)
- ... mais les horreurs sont toujours possibles en C++ :

```
int& getLocalVariable()
{
    int x;
    return x;
}
```

# Références vs. pointeurs : à retenir

## Pour quoi les références sont utiles :

- La référence pointe toujours vers un objet, alors que l'utilisation d'un pointeur nécessite une vérification (pas NULL)
- ... mais les horreurs sont toujours possibles en C++ :

```
int& getLocalVariable()
{
    int x;
    return x;
}
```

- Sans faire rien de spécial, on évite de faire des copies au passage d'objets en paramètres de fonction

# Références vs. pointeurs : à retenir

## Pour quoi les références sont utiles :

- La référence pointe toujours vers un objet, alors que l'utilisation d'un pointeur nécessite une vérification (pas NULL)
- ... mais les horreurs sont toujours possibles en C++ :

```
int& getLocalVariable()
{
    int x;
    return x;
}
```

- Sans faire rien de spécial, on évite de faire des copies au passage d'objets en paramètres de fonction
- ... et on peut décider si la fonction a le droit de les modifier ou pas :

```
int workWithClass(
  MyClass& a_class_object )
{
  ...
}
```

OK pour modifier l'objet

```
int workWithClass(
   const MyClass& a_class_object )
 {
   ...
 }
```

interdiction de modifier

# Références vs. pointeurs : à retenir

## Pour quoi les références sont utiles :

- La référence pointe toujours vers un objet, alors que l'utilisation d'un pointeur nécessite une vérification (pas NULL)
- ... mais les horreurs sont toujours possibles en C++ :

```
int& getLocalVariable()
{
    int x;
    return x;
}
```

- Sans faire rien de spécial, on évite de faire des copies au passage d'objets en paramètres de fonction
- ... et on peut décider si la fonction a le droit de les modifier ou pas :

```
int workWithClass(
  MyClass& a_class_object )
{
  ...
}
```

OK pour modifier l'objet

```
int workWithClass(
   const MyClass& a_class_object )
 {
   ...
 }
```

interdiction de modifier

- Attention : références vers objets à allocation dynamique (ambiguïtés)

# Références vs. pointeurs : à retenir

## Pour quoi les références sont utiles :

- La référence pointe toujours vers un objet, alors que l'utilisation d'un pointeur nécessite une vérification (pas NULL)
- ... mais les horreurs sont toujours possibles en C++ :

```
int& getLocalVariable()
{
    int x;
    return x;
}
```

- Sans faire rien de spécial, on évite de faire des copies au passage d'objets en paramètres de fonction
- ... et on peut décider si la fonction a le droit de les modifier ou pas :

```
int workWithClass(
  MyClass& a_class_object )
{
  ...
}
```

OK pour modifier l'objet

```
int workWithClass(
   const MyClass& a_class_object )
 {
   ...
 }
```

interdiction de modifier

- Attention : références vers objets à allocation dynamique (ambiguïtés)
- Le standard ISO n'impose pas une

# Exemple simple : références

```
float add1(float x1,float x2){
  return x1+x2;
}

void add2(float x1,float x2, float* res){
  *res=x1+x2;
}

void add3(float x1,float x2, float& res){
  res=x1+x2;
}

int main()
{
  float x1=0.5, x2=1, y;
  y = add1(x1,x2); // version retour de fonction
  add2(x1,x2,&y);  // version pointeur
  add3(x1,x2,y);   // version reference
  return 0;
}
```

# Types disponibles par agrégation

- types **homogènes** (vecteurs d'éléments ou tableaux), une matrice 2D étant un vecteur de vecteurs. Caractéristiques :
  - type de l'élément
  - nombre d'éléments (spécifié à la création !)
  - adresse du premier élément

# Types disponibles par agrégation

- types **homogènes** (vecteurs d'éléments ou tableaux), une matrice 2D étant un vecteur de vecteurs. Caractéristiques :
  - type de l'élément
  - nombre d'éléments (spécifié à la création !)
  - adresse du premier élément
- types **hybrides** (structures, unions, classes). Caractéristiques :
  - liste de **champs** avec droit d'accès, type et nom
  - pour classes : liste et nom des **méthodes**

# Les tableaux à taille fixe

- Déclaration possible comme variable locale, globale ou statique. Pourquoi une taille fixe ?

# Les tableaux à taille fixe

- Déclaration possible comme variable locale, globale ou statique. Pourquoi une taille fixe ?
- En tant que paramètre de fonction il n'est pas copié dans la pile ! (règle historique C : les tableaux sont interprétés comme des pointeurs s'ils sont passés en paramètre)

# Les tableaux à taille fixe

- Déclaration possible comme variable locale, globale ou statique. Pourquoi une taille fixe ?
- En tant que paramètre de fonction il n'est pas copié dans la pile ! (règle historique C : les tableaux sont interprétés comme des pointeurs s'ils sont passés en paramètre)
- Pour s'amuser, le passage par valeur est possible de manière indirecte :

```
struct A { int arr[2]; };
void func(A a){ // copie profonde
  ...
}
```

# Les tableaux à taille fixe

- Déclaration possible comme variable locale, globale ou statique. Pourquoi une taille fixe ?
- En tant que paramètre de fonction il n'est pas copié dans la pile ! (règle historique C : les tableaux sont interprétés comme des pointeurs s'ils sont passés en paramètre)
- Pour s'amuser, le passage par valeur est possible de manière indirecte :

```
struct A { int arr[2]; };
void func(A a){ // copie profonde
  ...
}
```

- L'opérateur sizeof donne sa vraie taille

# Les tableaux à taille fixe

- Déclaration possible comme variable locale, globale ou statique. Pourquoi une taille fixe ?
- En tant que paramètre de fonction il n'est pas copié dans la pile ! (règle historique C : les tableaux sont interprétés comme des pointeurs s'ils sont passés en paramètre)
- Pour s'amuser, le passage par valeur est possible de manière indirecte :

```
struct A { int arr[2]; };
void func(A a){ // copie profonde
  ...
}
```

- L'opérateur sizeof donne sa vraie taille
- Il n'y a pas de tableaux 2-D ou N-D

# Les tableaux à taille fixe

- Déclaration possible comme variable locale, globale ou statique. Pourquoi une taille fixe ?
- En tant que paramètre de fonction il n'est pas copié dans la pile ! (règle historique C : les tableaux sont interprétés comme des pointeurs s'ils sont passés en paramètre)
- Pour s'amuser, le passage par valeur est possible de manière indirecte :

```
struct A { int arr[2]; };
void func(A a){ // copie profonde
  ...
}
```

- L'opérateur sizeof donne sa vraie taille
- Il n'y a pas de tableaux 2-D ou N-D
- Considérons :

```
int tab[3][5];
```

# Les tableaux à taille fixe

- Déclaration possible comme variable locale, globale ou statique. Pourquoi une taille fixe ?
- En tant que paramètre de fonction il n'est pas copié dans la pile ! (règle historique C : les tableaux sont interprétés comme des pointeurs s'ils sont passés en paramètre)
- Pour s'amuser, le passage par valeur est possible de manière indirecte :

```
struct A { int arr[2]; };
void func(A a){ // copie profonde
  ...
}
```

- L'opérateur sizeof donne sa vraie taille
- Il n'y a pas de tableaux 2-D ou N-D
- Considérons :

```
int tab[3][5];
```

- Rappel : une l-value est une expression qui désigne la localisation d'un objet en mémoire qui peut être examinée et réécrite ; une r-value est une expression qui a une valeur

# Les tableaux à taille fixe

- Déclaration possible comme variable locale, globale ou statique. Pourquoi une taille fixe ?
- En tant que paramètre de fonction il n'est pas copié dans la pile ! (règle historique C : les tableaux sont interprétés comme des pointeurs s'ils sont passés en paramètre)
- Pour s'amuser, le passage par valeur est possible de manière indirecte :

```
struct A { int arr[2]; };
void func(A a){ // copie profonde
  ...
}
```

- L'opérateur sizeof donne sa vraie taille
- Il n'y a pas de tableaux 2-D ou N-D
- Considérons :

```
int tab[3][5];
```

- Rappel : une l-value est une expression qui désigne la localisation d'un objet en mémoire qui peut être examinée et réécrite ; une r-value est une expression qui a une valeur
- Q : quel est son emplacement et structure ?

# Les tableaux à taille fixe

- Déclaration possible comme variable locale, globale ou statique. Pourquoi une taille fixe ?
- En tant que paramètre de fonction il n'est pas copié dans la pile ! (règle historique C : les tableaux sont interprétés comme des pointeurs s'ils sont passés en paramètre)
- Pour s'amuser, le passage par valeur est possible de manière indirecte :

```
struct A { int arr[2]; };
void func(A a){ // copie profonde
  ...
}
```

- L'opérateur sizeof donne sa vraie taille
- Il n'y a pas de tableaux 2-D ou N-D
- Considérons :

```
int tab[3][5];
```

- Rappel : une l-value est une expression qui désigne la localisation d'un objet en mémoire qui peut être examinée et réécrite ; une r-value est une expression qui a une valeur
- Q : quel est son emplacement et structure ?
- Q : type de tab ? l-value/r-value ?

