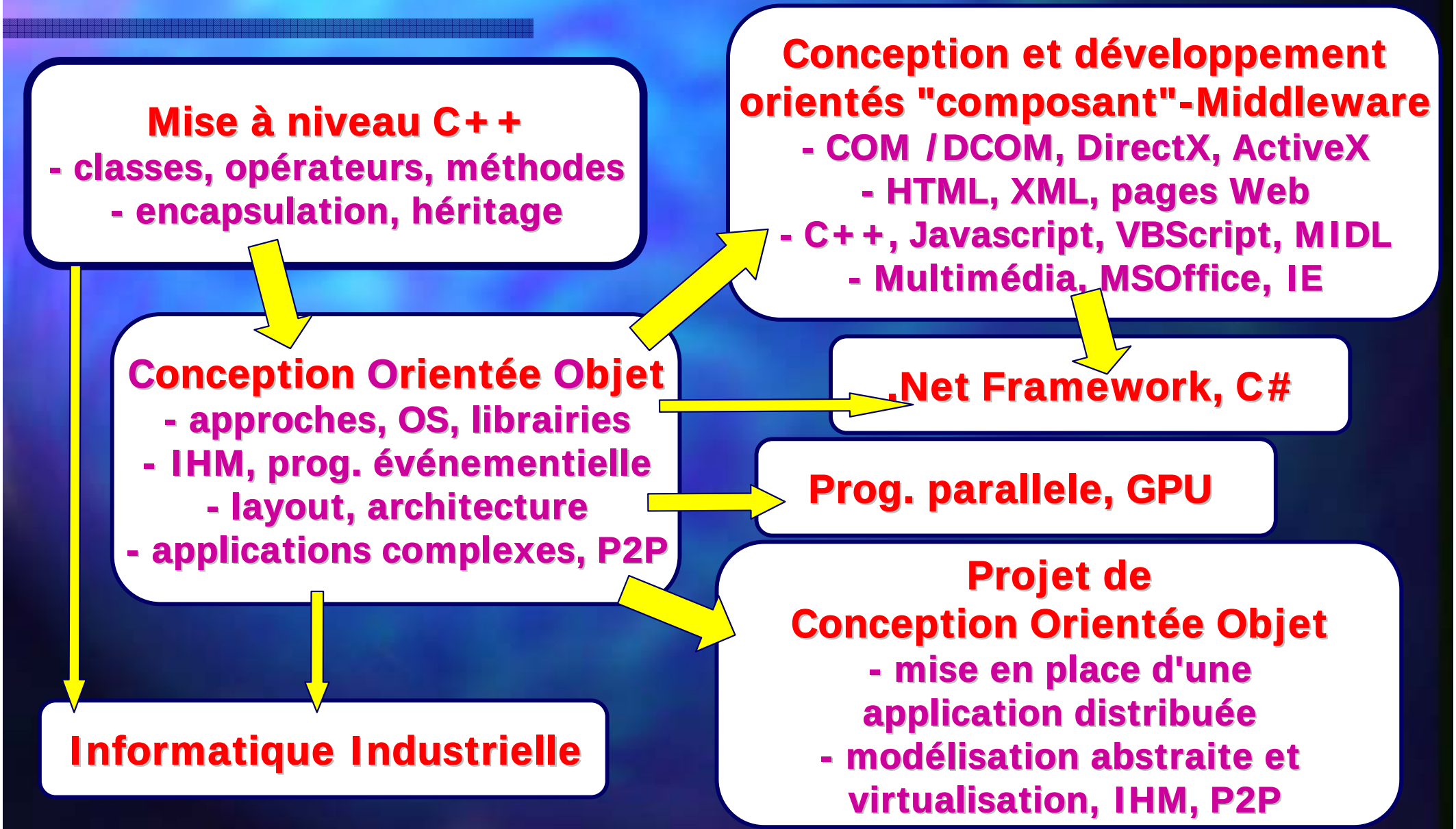
Techniques de conception et de programmation en Langage C++

Mise à niveau M2IST

2013/2014

Marius VASILIU

"Learning path"



Sommaire de la mise au niveau

- Introduction, extensions C++ : références, etc.
- Données en C++ : types, durée de vie, allocation
- Approche objet, construction et destruction
- Attributs, méthodes, pointeur *this*
- Encapsulation, droits d'accès, accèsseurs
- Surcharge d'opérateurs, flux d'entrée / sortie C++
- Instances et membres statiques,
- Héritage simple, multiple, virtuel

Sommaire de la mise au niveau

- Héritage et polymorphisme
- Polymorphisme statique et dynamique
- Bases abstraites, interfaces, gestion polymorphique
- *Templates* (patrons) de classes et fonctions, spécialisation
- Bibliothèques C++: STL, boost, OpenCV, IPP, Cuda, PhysX, SystemC etc.
- Gestion des exceptions

Origines et approche C++

- Extension syntactique du langage C
- Dernières définitions: ANSI/ISO 1995,1998,C0x
- Un compilateur C++ compile aussi du C
- Principale extension : les classes
- Différence importante dans l'approche de programmation / architecture du programme
- Compréhension du langage = compréhension de la réalité « physique » des données

  (condition essentielle pour l'info. industrielle)

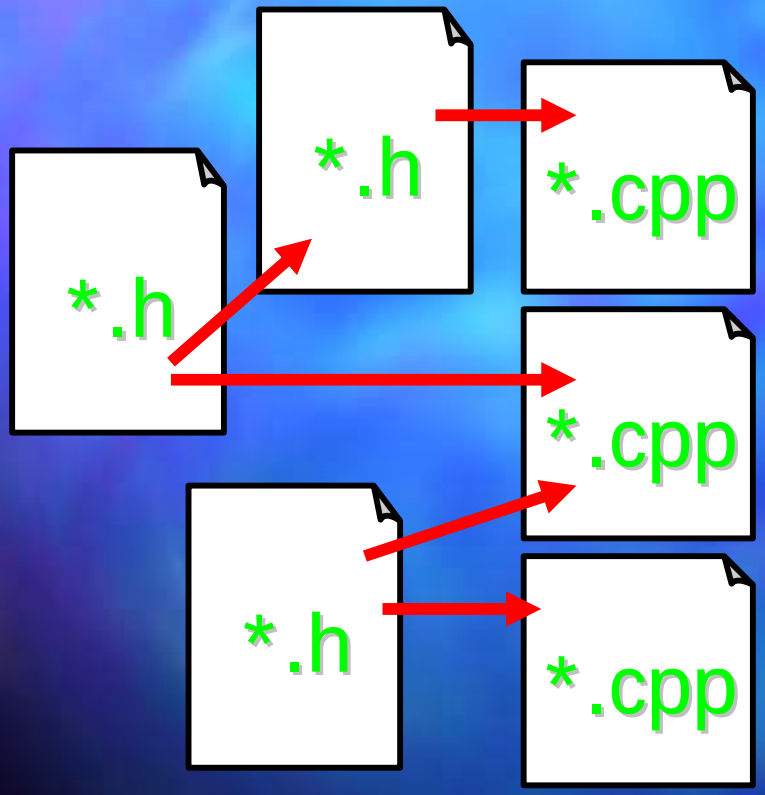
Extensions C++

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

Conception : que fait-il un programme C/C++ ?