# Les tableaux à taille fixe

- Déclaration possible comme variable locale, globale ou statique. Pourquoi une taille fixe ?
- En tant que paramètre de fonction il n'est pas copié dans la pile ! (règle historique C : les tableaux sont interprétés comme des pointeurs s'ils sont passés en paramètre)
- Pour s'amuser, le passage par valeur est possible de manière indirecte :

```
struct A { int arr[2]; };
void func(A a){ // copie profonde
  ...
}
```

- L'opérateur sizeof donne sa vraie taille
- Il n'y a pas de tableaux 2-D ou N-D
- Considérons :

```
int tab[3][5];
```

- Rappel : une l-value est une expression qui désigne la localisation d'un objet en mémoire qui peut être examinée et réécrite ; une r-value est une expression qui a une valeur
- Q : quel est son emplacement et structure ?
- Q : type de tab ? l-value/r-value ?
- R : int[3][5] ; l-value non-modifiable (comme les types incomplets et les types const)

# Les tableaux à taille fixe

- Déclaration possible comme variable locale, globale ou statique. Pourquoi une taille fixe ?
- En tant que paramètre de fonction il n'est pas copié dans la pile ! (règle historique C : les tableaux sont interprétés comme des pointeurs s'ils sont passés en paramètre)
- Pour s'amuser, le passage par valeur est possible de manière indirecte :

```
struct A { int arr[2]; };
void func(A a){ // copie profonde
  ...
}
```

- L'opérateur sizeof donne sa vraie taille
- Il n'y a pas de tableaux 2-D ou N-D
- Considérons :

```
int tab[3][5];
```

- Rappel : une l-value est une expression qui désigne la localisation d'un objet en mémoire qui peut être examinée et réécrite ; une r-value est une expression qui a une valeur
- Q : quel est son emplacement et structure ?
- Q : type de tab ? l-value/r-value ?
- R : int[3][5] ; l-value non-modifiable (comme les types incomplets et les types const)
- Q : type de tab[1] ? l-value/r-value ?

# Les tableaux à taille fixe

- Déclaration possible comme variable locale, globale ou statique. Pourquoi une taille fixe ?
- En tant que paramètre de fonction il n'est pas copié dans la pile ! (règle historique C : les tableaux sont interprétés comme des pointeurs s'ils sont passés en paramètre)
- Pour s'amuser, le passage par valeur est possible de manière indirecte :

```
struct A { int arr[2]; };
void func(A a){ // copie profonde
  ...
}
```

- L'opérateur sizeof donne sa vraie taille
- Il n'y a pas de tableaux 2-D ou N-D
- Considérons :

```
int tab[3][5];
```

- Rappel : une l-value est une expression qui désigne la localisation d'un objet en mémoire qui peut être examinée et réécrite ; une r-value est une expression qui a une valeur
- Q : quel est son emplacement et structure ?
- Q : type de tab ? l-value/r-value ?
- R : int[3][5] ; l-value non-modifiable (comme les types incomplets et les types const)
- Q : type de tab[1] ? l-value/r-value ?
- R : int[5] ; l-value non-modifiable

# Les tableaux à taille fixe

- Déclaration possible comme variable locale, globale ou statique. Pourquoi une taille fixe ?
- En tant que paramètre de fonction il n'est pas copié dans la pile ! (règle historique C : les tableaux sont interprétés comme des pointeurs s'ils sont passés en paramètre)
- Pour s'amuser, le passage par valeur est possible de manière indirecte :

```
struct A { int arr[2]; };
void func(A a){ // copie profonde
  ...
}
```

- L'opérateur sizeof donne sa vraie taille
- Il n'y a pas de tableaux 2-D ou N-D
- Considérons :

```
int tab[3][5];
```

- Rappel : une l-value est une expression qui désigne la localisation d'un objet en mémoire qui peut être examinée et réécrite ; une r-value est une expression qui a une valeur
- Q : quel est son emplacement et structure ?
- Q : type de tab ? l-value/r-value ?
- R : int[3][5] ; l-value non-modifiable (comme les types incomplets et les types const)
- Q : type de tab[1] ? l-value/r-value ?
- R : int[5] ; l-value non-modifiable
- Q : type de (tab+1) ? l-value/r-value ?

# Les tableaux à taille fixe

- Déclaration possible comme variable locale, globale ou statique. Pourquoi une taille fixe ?
- En tant que paramètre de fonction il n'est pas copié dans la pile ! (règle historique C : les tableaux sont interprétés comme des pointeurs s'ils sont passés en paramètre)
- Pour s'amuser, le passage par valeur est possible de manière indirecte :

```
struct A { int arr[2]; };
void func(A a){ // copie profonde
  ...
}
```

- L'opérateur sizeof donne sa vraie taille
- Il n'y a pas de tableaux 2-D ou N-D
- Considérons :

```
int tab[3][5];
```

- Rappel : une l-value est une expression qui désigne la localisation d'un objet en mémoire qui peut être examinée et réécrite ; une r-value est une expression qui a une valeur
- Q : quel est son emplacement et structure ?
- Q : type de tab ? l-value/r-value ?
- R : int[3][5] ; l-value non-modifiable (comme les types incomplets et les types const)
- Q : type de tab[1] ? l-value/r-value ?
- R : int[5] ; l-value non-modifiable
- Q : type de (tab+1) ? l-value/r-value ?
- R : int(*)[5] ; r-value

# Les tableaux à taille fixe

- Déclaration possible comme variable locale, globale ou statique. Pourquoi une taille fixe ?
- En tant que paramètre de fonction il n'est pas copié dans la pile ! (règle historique C : les tableaux sont interprétés comme des pointeurs s'ils sont passés en paramètre)
- Pour s'amuser, le passage par valeur est possible de manière indirecte :

```
struct A { int arr[2]; };
void func(A a){ // copie profonde
  ...
}
```

- L'opérateur sizeof donne sa vraie taille
- Il n'y a pas de tableaux 2-D ou N-D
- Considérons :

```
int tab[3][5];
```

- Rappel : une l-value est une expression qui désigne la localisation d'un objet en mémoire qui peut être examinée et réécrite ; une r-value est une expression qui a une valeur
- Q : quel est son emplacement et structure ?
- Q : type de tab ? l-value/r-value ?
- R : int[3][5] ; l-value non-modifiable (comme les types incomplets et les types const)
- Q : type de tab[1] ? l-value/r-value ?
- R : int[5] ; l-value non-modifiable
- Q : type de (tab+1) ? l-value/r-value ?
- R : int(*)[5] ; r-value
- Q : type de tab[0][3] ? l-value/r-value ?

# Les tableaux à taille fixe

- Déclaration possible comme variable locale, globale ou statique. Pourquoi une taille fixe ?
- En tant que paramètre de fonction il n'est pas copié dans la pile ! (règle historique C : les tableaux sont interprétés comme des pointeurs s'ils sont passés en paramètre)
- Pour s'amuser, le passage par valeur est possible de manière indirecte :

```
struct A { int arr[2]; };
void func(A a){ // copie profonde
  ...
}
```

- L'opérateur sizeof donne sa vraie taille
- Il n'y a pas de tableaux 2-D ou N-D
- Considérons :

```
int tab[3][5];
```

- Rappel : une l-value est une expression qui désigne la localisation d'un objet en mémoire qui peut être examinée et réécrite ; une r-value est une expression qui a une valeur
- Q : quel est son emplacement et structure ?
- Q : type de tab ? l-value/r-value ?
- R : int[3][5] ; l-value non-modifiable (comme les types incomplets et les types const)
- Q : type de tab[1] ? l-value/r-value ?
- R : int[5] ; l-value non-modifiable
- Q : type de (tab+1) ? l-value/r-value ?
- R : int(*)[5] ; r-value
- Q : type de tab[0][3] ? l-value/r-value ?
- R : int ;l-value

# Allocation dynamique

- En C++ on utilise les opérateurs new pour la création et delete pour la destruction

# Allocation dynamique

- En C++ on utilise les opérateurs new pour la création et delete pour la destruction
- L'opérateur new retourne un pointeur vers une variable du type demandé, qui sera allouée dans le tas :

```
type_t * ptr1 =( type_t *) malloc (sizeof ( type_t )); // version C
type_t * ptr2 =new type_t ;                            // version C++
```

# Allocation dynamique

- En C++ on utilise les opérateurs new pour la création et delete pour la destruction
- L'opérateur new retourne un pointeur vers une variable du type demandé, qui sera allouée dans le tas :

```
type_t * ptr1 =( type_t *) malloc (sizeof ( type_t )); // version C
type_t * ptr2 =new type_t ;                             // version C++
```

- L'opérateur new[] retourne un pointeur vers N variables du type demandé ; allocation également dans le tas :

```
type_t * ptr1 =( type_t *) malloc (sizeof ( type_t ) * N); // version C
type_t * ptr2 =new type_t[N] ;                              // version C++
```

# Allocation dynamique

- En C++ on utilise les opérateurs new pour la création et delete pour la destruction
- L'opérateur new retourne un pointeur vers une variable du type demandé, qui sera allouée dans le tas :

```
type_t * ptr1 =( type_t *) malloc (sizeof ( type_t )); // version C
type_t * ptr2 =new type_t ;                            // version C++
```

- L'opérateur new[] retourne un pointeur vers N variables du type demandé ; allocation également dans le tas :

```
type_t * ptr1 =( type_t *) malloc (sizeof ( type_t ) * N); // version C
type_t * ptr2 =new type_t[N] ;                             // version C++
```

- L'opérateur delete libère la place occupée par la variable pointée par le pointeur passé en paramètre :

```
free( ptr1 );  // version C
delete ptr2 ;  // version C++
```

# Allocation dynamique

- En C++ on utilise les opérateurs **new** pour la création et **delete** pour la destruction
- L'opérateur **new** retourne un pointeur vers une variable du type demandé, qui sera allouée dans le **tas** :

```
type_t * ptr1 =( type_t *) malloc (sizeof ( type_t )); // version C
type_t * ptr2 =new type_t ;                             // version C++
```

- L'opérateur **new[]** retourne un pointeur vers N variables du type demandé ; allocation également dans le **tas** :

```
type_t * ptr1 =( type_t *) malloc (sizeof ( type_t ) * N); // version C
type_t * ptr2 =new type_t[N] ;                             // version C++
```

- L'opérateur **delete** libère la place occupée par la variable pointée par le pointeur passé en paramètre :

```
free( ptr1 );  // version C
delete ptr2 ;  // version C++
```