- Approche « classique » C:
  - Crée, écrit, lit et détruit de données
  - Appelle des fonctions en passant et en recevant des paramètres
- Approche C++:
  - Crée et détruit d'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

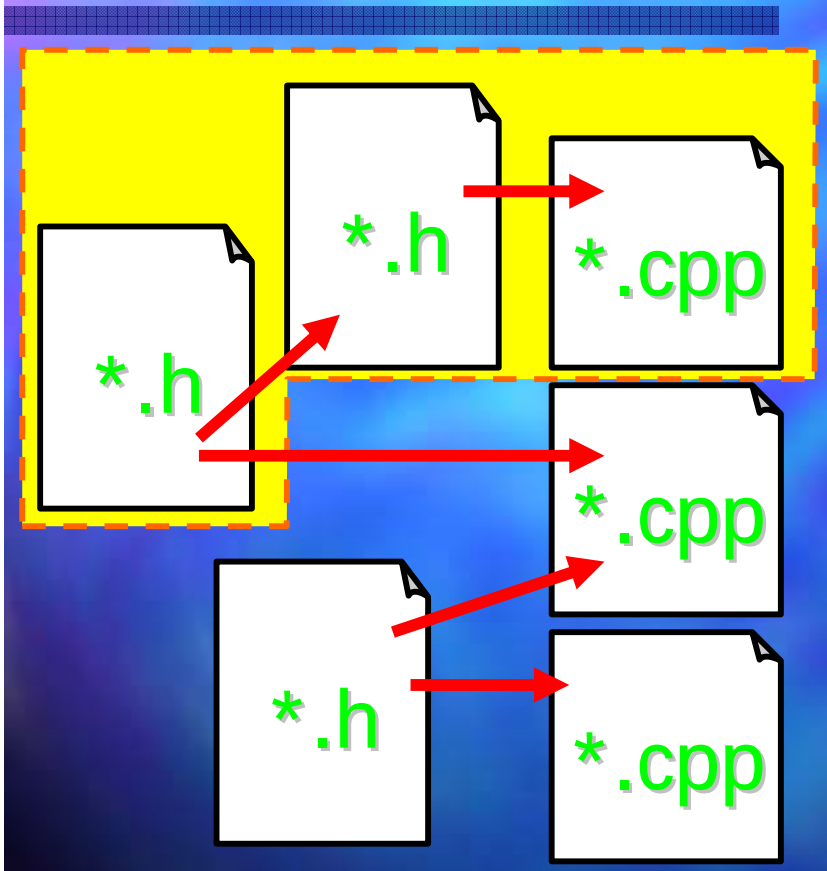
Comment le rendre opérationnel ?



Architecture (diagramme) de fichiers :

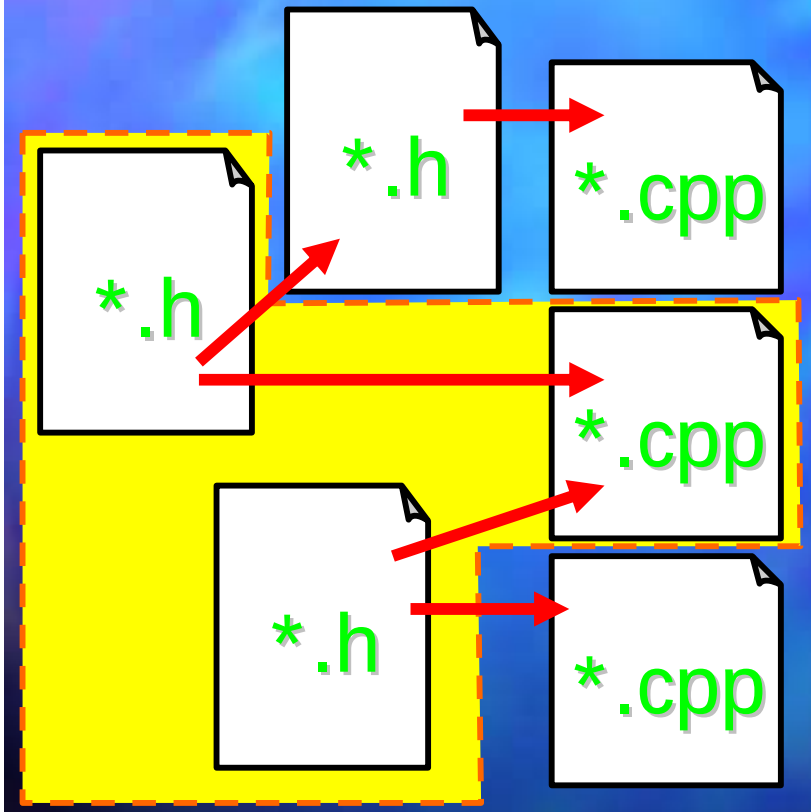
- par exemple, le programme a 3 modules sources

Comment le rendre opérationnel ?



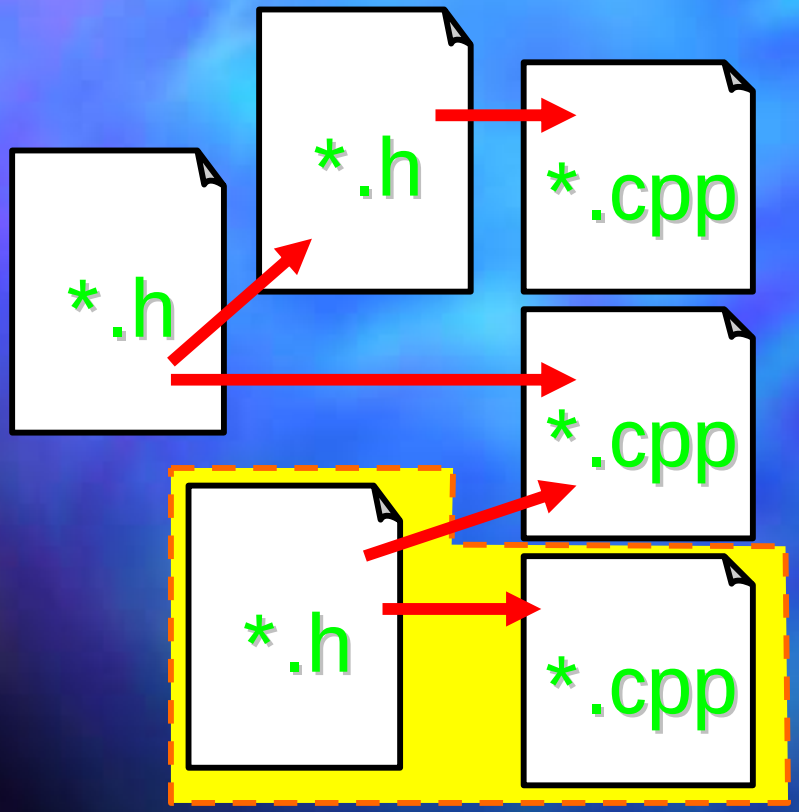
1er module

Comment le rendre opérationnel ?



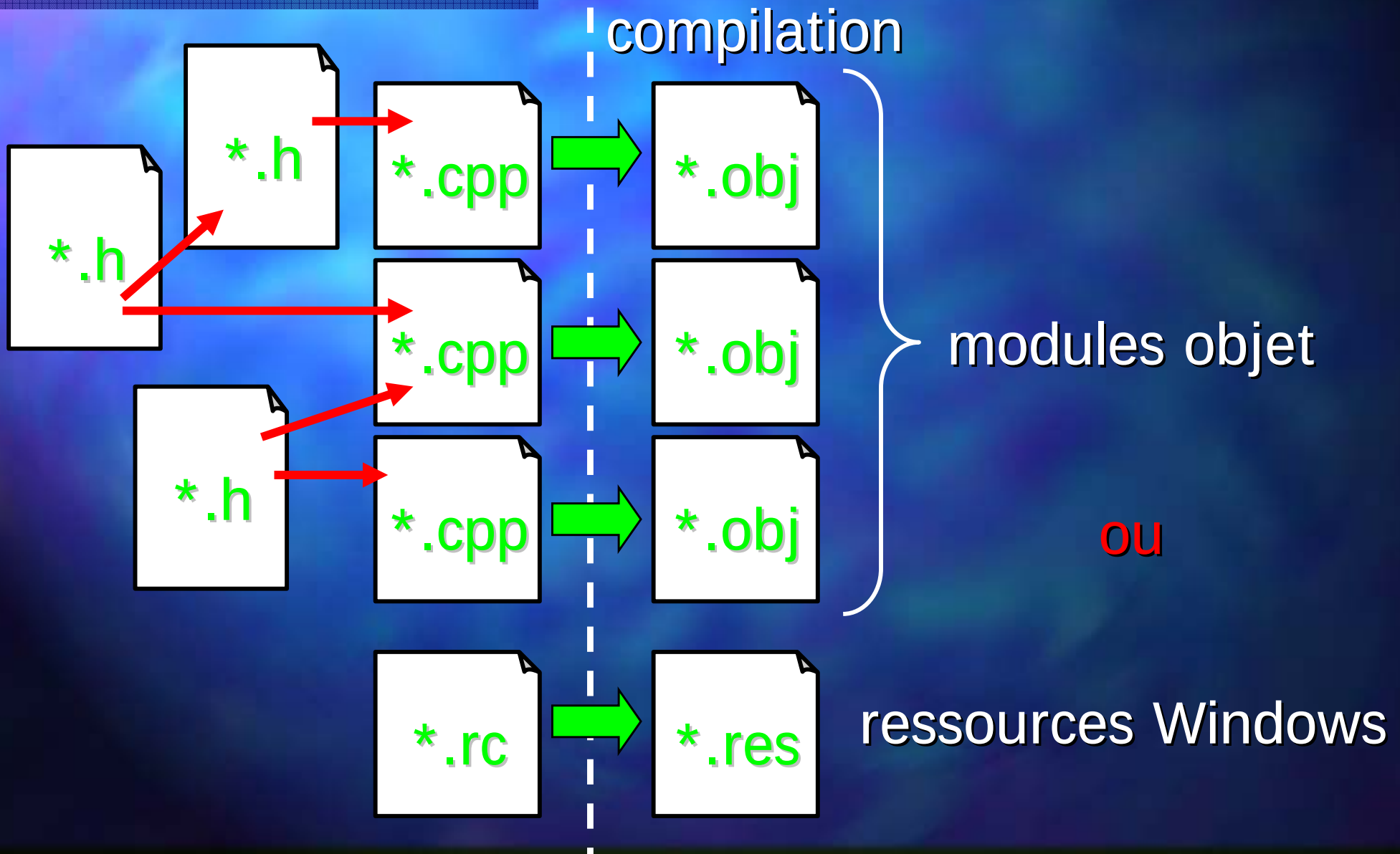
2ème module

Comment le rendre opérationnel ?