- L'opérateur **delete[]** libère la place occupée par les N variables pointées par le pointeur passé en paramètre :

```
free( ptr1 );  // version C
delete[] ptr2 ;  // version C++
```

# Allocation dynamique

- En C++ on utilise les opérateurs new pour la création et delete pour la destruction
- L'opérateur new retourne un pointeur vers une variable du type demandé, qui sera allouée dans le tas :

```
type_t * ptr1 =( type_t *) malloc (sizeof ( type_t )); // version C
type_t * ptr2 =new type_t ;                            // version C++
```

- L'opérateur new[] retourne un pointeur vers N variables du type demandé ; allocation également dans le tas :

```
type_t * ptr1 =( type_t *) malloc (sizeof ( type_t ) * N); // version C
type_t * ptr2 =new type_t[N] ;                             // version C++
```

- L'opérateur delete libère la place occupée par la variable pointée par le pointeur passé en paramètre :

```
free( ptr1 );  // version C
delete ptr2 ;  // version C++
```

- L'opérateur delete[] libère la place occupée par les N variables pointées par le pointeur passé en paramètre :

```
free( ptr1 );  // version C
delete[] ptr2 ;  // version C++
```

- Le pointeur fourni par new est votre seul lien avec la (les) variable(s) allouée(s) ; ne le modifiez pas car il faut le passer à delete pour libérer la mémoire

# Allocation dynamique

- En C++ on utilise les opérateurs new pour la création et delete pour la destruction
- L'opérateur new retourne un pointeur vers une variable du type demandé, qui sera allouée dans le tas :

```
type_t * ptr1 =( type_t *) malloc (sizeof ( type_t )); // version C
type_t * ptr2 =new type_t ;                             // version C++
```

- L'opérateur new[] retourne un pointeur vers N variables du type demandé ; allocation également dans le tas :

```
type_t * ptr1 =( type_t *) malloc (sizeof ( type_t ) * N); // version C
type_t * ptr2 =new type_t[N] ;                              // version C++
```

- L'opérateur delete libère la place occupée par la variable pointée par le pointeur passé en paramètre :

```
free( ptr1 );  // version C
delete ptr2 ;  // version C++
```

- L'opérateur delete[] libère la place occupée par les N variables pointées par le pointeur passé en paramètre :

```
free( ptr1 );  // version C
delete[] ptr2 ;  // version C++
```

- Le pointeur fourni par new est votre seul lien avec la (les) variable(s) allouée(s) ; ne le modifiez pas car il faut le passer à delete pour libérer la mémoire
- C'est à vous de mémoriser le nombre d'éléments alloués, il n'y a pas d'autre moyen pour retrouver l'information.

# Allocation dynamique

- En C++ on utilise les opérateurs new pour la création et delete pour la destruction
- L'opérateur new retourne un pointeur vers une variable du type demandé, qui sera allouée dans le tas :

```
type_t * ptr1 =( type_t *) malloc (sizeof ( type_t )); // version C
type_t * ptr2 =new type_t ;                            // version C++
```

- L'opérateur new[] retourne un pointeur vers N variables du type demandé ; allocation également dans le tas :

```
type_t * ptr1 =( type_t *) malloc (sizeof ( type_t ) * N); // version C
type_t * ptr2 =new type_t[N] ;                             // version C++
```

- L'opérateur delete libère la place occupée par la variable pointée par le pointeur passé en paramètre :

```
free( ptr1 );  // version C
delete ptr2 ;  // version C++
```

- L'opérateur delete[] libère la place occupée par les N variables pointées par le pointeur passé en paramètre :

```
free( ptr1 );  // version C
delete[] ptr2 ;  // version C++
```

- Le pointeur fourni par new est votre seul lien avec la (les) variable(s) allouée(s) ; ne le modifiez pas car il faut le passer à delete pour libérer la mémoire
- C'est à vous de mémoriser le nombre d'éléments alloués, il n'y a pas d'autre moyen pour retrouver l'information.
- Comment est-ce que delete[] connaît la taille du bloc mémoire à libérer ?

# Allocation dynamique

- En C++ on utilise les opérateurs **new** pour la création et **delete** pour la destruction
- L'opérateur **new** retourne un pointeur vers une variable du type demandé, qui sera allouée dans le **tas** :

```
type_t * ptr1 =( type_t *) malloc (sizeof ( type_t )); // version C
type_t * ptr2 =new type_t ;                             // version C++
```

- L'opérateur **new[]** retourne un pointeur vers N variables du type demandé ; allocation également dans le **tas** :

```
type_t * ptr1 =( type_t *) malloc (sizeof ( type_t ) * N); // version C
type_t * ptr2 =new type_t[N] ;                              // version C++
```

- L'opérateur **delete** libère la place occupée par la variable pointée par le pointeur passé en paramètre :

```
free( ptr1 );  // version C
delete ptr2 ;  // version C++
```

- L'opérateur **delete[]** libère la place occupée par les N variables pointées par le pointeur passé en paramètre :

```
free( ptr1 );  // version C
delete[] ptr2 ;  // version C++
```

- Le pointeur fourni par **new** est votre seul lien avec la (les) variable(s) allouée(s) ; ne le modifiez pas car il faut le passer à **delete** pour libérer la mémoire
- C'est à vous de mémoriser le nombre d'éléments alloués, il n'y a pas d'autre moyen pour retrouver l'information.
- Comment est-ce que **delete[]** connaît la taille du bloc mémoire à libérer ?
- Le pointeur n'est pas modifié après la libération, il garde toujours l'adresse qui maintenant est invalide ! On conseille de le réinitialiser à zéro.

# Allocation dynamique : exemple

- Allocation, utilisation et libération d'un tableau de double :

```
void test1( int taille){
  double * tabd = new double[taille];
  for( int i=0; i<taille; i++)
    tabd [i]=rand()/1000;
  delete [] tabd ;
}
```

# Allocation dynamique : exemple

- Allocation, utilisation et libération d'un tableau de double :

```
void test1( int taille){
  double * tabd = new double[taille];
  for( int i=0; i<taille; i++)
    tabd [i]=rand()/1000;
  delete [] tabd ;
}
```

- Testez la fonction et dessiner l'emplacement de chaque variable.

# Allocation dynamique : exemple

- Allocation, utilisation et libération d'un tableau de double :

```
void test1( int taille){
  double * tabd = new double[taille];
  for( int i=0; i<taille; i++)
    tabd [i]=rand()/1000;
  delete [] tabd ;
}
```

- Testez la fonction et dessiner l'emplacement de chaque variable.
- Implémentez une fonction qui alloue une matrice 2D d'entiers signés, de taille voulue, et une autre qui la libère :
  - il faut passer par un tableau de pointeurs pour les lignes
  - dessinez l'emplacement des variables en mémoire et leurs relations (architecture de données)
  - testez la matrice dans le programme principal, en comparant son utilisation à une matrice 2D locale (initialisation et affichage)

# Fonctions C++

- Chaque fonction en C++ est identifiée uniquement par sa signature : nom et types d'arguments :

```
type_ret Fonct1( type1 p1, type2 p2, type3 p3);
```

# Fonctions C++

- Chaque fonction en C++ est identifiée uniquement par sa signature : nom et types d'arguments :

```
type_ret Fonct1( type1 p1, type2 p2, type3 p3);
```

- La signature ainsi obtenue détermine le nom décoré (mangled) de la fonction

# Fonctions C++

- Chaque fonction en C++ est identifiée uniquement par sa signature : nom et types d'arguments :

```
type_ret Fonct1( type1 p1, type2 p2, type3 p3);
```

- La signature ainsi obtenue détermine le nom décoré (mangled) de la fonction
- On peut avoir des fonctions C++ avec le même nom mais des signatures différentes

# Fonctions C++

- Chaque fonction en C++ est identifiée uniquement par sa signature : nom et types d'arguments :

```
type_ret Fonct1( type1 p1, type2 p2, type3 p3);
```

- La signature ainsi obtenue détermine le nom décoré (mangled) de la fonction
- On peut avoir des fonctions C++ avec le même nom mais des signatures différentes
- On peut avoir des paramètres avec des valeurs par défaut

# Fonctions C++ : les paramètres par défaut

- Les paramètres avec des valeurs par défaut sont toujours les derniers :

```
float Mult ( float p1, float p2, float p3 =1.0f , float p4 =1.0f );
```

# Fonctions C++ : les paramètres par défaut

- Les paramètres avec des valeurs par défaut sont toujours les derniers :

```
float Mult ( float p1, float p2, float p3 =1.0f , float p4 =1.0f );
```

- Il faut déclarer une seule fois les valeurs par défaut :
  - dans le prototype s'il existe
  - sinon dans la définition

# Fonctions C++ : les paramètres par défaut

- Les paramètres avec des valeurs par défaut sont toujours les derniers :

```
float Mult ( float p1, float p2, float p3 =1.0f , float p4 =1.0f );
```

- Il faut déclarer une seule fois les valeurs par défaut :
  - dans le prototype s'il existe
  - sinon dans la définition
- On peut appeler Mult avec 2, 3 ou 4 paramètres, le compilateur rajoute ceux qui manquent.

# Fonctions C++ : les paramètres par défaut

- Les paramètres avec des valeurs par défaut sont toujours les derniers :

```
float Mult ( float p1, float p2, float p3 =1.0f , float p4 =1.0f );
```

- Il faut déclarer une seule fois les valeurs par défaut :
  - dans le prototype s'il existe
  - sinon dans la définition
- On peut appeler Mult avec 2, 3 ou 4 paramètres, le compilateur rajoute ceux qui manquent.
- Écrire la fonction, faites afficher les paramètres et testez-là avec 2, 3 et 4 paramètres

# Fonctions C++ : protection à la modification

- Parfois on utilise des références ou des pointeurs pour réduire le temps d'appel et la mémoire occupée dans la pile.