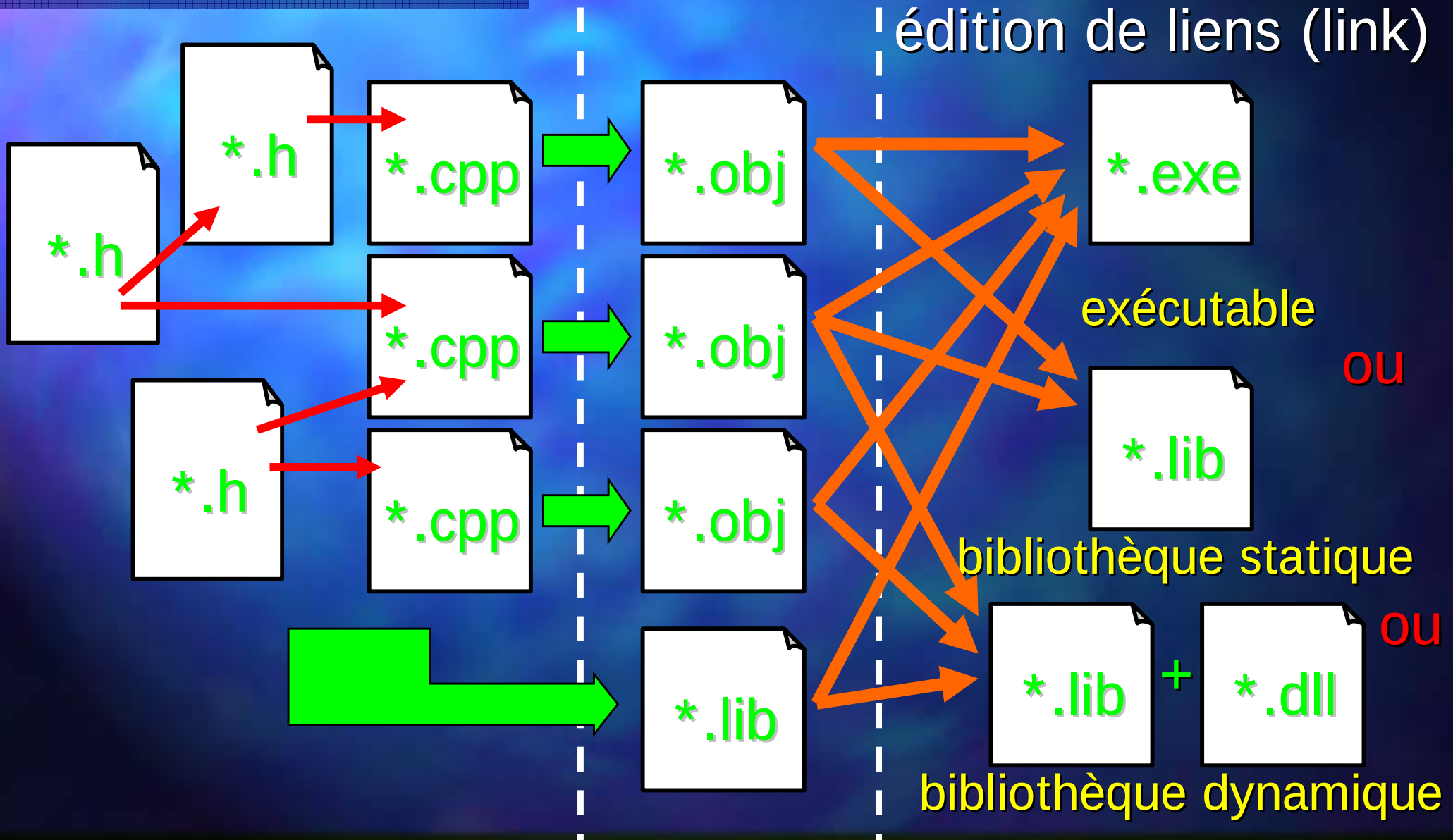


3ème module

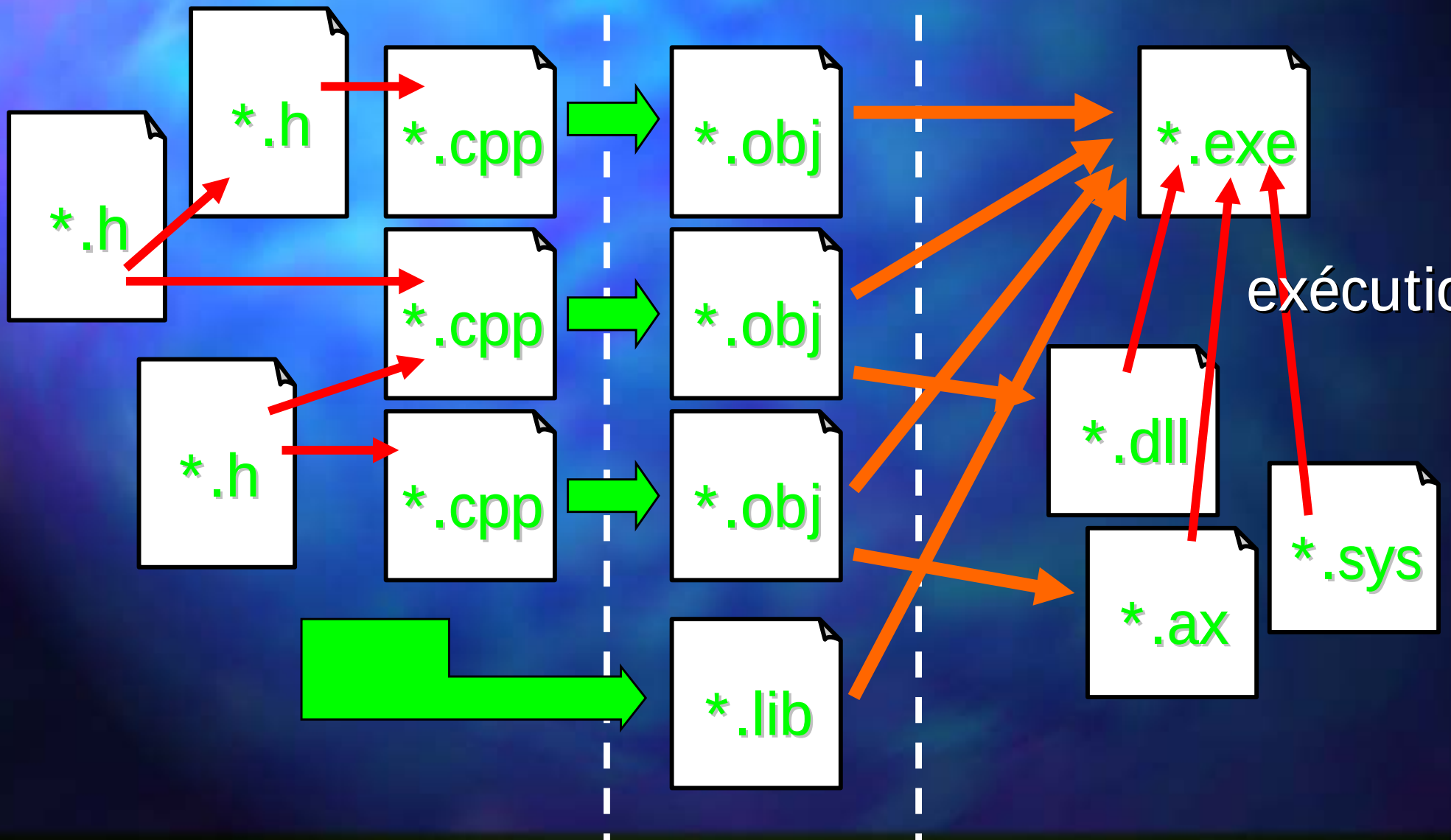
Comment le rendre opérationnel ?



Comment le rendre opérationnel ?



Comment le rendre opérationnel ?