# Fonctions C++ : protection à la modification

- Parfois on utilise des références ou des pointeurs pour réduire le temps d'appel et la mémoire occupée dans la pile.
- Mais on ne veut pas modifier les valeurs pointées ou référencées. Alors, il faut utiliser le mot **const** :

```
void AfficheTab (double* t1, int taille);          // pas de protection
void AfficheTab (const double* t1, int taille);    // protection

void Add (float & a, float & b, float & res);                // pas de prot.
void Add (const float & a, const float & b, float & res); // protection
```

# Fonctions C++ : protection à la modification

- Parfois on utilise des références ou des pointeurs pour réduire le temps d'appel et la mémoire occupée dans la pile.
- Mais on ne veut pas modifier les valeurs pointées ou référencées. Alors, il faut utiliser le mot **const** :

```
void AfficheTab (double* t1, int taille);          // pas de protection
void AfficheTab (const double* t1, int taille);    // protection

void Add (float & a, float & b, float & res);                // pas de prot.
void Add (const float & a, const float & b, float & res); // protection
```

- Le compilateur vérifie que la variable pointée ou référencée n'est pas modifiée par le code

# Fonctions C++ : protection à la modification

- Parfois on utilise des références ou des pointeurs pour réduire le temps d'appel et la mémoire occupée dans la pile.
- Mais on ne veut pas modifier les valeurs pointées ou référencées. Alors, il faut utiliser le mot **const** :

```
void AfficheTab (double* t1, int taille);          // pas de protection
void AfficheTab (const double* t1, int taille);    // protection

void Add (float & a, float & b, float & res);                // pas de prot.
void Add (const float & a, const float & b, float & res); // protection
```

- Le compilateur vérifie que la variable pointée ou référencée n'est pas modifiée par le code
- Idem pour les références ou pour les pointeurs de retour

# Exemple : protection à la modification

- Concevoir une fonction SearchMin qui recherche l'élément de valeur minimale dans un tableau de float passé en paramètre (avec sa taille) et qui retourne l'adresse de cet élément.

# Exemple : protection à la modification

- Concevoir une fonction SearchMin qui recherche l'élément de valeur minimale dans un tableau de float passé en paramètre (avec sa taille) et qui retourne l'adresse de cet élément.
- Testez-la avec un tableau à 5 éléments. Mémoriser l'adresse retournée et afficher dans la fonction main la valeur minimale trouvée.

# Exemple : protection à la modification

- Concevoir une fonction SearchMin qui recherche l'élément de valeur minimale dans un tableau de float passé en paramètre (avec sa taille) et qui retourne l'adresse de cet élément.
- Testez-la avec un tableau à 5 éléments. Mémoriser l'adresse retournée et afficher dans la fonction main la valeur minimale trouvée.
- Protéger à la modification les éléments du tableau qui arrive en paramètre de SearchMin . Quel est le problème soulevé par le compilateur ?

# Exemple : protection à la modification

- Concevoir une fonction SearchMin qui recherche l'élément de valeur minimale dans un tableau de float passé en paramètre (avec sa taille) et qui retourne l'adresse de cet élément.
- Testez-la avec un tableau à 5 éléments. Mémoriser l'adresse retournée et afficher dans la fonction main la valeur minimale trouvée.
- Protéger à la modification les éléments du tableau qui arrive en paramètre de SearchMin . Quel est le problème soulevé par le compilateur ?
- Comment le résoudre ? Suivre la propagation de la contrainte.

# Le qualificatif volatile

- Si une variable risque d'être modifiée par plusieurs régions de code, qui s'exécutent en parallèle (multi-tâche , multi-thread ou interruptions), alors lui donner le qualificatif volatile oblige le compilateur à désactiver toute optimisation liée aux accès

# Le qualificatif volatile

- Si une variable risque d'être modifiée par plusieurs régions de code, qui s'exécutent en parallèle (multi-tâche , multi-thread ou interruptions), alors lui donner le qualificatif volatile oblige le compilateur à désactiver toute optimisation liée aux accès
- il va lire chaque fois la variable même s'il l'avait déjà fait une instruction avant

# Le qualificatif volatile

- Si une variable risque d'être modifiée par plusieurs régions de code, qui s'exécutent en parallèle (multi-tâche , multi-thread ou interruptions), alors lui donner le qualificatif volatile oblige le compilateur à désactiver toute optimisation liée aux accès
- il va lire chaque fois la variable même s'il l'avait déjà fait une instruction avant
- il va l'écrire aussitôt, sans la garder dans la mémoire cache ou dans des registres

# Le qualificatif volatile

- Si une variable risque d'être modifiée par plusieurs régions de code, qui s'exécutent en parallèle (multi-tâche , multi-thread ou interruptions), alors lui donner le qualificatif volatile oblige le compilateur à désactiver toute optimisation liée aux accès
- il va lire chaque fois la variable même s'il l'avait déjà fait une instruction avant
- il va l'écrire aussitôt, sans la garder dans la mémoire cache ou dans des registres
- attention : cela ne rend pas l'accès atomique

# Le qualificatif volatile

- Si une variable risque d'être modifiée par plusieurs régions de code, qui s'exécutent en parallèle (multi-tâche , multi-thread ou interruptions), alors lui donner le qualificatif volatile oblige le compilateur à désactiver toute optimisation liée aux accès
- il va lire chaque fois la variable même s'il l'avait déjà fait une instruction avant
- il va l'écrire aussitôt, sans la garder dans la mémoire cache ou dans des registres
- attention : cela ne rend pas l'accès atomique
- attention : la vitesse d'accès diminue dramatiquement (10-100 fois)

# Types en C++

- natifs au langage : scalaires (et pointeurs)

# Types en C++

- natifs au langage : scalaires (et pointeurs)
- complexes par agrégation homogène (+ **typedef**)

# Types en C++

- natifs au langage : scalaires (et pointeurs)
- complexes par agrégation homogène (+ **typedef**)
- complexes par agrégation hybride : union/structure/classe + **typedef**

# Types en C++

- natifs au langage : scalaires (et pointeurs)
- complexes par agrégation homogène (+ **typedef**)
- complexes par agrégation hybride : union/structure/classe + **typedef**
- mais le **typedef** n'est pas obligatoire en C++ pour l'agrégation hybride car la syntaxe est plus libre ...

# Types en C++

- natifs au langage : scalaires (et pointeurs)
- complexes par agrégation homogène (+ **typedef**)
- complexes par agrégation hybride : union/structure/classe + **typedef**
- mais le **typedef** n'est pas obligatoire en C++ pour l'agrégation hybride car la syntaxe est plus libre ...
- avec **typedef** on ne déclare que des types :

```
typedef union UnionType1
{
// ...
} UnionType2 ;
typedef struct StructType1
{
// ...
} StructType2 ;
typedef class ClassType1
{
// ...
} ClassType2 ;
```

# Types en C++

- natifs au langage : scalaires (et pointeurs)
- complexes par agrégation homogène (+ **typedef**)
- complexes par agrégation hybride : union/structure/classe + **typedef**
- mais le **typedef** n'est pas obligatoire en C++ pour l'agrégation hybride car la syntaxe est plus libre ...
- avec **typedef** on ne déclare que des types :

```
typedef union UnionType1
{
// ...
} UnionType2 ;
typedef struct StructType1
{
// ...
} StructType2 ;
typedef class ClassType1
{
// ...
} ClassType2 ;
```

- **type** en C : `struct StructType1` ou `StructType2`

# Types en C++

- natifs au langage : scalaires (et pointeurs)
- complexes par agrégation homogène (+ **typedef**)
- complexes par agrégation hybride : union/structure/classe + **typedef**
- mais le **typedef** n'est pas obligatoire en C++ pour l'agrégation hybride car la syntaxe est plus libre ...
- avec **typedef** on ne déclare que des types :

```
typedef union UnionType1
{
// ...
} UnionType2 ;
typedef struct StructType1
{
// ...
} StructType2 ;
typedef class ClassType1
{
// ...
} ClassType2 ;
```

- **type** en C : `struct StructType1` ou `StructType2`
- **type** en C++ : `StructType1` ou `StructType2`

# Types en C++

- sans **typedef**, on déclare des types et des variables

```
union UnionType1
{
// ...
} Union1 , Union2 ;
struct StructType1
{
// ...
} Struct1 , Struct2 ;
class ClassType1
{
// ...
} Class1 , Class2
```

# Types en C++

- sans **typedef**, on déclare des types et des variables

```
union UnionType1
{
// ...
} Union1 , Union2 ;
struct StructType1
{
// ...
} Struct1 , Struct2 ;
class ClassType1
{
// ...
} Class1 , Class2
```

- en C : **variable** `Struct1` de **type** `struct StructType1` ;

# Types en C++

- sans **typedef**, on déclare des types et des variables

```
union UnionType1
{
// ...
} Union1 , Union2 ;
struct StructType1
{
// ...
} Struct1 , Struct2 ;
class ClassType1
{
// ...
} Class1 , Class2
```

- en C : **variable** `Struct1` de **type** `struct StructType1` ;
- en C++ : **variable** `Struct1` de **type** `StructType1` ;

# Types en C++

- sans **typedef**, on déclare des types et des variables

```
union UnionType1
{
// ...
} Union1 , Union2 ;
struct StructType1
{
// ...
} Struct1 , Struct2 ;
class ClassType1
{
// ...
} Class1 , Class2
```

- en C : **variable** `Struct1` de **type** `struct StructType1` ;
- en C++ : **variable** `Struct1` de **type** `StructType1` ;
- ...mais les règles de bonne programmation nous imposent de déclarer séparément les types et les variables :

```
class ClassType1
{
// ...
};
...
ClassType1 Class1 , Class2;
```

# L'approche objet

## La notion de classe

- un type de données parmi d'autres

# L'approche objet

## La notion de classe

- un type de données parmi d'autres
- une instance d'une classe est appelée **objet**

# L'approche objet

## La notion de classe

- un type de données parmi d'autres
- une instance d'une classe est appelée **objet**
- déclaration : syntaxe modifiée d'une structure
  - **droits** : public, protected, private
  - **méthodes** : les fonctions de l'objet

# L'approche objet

## La notion de classe

- un type de données parmi d'autres
- une instance d'une classe est appelée **objet**
- déclaration : syntaxe modifiée d'une structure
  - **droits** : public, protected, private
  - **méthodes** : les fonctions de l'objet
- méthodes spéciales :
  - **constructeurs** : appelés à la création, sans type de retour, entre zéro et plusieurs paramètres : `type_classe (...)`
  - **destructeur** : unique, appelé à la destruction, sans paramètres, sans retour : `type_classe ()`

# Quelle est la vie d'un objet ?

## Création

- allocation de mémoire : `sizeof (type_classe)` octets

# Quelle est la vie d'un objet ?

## Création

- allocation de mémoire : `sizeof (type_classe)` octets
- appel d'un des constructeurs

# Quelle est la vie d'un objet ?

## Création

- allocation de mémoire : `sizeof (type_classe)` octets
- appel d'un des constructeurs

## Utilisation

# Quelle est la vie d'un objet ?

## Création

- allocation de mémoire : `sizeof (type_classe)` octets
- appel d'un des constructeurs

## Utilisation

- lecture/écriture des attributs accessibles

# Quelle est la vie d'un objet ?

## Création

- allocation de mémoire : `sizeof (type_classe)` octets
- appel d'un des constructeurs

## Utilisation

- lecture/écriture des attributs accessibles
- appel des méthodes accessibles

# Quelle est la vie d'un objet ?

## Création

- allocation de mémoire : `sizeof (type_classe)` octets
- appel d'un des constructeurs

## Utilisation

- lecture/écriture des attributs accessibles
- appel des méthodes accessibles

## Destruction

# Quelle est la vie d'un objet ?

## Création

- allocation de mémoire : `sizeof (type_classe)` octets
- appel d'un des constructeurs

## Utilisation

- lecture/écriture des attributs accessibles
- appel des méthodes accessibles

## Destruction

- appel de l'unique destructeur

# Quelle est la vie d'un objet ?

## Création

- allocation de mémoire : `sizeof (type_classe)` octets
- appel d'un des constructeurs

## Utilisation

- lecture/écriture des attributs accessibles
- appel des méthodes accessibles

## Destruction

- appel de l'unique destructeur
- libération de la mémoire : `sizeof (type_classe)` octets

# L'approche objet

- en C++ toute **donnée** est un **objet** et inversement

# L'approche objet

- en C++ toute **donnée** est un **objet** et inversement
- au début de la vie : **construction** $\Rightarrow$ appel d'un constructeur (même un qui ne fait rien)

# L'approche objet

- en C++ toute **donnée** est un **objet** et inversement
- au début de la vie : **construction** $\Rightarrow$ appel d'un constructeur (même un qui ne fait rien)
- tout objet possède deux constructeurs **par défaut** :
  - un constructeur sans paramètres
  - un constructeur de copie

# L'approche objet

- en C++ toute **donnée** est un **objet** et inversement
- au début de la vie : **construction** $\Rightarrow$ appel d'un constructeur (même un qui ne fait rien)
- tout objet possède deux constructeurs **par défaut** :
  - un constructeur sans paramètres
  - un constructeur de copie
- a la fin de la vie : **destruction** $\Rightarrow$ appel d'un destructeur (même un qui ne fait rien)

# L'approche objet

- en C++ toute **donnée** est un **objet** et inversement
- au début de la vie : **construction** $\Rightarrow$ appel d'un constructeur (même un qui ne fait rien)
- tout objet possède deux constructeurs **par défaut** :
  - un constructeur sans paramètres
  - un constructeur de copie
- a la fin de la vie : **destruction** $\Rightarrow$ appel d'un destructeur (même un qui ne fait rien)
- tous les objets (y compris les objets natifs) possèdent un destructeur par défaut qui ne fait rien

# Exemple : le type natif `int`

```
void main()
{
  int i;  // appel de int ()
  int j(12); // int j=12, appel de int ( const int & val)
  int k(j);  // int k=j, appel de int ( const int & val)
  int m=k+j; // int m(k+j), appel de int ( const int & val)
  int* pi1=new int (k*j); // appel de int ( const int & val)
  delete pi1; // appel de ~ int ()
}
// quatre appels de ~ int ()
```

- le constructeur `int()` ne fait rien

# Exemple : le type natif `int`

```
void main()
{
  int i;  // appel de int ()
  int j(12); // int j=12, appel de int ( const int & val)
  int k(j);  // int k=j, appel de int ( const int & val)
  int m=k+j; // int m(k+j), appel de int ( const int & val)
  int* pi1=new int (k*j); // appel de int ( const int & val)
  delete pi1; // appel de ~ int ()
}
// quatre appels de ~ int ()
```

- le constructeur `int()` ne fait rien
- le constructeur `int(const int & val)` copie `val` dans l'objet

# Exemple : le type natif `int`

```
void main()
{
  int i;  // appel de int ()
  int j(12); // int j=12, appel de int ( const int & val)
  int k(j);  // int k=j, appel de int ( const int & val)
  int m=k+j; // int m(k+j), appel de int ( const int & val)
  int* pi1=new int (k*j); // appel de int ( const int & val)
  delete pi1; // appel de ~ int ()
}
// quatre appels de ~ int ()
```

- le constructeur `int()` ne fait rien
- le constructeur `int(const int & val)` copie `val` dans l'objet
- le destructeur `int()` ne fait rien

# Les méthodes (fonctions membres)

Une méthode est une fonction qui travaille pour l'objet (l'instance) pour lequel on l'appelle

- son code a un **accès direct** (par nom) à tous les attributs de l'objet et les méthodes de la classe

# Les méthodes (fonctions membres)

Une méthode est une fonction qui travaille pour l'objet (l'instance) pour lequel on l'appelle

- son code a un **accès direct** (par nom) à tous les attributs de l'objet et les méthodes de la classe
- elle a toujours un premier paramètre caché appelé **this** qui représente l'adresse de l'objet pour lequel on l'appelle - donc si l'on appelle de l'extérieur il faut préciser l'objet

# Les méthodes (fonctions membres)

Une méthode est une fonction qui travaille pour l'objet (l'instance) pour lequel on l'appelle

- son code a un **accès direct** (par nom) à tous les attributs de l'objet et les méthodes de la classe
- elle a toujours un premier paramètre caché appelé **this** qui représente l'adresse de l'objet pour lequel on l'appelle - donc si l'on appelle de l'extérieur il faut préciser l'objet
- appel de l'intérieur d'une autre méthode (accès direct) :

```
NomMethode (parametres)
```

# Les méthodes (fonctions membres)

Une méthode est une fonction qui travaille pour l'objet (l'instance) pour lequel on l'appelle

- son code a un **accès direct** (par nom) à tous les attributs de l'objet et les méthodes de la classe
- elle a toujours un premier paramètre caché appelé **this** qui représente l'adresse de l'objet pour lequel on l'appelle - donc si l'on appelle de l'extérieur il faut préciser l'objet
- appel de l'intérieur d'une autre méthode (accès direct) :

```
NomMethode (parametres)
```

- appel de l'extérieur de la classe (accès indirect) :

```
objet.NomMethode (parametres)  // ou
ptrobjet->NomMethode (parametres) // idem a : (*ptrobjet).
```

# Définitions des méthodes

## Définition inline

- déclaration suivie de définition à l'intérieur de la déclaration de classe

## Définition outline

# Définitions des méthodes

## Définition inline

- déclaration suivie de définition à l'intérieur de la déclaration de classe
- typiquement dans un fichier header

```
class CCercle{
  float rayon;
  //...
  public:
    void SetRay ( float _rayon) {rayon=_rayon;}
  };
```

## Définition outline

# Définitions des méthodes

## Définition inline

- déclaration suivie de définition à l'intérieur de la déclaration de classe
- typiquement dans un fichier header

```
class CCercle{
  float rayon;
  //...
  public:
    void SetRay ( float _rayon) {rayon=_rayon;}
  };
```

## Définition outline

- déclaration à l'intérieur (fichier header), et définition à l'extérieur de la déclaration de classe (fisier .cpp)

```
class CCercle{ //declaration
  float rayon;
  //...
  public:
  void SetRay ( float _rayon);
};
```

fichier header

```
//definition

void CCercle::SetRay (float _rayon)
{
  rayon=_rayon;
}
```

fichier source

# Exemple : CFract

## Modéliser une fraction entière :

```
class CFract{
  //private par defaut
  public:
    int a, b;
    void Afficher(){
      printf ("Fraction : (%d/%d) \ n",a,b);
    }
};
```

CFract a déjà deux constructeurs par défaut et un destructeur par défaut. Sans rien demander, le compilateur génère le code suivant (pas inclus dans les sources) :

```
//constructeur sans parametres
CFract () {}

// constructeur de copie
CFract ( const CFract & f) { a=f.a; b=f.b }

//destructeur par defaut
~CFract () {}

//surcharge de l'operateur =
CFract & operator = ( const CFract & f)
{ a=f.a; b=f.b; return * this ; }
```

Toute déclaration explicite d'un constructeur (sauf celui de copie) désactive la génération automatique du constructeur par défaut sans paramètres.