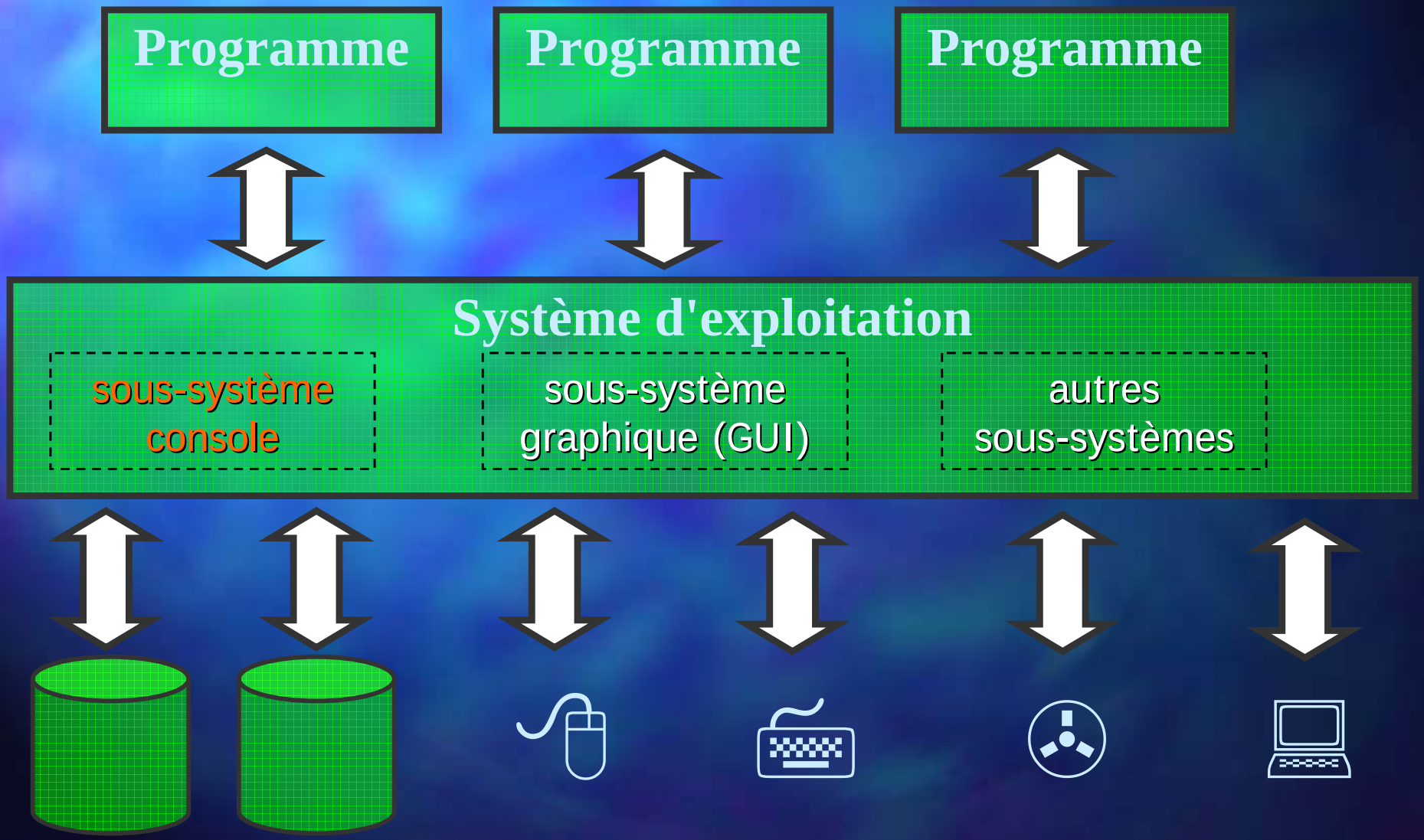
Comment fait-il (réellement) ?

- 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 + seg. de données
- Segment de code = instructions assembleur

(il n'y a pas d'interprétation ni de machine virtuelle)

- Exécution par un/plusieurs µP des instructions assembleur en mode exclusif ou en même temps que d’autres programmes.

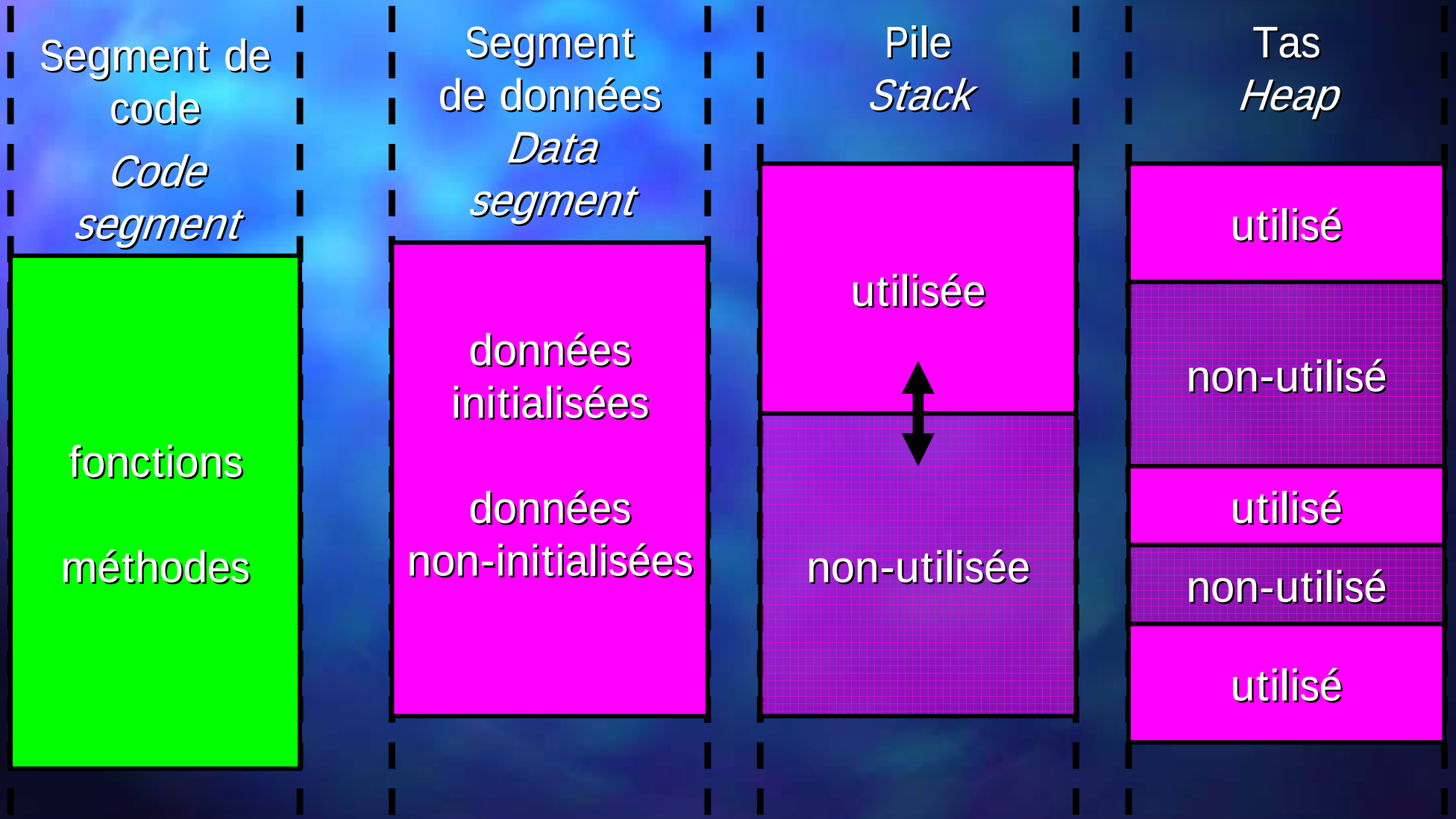
Exécution sous l'OS cible



Pour bien maîtriser les variables (objets), il faut ...