# Les constructeurs/destructeurs en action

Un rappel des étapes d'utilisation de la pile à l'appel d'une fonction / méthode :

(i) l'adresse de l'instruction suivant l'appel de fonction est placée sur la pile

# Les constructeurs/destructeurs en action

Un rappel des étapes d'utilisation de la pile à l'appel d'une fonction / méthode :

(i) l'adresse de l'instruction suivant l'appel de fonction est placée sur la pile

(ii) on réserve sur la pile de la place pour la valeur de retour de la fonction

# Les constructeurs/destructeurs en action

Un rappel des étapes d'utilisation de la pile à l'appel d'une fonction / méthode :

(i) l'adresse de l'instruction suivant l'appel de fonction est placée sur la pile

(ii) on réserve sur la pile de la place pour la valeur de retour de la fonction

(iii) la position de la pile est notée, et toutes les variables placées par la suite seront considérées locales pour la fonction

# Les constructeurs/destructeurs en action

Un rappel des étapes d'utilisation de la pile à l'appel d'une fonction / méthode :

(i) l'adresse de l'instruction suivant l'appel de fonction est placée sur la pile

(ii) on réserve sur la pile de la place pour la valeur de retour de la fonction

(iii) la position de la pile est notée, et toutes les variables placées par la suite seront considérées locales pour la fonction

(iv) les paramètres de la fonction sont placés sur la pile

# Les constructeurs/destructeurs en action

Un rappel des étapes d'utilisation de la pile à l'appel d'une fonction / méthode :

(i) l'adresse de l'instruction suivant l'appel de fonction est placée sur la pile

(ii) on réserve sur la pile de la place pour la valeur de retour de la fonction

(iii) la position de la pile est notée, et toutes les variables placées par la suite seront considérées locales pour la fonction

(iv) les paramètres de la fonction sont placés sur la pile

(v) la fonction commence à s'exécuter

# Les constructeurs/destructeurs en action

Un rappel des étapes d'utilisation de la pile à l'appel d'une fonction / méthode :

(i) l'adresse de l'instruction suivant l'appel de fonction est placée sur la pile

(ii) on réserve sur la pile de la place pour la valeur de retour de la fonction

(iii) la position de la pile est notée, et toutes les variables placées par la suite seront considérées locales pour la fonction

(iv) les paramètres de la fonction sont placés sur la pile

(v) la fonction commence à s'exécuter

(vi) les variables locales sont placées sur la pile suivant leur déclaration

# Les constructeurs/destructeurs en action

Un rappel des étapes d'utilisation de la pile à l'appel d'une fonction / méthode :

(i) l'adresse de l'instruction suivant l'appel de fonction est placée sur la pile

(ii) on réserve sur la pile de la place pour la valeur de retour de la fonction

(iii) la position de la pile est notée, et toutes les variables placées par la suite seront considérées locales pour la fonction

(iv) les paramètres de la fonction sont placés sur la pile

(v) la fonction commence à s'exécuter

(vi) les variables locales sont placées sur la pile suivant leur déclaration

(vii) la valeur de retour est placée à l'endroit réservé à l'étape (ii)

# Les constructeurs/destructeurs en action

Un rappel des étapes d'utilisation de la pile à l'appel d'une fonction / méthode :

(i) l'adresse de l'instruction suivant l'appel de fonction est placée sur la pile

(ii) on réserve sur la pile de la place pour la valeur de retour de la fonction

(iii) la position de la pile est notée, et toutes les variables placées par la suite seront considérées locales pour la fonction

(iv) les paramètres de la fonction sont placés sur la pile

(v) la fonction commence à s'exécuter

(vi) les variables locales sont placées sur la pile suivant leur déclaration

(vii) la valeur de retour est placée à l'endroit réservé à l'étape (ii)

(viii) la pile est vidée jusqu'à la position notée à l'étape (iii)

# Les constructeurs/destructeurs en action

Un rappel des étapes d'utilisation de la pile à l'appel d'une fonction / méthode :

(i) l'adresse de l'instruction suivant l'appel de fonction est placée sur la pile

(ii) on réserve sur la pile de la place pour la valeur de retour de la fonction

(iii) la position de la pile est notée, et toutes les variables placées par la suite seront considérées locales pour la fonction

(iv) les paramètres de la fonction sont placés sur la pile

(v) la fonction commence à s'exécuter

(vi) les variables locales sont placées sur la pile suivant leur déclaration

(vii) la valeur de retour est placée à l'endroit réservé à l'étape (ii)

(viii) la pile est vidée jusqu'à la position notée à l'étape (iii)

(ix) la valeur de retour est assignée à la variable qui récupère le résultat de la fonction, et est enlevée de la pile

# Les constructeurs/destructeurs en action

Qu'est-ce que le programme suivant affiche ?

```
#include <iostream>
using namespace std;
static int ctr = 0;

class C {
private: int counter;
public:
  C() {counter = ctr;
       std::cout << "Constructor " << counter << endl; ctr++;}
  C(const C&) { counter = ctr;
      std::cout << "Copy " << counter << " was made.\n"; ctr++; }
  ~C() { std::cout << "Destructor " << counter << endl; }

};

C f(C c) {
  std::cout << "Start function\n";
  return c;
}

int main() {
  C obj = f(C());
  std::cout << "Hello World!\n";
  return 0;
}
```

# Les constructeurs/destructeurs en action

Compilation par défaut :

```
C f(C c) {
  std::cout << "Start function\n";
  return c;
}

int main() {
  C obj = f(C());
  std::cout << "Hello World!\n";
  return 0;
}
```

```
Constructor 0
Start function
Copy 1 was made.
Destructor 0
Hello World!
Destructor 1
```

Les compilateurs appliquent une technique appelée RVO (return value optimization), qui représente un des rares cas ou le comportement du programme optimisé peut être différent de celui du programme initial.

# Les constructeurs/destructeurs en action

Compilation avec désactivation explicite de la RVO :

```
C f(C c) {
  std::cout << "Start function\n";
  return c;
}

int main() {
  C obj = f(C());
  std::cout << "Hello World!\n";
  return 0;
}
```

```
Constructor 0
Copy 1 was made.
Start function
Copy 2 was made.
Copy 3 was made.
Destructor 2
Destructor 1
Destructor 0
Hello World!
Destructor 3
```

# Exemple : CFract

Avec les constructeurs / destructeur par défaut :

- Déclarez la classe puis des variables locales `f1` et `f2` de type `CFract` .
- Faire afficher `f1` et `f2` .
- Modifiez le programme, rajoutez une variable `f3` et utilisez le constructeur de copie pour faire que `f3` ait le même contenu que `f1` .
- Même exercice avec une instance dynamique de la classe `CFract` pointée par `pf4` . Faire que cette variable ait le contenu de `f2`.

# Exemple : CFract

Comment initialiser les attributs ?

- De l'extérieur : il faut qu'ils soient accessibles, et c'est contraire à l'approche objet

# Exemple : CFract

Comment initialiser les attributs ?

- De l'extérieur : il faut qu'ils soient accessibles, et c'est contraire à l'approche objet
- De l'intérieur : par une méthode publique Init ou par le constructeur de classe :

```
class CFract
{
  int a, b;
  // ...
  public:
    CFract ( int _a, int _b): a(_a), b(_b) {}
};
```

# Exemple : CFract

Comment initialiser les attributs ?

- De l'extérieur : il faut qu'ils soient accessibles, et c'est contraire à l'approche objet
- De l'intérieur : par une méthode publique Init ou par le constructeur de classe :

```
class CFract
{
  int a, b;
  // ...
  public:
    CFract ( int _a, int _b): a(_a), b(_b) {}
};
```

- Modifier le programme en rajoutant le constructeur explicite. Est-ce que les deux constructeurs par défaut restent encore valables ?

# Exemple : CFract

Comment initialiser les attributs ?

- De l'extérieur : il faut qu'ils soient accessibles, et c'est contraire à l'approche objet
- De l'intérieur : par une méthode publique Init ou par le constructeur de classe :

```
class CFract
{
  int a, b;
  // ...
  public:
    CFract ( int _a, int _b): a(_a), b(_b) {}
};
```

- Modifier le programme en rajoutant le constructeur explicite. Est-ce que les deux constructeurs par défaut restent encore valables ?
- Modifier le programme de test ...

# Exemple : CFract

Comment initialiser les attributs ?

- De l'extérieur : il faut qu'ils soient accessibles, et c'est contraire à l'approche objet
- De l'intérieur : par une méthode publique Init ou par le constructeur de classe :

```
class CFract
{
  int a, b;
  // ...
  public:
    CFract ( int _a, int _b): a(_a), b(_b) {}
};
```

- Modifier le programme en rajoutant le constructeur explicite. Est-ce que les deux constructeurs par défaut restent encore valables ?
- Modifier le programme de test ...
- Rajouter un constructeur copie et un destructeur et affichez des messages d'informations dans chacun.

# Exemple : CFract

Comment initialiser les attributs ?

- De l'extérieur : il faut qu'ils soient accessibles, et c'est contraire à l'approche objet
- De l'intérieur : par une méthode publique Init ou par le constructeur de classe :

```
class CFract
{
  int a, b;
  // ...
  public:
    CFract ( int _a, int _b): a(_a), b(_b) {}
};
```

- Modifier le programme en rajoutant le constructeur explicite. Est-ce que les deux constructeurs par défaut restent encore valables ?
- Modifier le programme de test ...
- Rajouter un constructeur copie et un destructeur et affichez des messages d'informations dans chacun.
- Déclarez des tableaux locaux de 6 `CFract` et dynamiques de 4 `CFract` .

# Exemple : CFract

Comment initialiser les attributs ?

- De l'extérieur : il faut qu'ils soient accessibles, et c'est contraire à l'approche objet
- De l'intérieur : par une méthode publique Init ou par le constructeur de classe :

```
class CFract
{
  int a, b;
  // ...
  public:
    CFract ( int _a, int _b): a(_a), b(_b) {}
};
```

- Modifier le programme en rajoutant le constructeur explicite. Est-ce que les deux constructeurs par défaut restent encore valables ?
- Modifier le programme de test ...
- Rajouter un constructeur copie et un destructeur et affichez des messages d'informations dans chacun.
- Déclarez des tableaux locaux de 6 `CFract` et dynamiques de 4 `CFract` .
- Quelle est l'erreur ? Comment retrouver un constructeur sans paramètres ?

# Exemple : CFract

Comment initialiser les attributs ?

- De l'extérieur : il faut qu'ils soient accessibles, et c'est contraire à l'approche objet
- De l'intérieur : par une méthode publique Init ou par le constructeur de classe :

```
class CFract
{
  int a, b;
  // ...
  public:
    CFract ( int _a, int _b): a(_a), b(_b) {}
};
```

- Modifier le programme en rajoutant le constructeur explicite. Est-ce que les deux constructeurs par défaut restent encore valables ?
- Modifier le programme de test ...
- Rajouter un constructeur copie et un destructeur et affichez des messages d'informations dans chacun.
- Déclarez des tableaux locaux de 6 `CFract` et dynamiques de 4 `CFract` .
- Quelle est l'erreur ? Comment retrouver un constructeur sans paramètres ?
- Surveiller la vie de tous les objets `CFract` (comment ?) .

# Exemple : CFract

Comment initialiser les attributs ?

- De l'extérieur : il faut qu'ils soient accessibles, et c'est contraire à l'approche objet
- De l'intérieur : par une méthode publique Init ou par le constructeur de classe :

```
class CFract
{
  int a, b;
  // ...
  public:
    CFract ( int _a, int _b): a(_a), b(_b) {}
};
```

- Modifier le programme en rajoutant le constructeur explicite. Est-ce que les deux constructeurs par défaut restent encore valables ?
- Modifier le programme de test ...
- Rajouter un constructeur copie et un destructeur et affichez des messages d'informations dans chacun.
- Déclarez des tableaux locaux de 6 `CFract` et dynamiques de 4 `CFract` .
- Quelle est l'erreur ? Comment retrouver un constructeur sans paramètres ?
- Surveiller la vie de tous les objets `CFract` (comment ?) .
- Rajouter une méthode `MultTo` pour multiplier une fraction par une autre.

# Exemple : CFract

Comment initialiser les attributs ?

- De l'extérieur : il faut qu'ils soient accessibles, et c'est contraire à l'approche objet
- De l'intérieur : par une méthode publique Init ou par le constructeur de classe :

```
class CFract
{
  int a, b;
  // ...
  public:
    CFract ( int _a, int _b): a(_a), b(_b) {}
};
```

- Modifier le programme en rajoutant le constructeur explicite. Est-ce que les deux constructeurs par défaut restent encore valables ?
- Modifier le programme de test ...
- Rajouter un constructeur copie et un destructeur et affichez des messages d'informations dans chacun.
- Déclarez des tableaux locaux de 6 `CFract` et dynamiques de 4 `CFract` .
- Quelle est l'erreur ? Comment retrouver un constructeur sans paramètres ?
- Surveiller la vie de tous les objets `CFract` (comment ?) .
- Rajouter une méthode `MultTo` pour multiplier une fraction par une autre.
- Rajouter une méthode `AddTo` pour additionner une fraction à une autre.

# Exemple : CFract

Comment initialiser les attributs ?

- De l'extérieur : il faut qu'ils soient accessibles, et c'est contraire à l'approche objet
- De l'intérieur : par une méthode publique Init ou par le constructeur de classe :

```
class CFract
{
  int a, b;
  // ...
  public:
    CFract ( int _a, int _b): a(_a), b(_b) {}
};
```

- Modifier le programme en rajoutant le constructeur explicite. Est-ce que les deux constructeurs par défaut restent encore valables ?
- Modifier le programme de test ...
- Rajouter un constructeur copie et un destructeur et affichez des messages d'informations dans chacun.
- Déclarez des tableaux locaux de 6 `CFract` et dynamiques de 4 `CFract` .
- Quelle est l'erreur ? Comment retrouver un constructeur sans paramètres ?
- Surveiller la vie de tous les objets `CFract` (comment ?) .
- Rajouter une méthode `MultTo` pour multiplier une fraction par une autre.
- Rajouter une méthode `AddTo` pour additionner une fraction à une autre.
- Rajouter une méthode `Norm` qui normalise la valeur de la fraction. Utilisez-la dans les endroits nécessaires.

# Exemple : CFract

Comment initialiser les attributs ?

- De l'extérieur : il faut qu'ils soient accessibles, et c'est contraire à l'approche objet
- De l'intérieur : par une méthode publique Init ou par le constructeur de classe :

```
class CFract
{
  int a, b;
  // ...
  public:
    CFract ( int _a, int _b): a(_a), b(_b) {}
};
```

- Modifier le programme en rajoutant le constructeur explicite. Est-ce que les deux constructeurs par défaut restent encore valables ?
- Modifier le programme de test ...
- Rajouter un constructeur copie et un destructeur et affichez des messages d'informations dans chacun.
- Déclarez des tableaux locaux de 6 `CFract` et dynamiques de 4 `CFract` .
- Quelle est l'erreur ? Comment retrouver un constructeur sans paramètres ?
- Surveiller la vie de tous les objets `CFract` (comment ?) .
- Rajouter une méthode `MultTo` pour multiplier une fraction par une autre.
- Rajouter une méthode `AddTo` pour additionner une fraction à une autre.
- Rajouter une méthode `Norm` qui normalise la valeur de la fraction. Utilisez-la dans les endroits nécessaires.
- Rendre explicites les différentes protections à la modification en utilisant `const`

# Exemple : CFract

Comment initialiser les attributs ?

- De l'extérieur : il faut qu'ils soient accessibles, et c'est contraire à l'approche objet
- De l'intérieur : par une méthode publique Init ou par le constructeur de classe :

```
class CFract
{
  int a, b;
  // ...
  public:
    CFract ( int _a, int _b): a(_a), b(_b) {}
};
```

- Modifier le programme en rajoutant le constructeur explicite. Est-ce que les deux constructeurs par défaut restent encore valables ?
- Modifier le programme de test ...
- Rajouter un constructeur copie et un destructeur et affichez des messages d'informations dans chacun.
- Déclarez des tableaux locaux de 6 `CFract` et dynamiques de 4 `CFract` .
- Quelle est l'erreur ? Comment retrouver un constructeur sans paramètres ?
- Surveiller la vie de tous les objets `CFract` (comment ?) .
- Rajouter une méthode `MultTo` pour multiplier une fraction par une autre.
- Rajouter une méthode `AddTo` pour additionner une fraction à une autre.
- Rajouter une méthode `Norm` qui normalise la valeur de la fraction. Utilisez-la dans les endroits nécessaires.
- Rendre explicites les différentes protections à la modification en utilisant `const`
- Réfléchir sur l'accessibilité à donner aux méthodes de `CFract`.

# Encapsulation

- Notion essentielle dans la programmation objet

# Encapsulation

- Notion essentielle dans la programmation objet
- Suppose une séparation entre l'interface publique (accessible) d'une classe et son fonctionnement interne qui ne doit pas être ni perturbé ni forcement connu par l'extérieur.

# Encapsulation

- Notion essentielle dans la programmation objet
- Suppose une séparation entre l'interface publique (accessible) d'une classe et son fonctionnement interne qui ne doit pas être ni perturbé ni forcement connu par l'extérieur.
- Outils pour la mise en ouvre de l'encapsulation

# Encapsulation

- Notion essentielle dans la programmation objet
- Suppose une séparation entre l'interface publique (accessible) d'une classe et son fonctionnement interne qui ne doit pas être ni perturbé ni forcement connu par l'extérieur.
- Outils pour la mise en ouvre de l'encapsulation
    - d'une architecture pensée et validée par le concepteur

# Encapsulation

- Notion essentielle dans la programmation objet
- Suppose une séparation entre l'interface publique (accessible) d'une classe et son fonctionnement interne qui ne doit pas être ni perturbé ni forcement connu par l'extérieur.
- Outils pour la mise en ouvre de l'encapsulation
  - d'une architecture pensée et validée par le concepteur
  - des droits et d'espaces d'accès

# Encapsulation

- Notion essentielle dans la programmation objet
- Suppose une séparation entre l'**interface publique** (accessible) d'une classe et son **fonctionnement interne** qui ne doit pas être ni perturbé ni forcement connu par l'extérieur.
- Outils pour la mise en ouvre de l'encapsulation
  - d'une architecture pensée et validée par le concepteur
  - des droits et d'espaces d'accès
  - renforcement / relâchement des contraintes

| Accès | class | struct | union |
|---|---|---|---|
| public | possible | par défaut | par défaut |
| protected | possible | possible | impossible |
| private | par défaut | possible | impossible |

# Encapsulation

- Notion essentielle dans la programmation objet
- Suppose une séparation entre l'**interface publique** (accessible) d'une classe et son **fonctionnement interne** qui ne doit pas être ni perturbé ni forcement connu par l'extérieur.
- Outils pour la mise en ouvre de l'encapsulation
  - d'une architecture pensée et validée par le concepteur
  - des droits et d'espaces d'accès
  - renforcement / relâchement des contraintes

| Accès | class | struct | union |
|---|---|---|---|
| public | possible | par défaut | par défaut |
| protected | possible | possible | impossible |
| private | par défaut | possible | impossible |

Droits d'accès :

| Accès | classe elle-même | classe dérivée | exterieur |
|---|---|---|---|
| public | OUI | OUI | OUI |
| protected | OUI | OUI | NON |
| private | OUI | NON | NON |

# Droits d'accès : protection par const

- Méthode/fonction, paramètre précédé par const :
  - seulement pour les références et les pointeurs
  - signifie que l'objet pointé (référencé) ne sera pas modifié par le code de la méthode

# Droits d'accès : protection par const

- Méthode/fonction, paramètre précédé par **const** :
  - seulement pour les références et les pointeurs
  - signifie que l'objet pointé (référencé) ne sera pas modifié par le code de la méthode
- Méthode suivie de **const** :
  - l'objet ***this** est protégé par rapport à la méthode

# Droits d'accès : protection par const

- Méthode/fonction, paramètre précédé par **const** :
  - seulement pour les références et les pointeurs
  - signifie que l'objet pointé (référencé) ne sera pas modifié par le code de la méthode
- Méthode suivie de **const** :
  - l'objet ***this** est protégé par rapport à la méthode
- Variable (objet) protégée par **const** :
  - on ne peut plus le modifier mais on peut lire ses attributs et appeler ses méthodes constantes
  - une variable entière constante peut servir comme taille de tableau non-dynamique, valeur de case etc.