- Savoir que le C++ est un langage typé : toute variable et toute expression ont un (unique) type
- Ne jamais confondre un type avec une variable !
- Savoir que toute variable occupe une place mémoire de la taille du type pendant sa durée de vie
- Savoir quelle est la durée de vie d'une variable
- Savoir dans quel espace mémoire vie la variable
- Savoir où et comment une variable est accessible
- Savoir si l'évaluation d'une expression mène vers une variable (un conteneur) ou non: *l-value* et *r-value*.

Architecture mémoire sous-entendue par le C++



Données - Emplacement par défaut (sans optimisation)

- Segment de données
  - variables globales et statiques
- Pile (*stack*)
  - variables locales, temporaires
  - paramètres d'appel et retour de fonctions / méthodes
- Registres µP (VisualC++ ➔ pile)
  - variables précédées par le mot clé register
- Tas (*heap*)
  - variables dynamiques (sans nom !)

Données - Emplacement suite à l'optimisation (*Release*)

- Segment de données
  - variables globales et statiques
- Pile (*stack*)
  - variables locales, temporaires
  - paramètres d ’appel et retour de fonctions / méthodes
- Registres µP
  - toute variable locale, temporaire ou paramètre de fonction *in-line* que le compilateur considère nécessaire
- Tas (*heap*)
  - variables dynamiques (sans nom !)

Données - Portée Accessibilité (lire / écrire)

- Directe
  - par le nom de la variable (= visibilité)
    - Globales: dans le module courant et dans les autres à partir de la redéclaration avec extern
    - Globales statiques: dans le module courant
    - Locales: jusqu'au bout du bloc courant
    - Paramètres: à l'intérieur de la fonction appelée
    - Dynamiques, retour ou temporaires: jamais (pas de nom !)
  - Une variable peut en cacher une autre !
    - (Statiques)Locales > Paramètres > Attributs > Globales
    - Espaces : instruction < bloc < fonction/méthode < classe < *namespace* < module < programme

Données - Portée Accessibilité (lire / écrire)

- 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 **::**
  - la complexité d'une expression n'a pas de limites
  - Danger si l'original n'existe pas ou n'existe plus :

    le langage C++ ne garanti 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 / méthodes et **()**
  - le nom seul d'une fonction est une adresse dont le type est déterminé par le prototype de la fonction
- 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

main.cpp

```
static int i1=30;
double d2=3.14;
float fn(float f1)
{
  char c='d';
  add(&i1,(int)c);
  return f1;
}
long var2=78;
void main()
{
}
```

prog2.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;
}
```

Données - Accessibilité

main.cpp

```
static int i1=30;
double d2=3.14;
float fn(float f1)
{
  char c='d';
  add(&i1,(int)c);
  return f1;
}
long var2=78;
void main()
{
}
```

prog2.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;
}
```

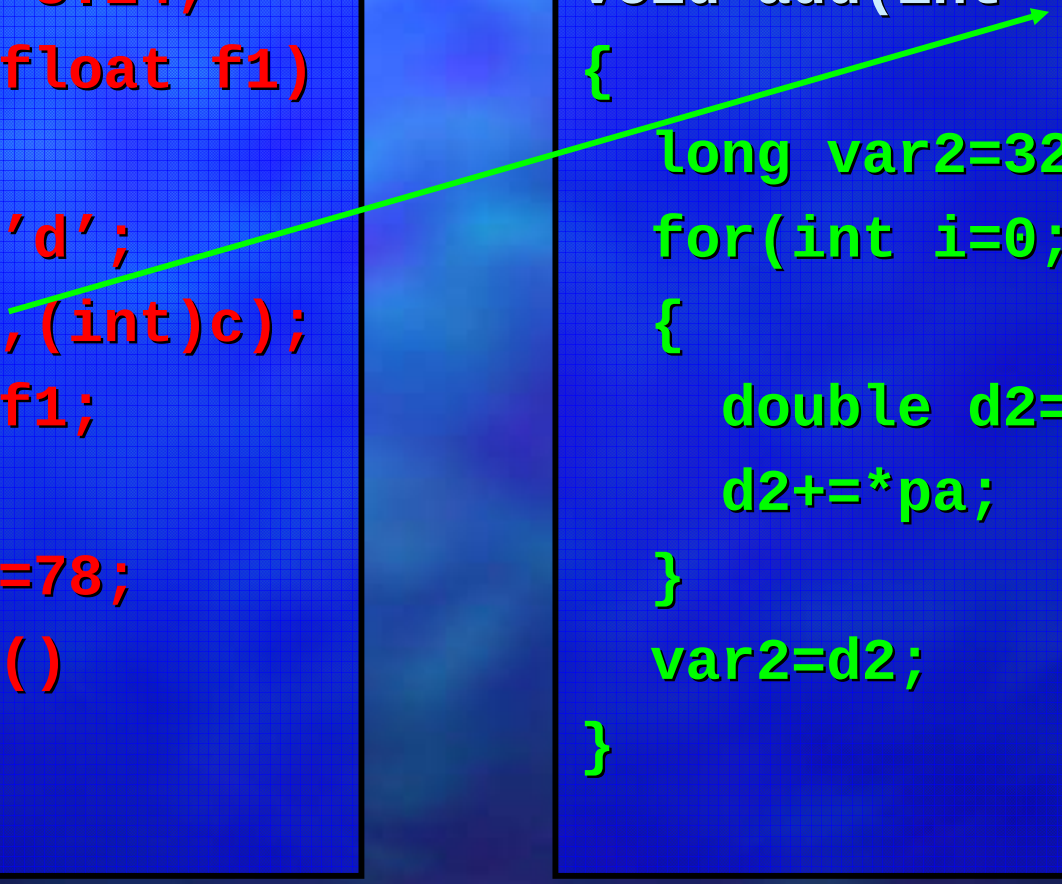
Données - Accessibilité

main.cpp

```
static int i1=30;
double d2=3.14;
float fn(float f1)
{
  char c='d';
  add(&i1,(int)c);
  return f1;
}
long var2=78;
void main()
{
}
```

prog2.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;
}
```



Données - Accessibilité

main.cpp

```
static int i1=30;
double d2=3.14;
float fn(float f1)
{
  char c='d';
  add(&i1,(int)c);
  return f1;
}
long var2=78;
void main()
{
}
```

prog2.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;
}
```

Données - Accessibilité

main.cpp

```
static int i1=30;
double d2=3.14;
float fn(float f1)
{
  char c='d';
  add(&i1,(int)c);
  return f1;
}
long var2=78;
void main()
{
}
```

prog2.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;
}
```

Données - Accessibilité

main.cpp

```
static int i1=30;
double d2=3.14;
float fn(float f1)
{
  char c='d';
  add(&i1,(int)c);
  return f1;
}
long var2=78;
void main()
{
}
```

prog2.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;
}
```

Données - Accessibilité

main.cpp

```
static int i1=30;
double d2=3.14;
float fn(float f1)
{
  char c='d';
  add(&i1,(int)c);
  return f1;
}
long var2=78;
void main()
{
}
```

prog2.cpp

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

Données - Accessibilité

main.cpp

```
static int i1=30;
double d2=3.14;
float fn(float f1)
{
  char c='d';
  add(&i1,(int)c);
  return f1;
}
long var2=78;
void main()
{
}
```

prog2.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;
}
```

Données - Accessibilité

main.cpp

```
static int i1=30;
double d2=3.14;
float fn(float f1)
{
  char c='d';
  add(&i1,(int)c);
  return f1;
}
long var2=78;
void main()
{
}
```

prog2.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;
}
```

normes
ANSI/ISO C++

Données - Accessibilité

main.cpp

```
static int i1=30;
double d2=3.14;
float fn(float f1)
{
  char c='d';
  add(&i1,(int)c);
  return f1;
}
long var2=78;
void main()
{
}
```

prog2.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;
}
```

Visual C++ 6.0

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
{
  // déclarations de type (et de classes)
  // déclarations de variables
  // déclarations et définitions de fonctions
}
```

- Y faire référence :

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

```
using namespace Espace1
variable
```

Espace de noms - Exemple

```
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]);
  }
}
```

Espace de noms imbriquées (accès indirect)

```
namespace EspaceCouleurs
{
  namespace Anglais
  {
    // ...
  }
  namespace Francais
  {
    // ...
  }
}

void main()
{
  EspaceCouleurs::Francais::Couleurs(2);
}
```

Espace de noms imbriquées (accès direct)

```
namespace EspaceCouleurs
{
  namespace Anglais
  {
    // ...
  }
  namespace Francais
  {
    // ...
  }
}

using namespace EspaceCouleurs;
using namespace Français;
void main()
{
  Couleurs(2);
}
```

Données - Durée de vie

- Entre la création = l'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
- Temporaires et ad-hoc
  - jusqu'à la fin de l'instruction courante « ; »
- Paramètres d'appel et de retour
  - pendant la durée d'appel de la fonction
- Dynamiques
  - entre création (new) et destruction (delete)

Types natifs 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
- 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 :

```
void main
{
  int i, j=10;
  int& ri=i; int &ri2 = ri;
  // afficher les valeurs de i et ri
  // afficher l'adresse de i et ri
}
```

Références - Utilisation

- On peut utiliser partout une référence à la place de sa variable associée (le type est le même à l'évaluation !)
- Copiez, modifiez la variable, puis la référence et affichez les résultats.
- La référence cache en réalité un pointeur vers la variable associée. La référence d'une variable var vaut toujours *(&var) .
- Peut-on avoir l'adresse d'une référence ?
- Peut-on modifier la variable associée ?

Passage de paramètres par référence

- Si un paramètre formel d'une fonction est une référence alors à l'appel on se retrouve avec la référence du paramètre d'appel :

```
double Add1(float a, short b)
{
  return a+b;
}
void Add2(float a, short b, double res)
{
  res=a+b;
}
void Add3(float a, short b, double& res)
{
  res=a+b;
}
```

Traduction référence - pointeur

- On peut toujours réécrire un programme utilisant une référence pour qu'il utilise un pointeur (l'inverse n'est pas toujours vrai) :

```
void Add3Ptr(float a, short b, double* pres)
{
  *pres=a+b;
}
void main()
{
  float a1=4.5f;
  int b=3;
  double result;
  Add3Ptr(a1,b,&result); // version pointeur
  Add3(a1,b,result);     // version référence
}
```

Types disponibles par agrégation

- Homogènes (vecteur d'éléments = tableau)

  (une matrice 2D est un vecteur de vecteurs de ...)

  - Type de l'élément (qui est une donnée ...)
  - Nombre d'éléments (à la création !)
  - Adresse du premier élément
- Hybrides (structures, unions, classes)
  - Liste de champs/attributs (qui sont des données)
    - Nom, type, droit d'accès
    - Pour entiers: taille en bits (champs de bits)
  - Pour classes: liste et nom des méthodes (qui *par défaut* ne sont pas de données ...)

Tableaux (à taille fixe)

- Par déclaration, comme variables locales, globales ou statiques. Pourquoi une taille fixe ?
- Entant que paramètre de fonction il n'est pas copié dans la pile !
- L'opérateur sizeof donne sa vraie taille
- Il n'y a pas de tableaux 2D ou ND :

```
void main
{
  int tab1[3][5]; // emplacement, architecture ?
  // quel type? est-ce une l- ou r-value ?
  //      tab1
  //      tab1[1]
  //      (tab1+1)
  //      *(tab1+1)
  //      tab1[0][3]
}
```

Allocation dynamique

- En C++ on utilise pratiquement jamais malloc et free. A leur place 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 allouée dans le tas :

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

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

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

Allocation dynamique

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

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

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

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

Allocation dynamique Attention !

- Le pointeur qui accueille l'adresse de new est votre seul lien avec la (les) variable(s) allouée(s)
- Ne modifiez pas ce pointeur 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.
- 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 Exemples - Exos

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

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

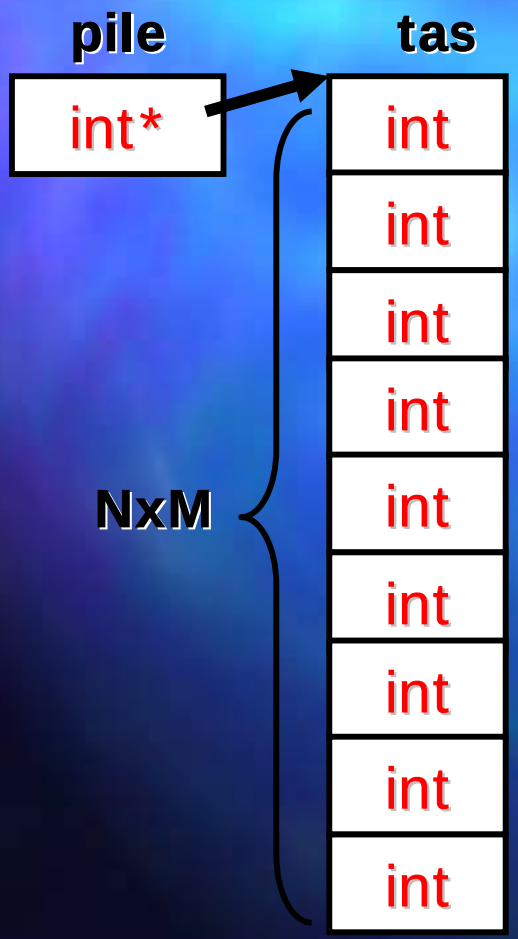
- Testez la fonction et dessiner l'emplacement de chaque variable.
- Ecrire une fonction qui alloue une matrice 2D de taille voulue.