# Droits d'accès : protection par const

- Méthode/fonction, paramètre précédé par **const** :
  - seulement pour les références et les pointeurs
  - signifie que l'objet pointé (référencé) ne sera pas modifié par le code de la méthode
- Méthode suivie de **const** :
  - l'objet ***this** est protégé par rapport à la méthode
- Variable (objet) protégée par **const** :
  - on ne peut plus le modifier mais on peut lire ses attributs et appeler ses méthodes constantes
  - une variable entière constante peut servir comme taille de tableau non-dynamique, valeur de case etc.
- Déclaration de constantes spécifiques pour une classe :
  - eviter **#define** (accessible partout, conflit de noms, **#undef**)
  - utiliser des variables de type **static const**

# Encapsulation : architecture

- Comment protéger l'accès à un attribut :

# Encapsulation : architecture

- Comment protéger l'accès à un attribut :
  - complètement : par droit d'accès **private**

# Encapsulation : architecture

- Comment protéger l'accès à un attribut :
  - complètement : par droit d'accès **private**
  - sauf pour les héritiers : par droit d'accès **protected**

# Encapsulation : architecture

- Comment protéger l'accès à un attribut :
  - complètement : par droit d'accès **private**
  - sauf pour les héritiers : par droit d'accès **protected**
  - interdiction d'écriture : **private** et méthode publique **type_attr GetAttribute()**

# Encapsulation : architecture

- Comment protéger l'accès à un attribut :
  - complètement : par droit d'accès **private**
  - sauf pour les héritiers : par droit d'accès **protected**
  - interdiction d'écriture : **private** et méthode publique **type_attr GetAttribute()**
  - filtrage d'écriture : **private** et méthode publique **bool SetAttribute(type_attr)**

# Encapsulation : architecture

- Comment protéger l'accès à un attribut :
  - complètement : par droit d'accès **private**
  - sauf pour les héritiers : par droit d'accès **protected**
  - interdiction d'écriture : **private** et méthode publique **type_attr GetAttribute()**
  - filtrage d'écriture : **private** et méthode publique **bool SetAttribute(type_attr)**
- les méthodes de type **GetXXX** et **SetXXX** sont appelées des accesseurs (setters et getters)

# Encapsulation : architecture

- Comment protéger l'accès à un attribut :
  - complètement : par droit d'accès **private**
  - sauf pour les héritiers : par droit d'accès **protected**
  - interdiction d'écriture : **private** et méthode publique **type_attr GetAttribute()**
  - filtrage d'écriture : **private** et méthode publique **bool SetAttribute(type_attr)**
- les méthodes de type **GetXXX** et **SetXXX** sont appelées des accesseurs (setters et getters)
- pour des raisons d'efficacité, on les déclare **inline**

# Encapsulation : exemple

- Comment s'assurer que le dénominateur de la fraction `CFract` ne sera jamais nul ? Et que les deux attributs ne seront jamais négatifs en même temps ?

# Encapsulation : exemple

- Comment s'assurer que le dénominateur de la fraction `CFract` ne sera jamais nul ? Et que les deux attributs ne seront jamais négatifs en même temps ?
- Réorganiser la classe `CFract` pour attendre ces objectifs, sans restreindre l'accès en lecture aux deux attributs de la fraction.

# Encapsulation : exemple

- Comment s'assurer que le dénominateur de la fraction `CFract` ne sera jamais nul ? Et que les deux attributs ne seront jamais négatifs en même temps ?
- Réorganiser la classe `CFract` pour attendre ces objectifs, sans restreindre l'accès en lecture aux deux attributs de la fraction.
- Et si l'on rajoute la contrainte suivante : "l'utilisateur ne peut jamais modifier directement les deux attributs (seulement les initialiser à la construction)" ?

# Surcharge des opérateurs

A quoi cela sert ?

- écrire `a+b` à la place de `Add(a,b)` ou `a.Add(b)`

Comment ?

Quels opérateurs ?

# Surcharge des opérateurs

A quoi cela sert ?

- écrire `a+b` à la place de `Add(a,b)` ou `a.Add(b)`

Comment ?

- par méthode

Quels opérateurs ?

# Surcharge des opérateurs

A quoi cela sert ?

- écrire `a+b` à la place de `Add(a,b)` ou `a.Add(b)`

Comment ?

- par méthode
- par fonction

Quels opérateurs ?

# Surcharge des opérateurs

A quoi cela sert ?

- écrire `a+b` à la place de `Add(a,b)` ou `a.Add(b)`

Comment ?

- par méthode
- par fonction

Quels opérateurs ?

- par méthode/fonction : + - / * | || & && « » < > ++ −

# Surcharge des opérateurs

A quoi cela sert ?

- écrire `a+b` à la place de `Add(a,b)` ou `a.Add(b)`

Comment ?

- par méthode
- par fonction

Quels opérateurs ?

- par méthode/fonction : + - / * | || & && « » < > ++ −
- uniquement par méthode : = -> [] ()

# Surcharge des opérateurs

A quoi cela sert ?

- écrire `a+b` à la place de `Add(a,b)` ou `a.Add(b)`

Comment ?

- par méthode
- par fonction

Quels opérateurs ?

- par méthode/fonction : + - / * | || & && « » < > ++ −
- uniquement par méthode : = -> [] ()
- pas de surcharge : . .* : : : :* ? : sizeof

# Surcharge - exemple par fonction

```
class complexe
{
  float r, i;
  public:
    complexe(float _r, float _i): r(_r), i(_i) {}
    friend const complexe operator+ (const complexe& c1,
                  const complexe& c2);
};

const complexe operator+ (const complexe& c1,
        const complexe& c2)
{
  return complexe(c1.r+c2.r, c1.i+c2.i);
}
```

# Surcharge - exemple par méthode

```
class complexe
{
  float r, i;
  public:
    complexe(float _r, float _i): r(_r), i(_i) {}
    const complexe operator+ (const complexe& c2) const
    {
        return complexe(r+c2.r, i+c2.i);
    }
};
```

# Opérateurs pour `CFract`

Doter la classe `CFract` :

- d'un opérateur d'addition suivie d'attribution `+=` et de multiplication suivie d'attribution `*=`

# Opérateurs pour `CFract`

Doter la classe `CFract` :

- d'un opérateur d'addition suivie d'attribution `+=` et de multiplication suivie d'attribution `*=`
- d'un opérateur d'addition `+` et de multiplication `*`

# Opérateurs pour `CFract`

Doter la classe `CFract` :

- d'un opérateur d'addition suivie d'attribution `+=` et de multiplication suivie d'attribution `*=`
- d'un opérateur d'addition `+` et de multiplication `*`
- de post- et pre-incrementation (`++`), de comparaison (`==`)

# Opérateurs pour `CFract`

Doter la classe `CFract` :

- d'un opérateur d'addition suivie d'attribution `+=` et de multiplication suivie d'attribution `*=`
- d'un opérateur d'addition `+` et de multiplication `*`
- de post- et pre-incrementation (`++`), de comparaison (`==`)
- d'un opérateur de conversion emphtypecast vers un double

# Opérateurs pour `CFract`

Doter la classe `CFract` :

- d'un opérateur d'addition suivie d'attribution `+=` et de multiplication suivie d'attribution `*=`
- d'un opérateur d'addition `+` et de multiplication `*`
- de post- et pre-incrementation (`++`), de comparaison (`==`)
- d'un opérateur de conversion emphtypecast vers un double
- tester les opérateurs ainsi surchargés

# Opérateurs pour `CFract`

Doter la classe `CFract` :

- d'un opérateur d'addition suivie d'attribution `+=` et de multiplication suivie d'attribution `*=`

# Opérateurs pour `CFract`

Doter la classe `CFract` :

- d'un opérateur d'addition suivie d'attribution `+=` et de multiplication suivie d'attribution `*=`
- d'un opérateur d'addition `+` et de multiplication `*`

# Opérateurs pour `CFract`

Doter la classe `CFract` :

- d'un opérateur d'addition suivie d'attribution `+=` et de multiplication suivie d'attribution `*=`
- d'un opérateur d'addition `+` et de multiplication `*`
- de post- et pre-incrementation (`++`), de comparaison (`==`)

# Opérateurs pour `CFract`

Doter la classe `CFract` :

- d'un opérateur d'addition suivie d'attribution `+=` et de multiplication suivie d'attribution `*=`
- d'un opérateur d'addition `+` et de multiplication `*`
- de post- et pre-incrementation (`++`), de comparaison (`==`)
- d'un opérateur de conversion emphtypecast vers un double

# Opérateurs pour `CFract`

Doter la classe `CFract` :

- d'un opérateur d'addition suivie d'attribution `+=` et de multiplication suivie d'attribution `*=`
- d'un opérateur d'addition `+` et de multiplication `*`
- de post- et pre-incrementation (`++`), de comparaison (`==`)
- d'un opérateur de conversion emphtypecast vers un double
- tester les opérateurs ainsi surchargés