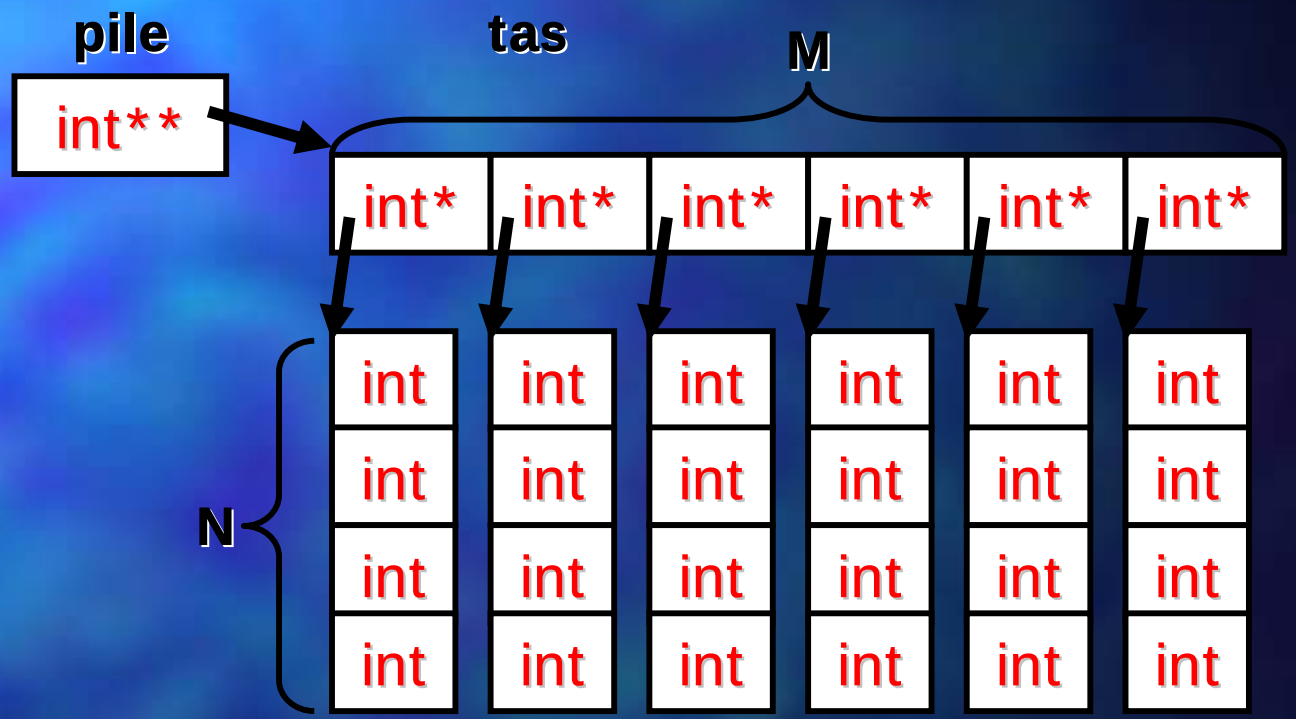
Allocation dynamique Exemples - Exos

- Ecrire 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 ou pour les colonnes.
  - testez la matrice dans le programme principal, en comparant son utilisation à une matrice 2D locale (initialisation et affichage).
  - dessiner l'emplacement des variables en mémoire et leurs relations (architecture de données).
- Même type de programme pour le triangle de Pascal.

Architecture (diagramme) de données

- 2 approches, une seule conforme aux demandes :





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 décorée 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++ Signature

- Rechercher dans msvcrt.dll des fonctions C++ (avec signature) et C (sans signature)
- Pour compiler comme en C il faut le demander:

```
#ifdef __cplusplus
extern "C"
{
#endif

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

#ifdef __cplusplus
}
#endif
```

Fonctions C++ Signature

- Créer une bibliothèque dynamique Win32 Test et choisir l'option A Dll that exports same symbols
- Rajouter dans l'entête la déclaration d'exportation des trois fonctions Add déjà utilisées et leur définition puis compilez la bibliothèque DLL.
- Regardez avec Depends les fonctions exportées. Quels sont les noms de vos fonctions Add ?
- Comment faire pour garder leur nom d'origine ?

Fonctions C++
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.

Ecrire la fonction, faites afficher les paramètres et testez-là avec 2, 3 et 4 paramètres

Fonctions C++
Protection à la modification

- Parfois on passe par des références ou 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);
void AfficheTab(const double* t1, int taille);
```

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

```
void Add(float& a, float& b, float& res);
void Add(const float& a,const float& b, float& res);
```

- Idem pour les références ou les pointeurs de retour

Protection à la modification Exemple

- 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 de test 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.

Autre 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 de son l'accès:
  - il va lire chaque fois la variable même s'il l'a eu une instruction plutôt
  - il va l'écrire aussitôt, sans la garder dans la mémoire cache ou dans des registres
- Attention :
  - cela ne rend pas l'accès atomique !
  - la vitesse d'accès diminue dramatiquement (10-100 fois)

Déclaration des types en C++

Types C++

types natifs

types complexes

types de classes

types de structures

types d'unions

Instances

Instances

Instances

Données

Variables

Objets

Types en C++

- Les types en C++ :
  - natifs au langage : scalaires (et pointeurs)
  - complexes par agrégation homogène (+ typedef)
  - complexes par agrégation hybride
    - nouveau type d'union (typedef + union)
    - nouveau type de structure (typedef + struct)
    - nouveau type de classe (typedef + class)

☺ mais le typedef n'est pas obligatoire en C++ pour l'agrégation hybride car la syntaxe est plus libre ...

Déclaration des types en C++

- Avec typedef on ne déclare que des types :

```
typedef union UnionType1
{
  // ...
} UnionType2;

typedef struct StructType1
{
  // ...
} StructType2;

typedef class ClassType1
{
  // ...
} ClassType2;
```

en C :
struct StructType1
ou
StructType2

en C++ :
StructType1
ou
StructType2

Déclaration des types en C++

- Sans typedef on déclare des types et des variables:

```
union UnionType1
{
  // ...
} Union1, Union2;

struct StructType1
{
  // ...
} Struct1, Struct2;

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

type en C :
struct StructType1

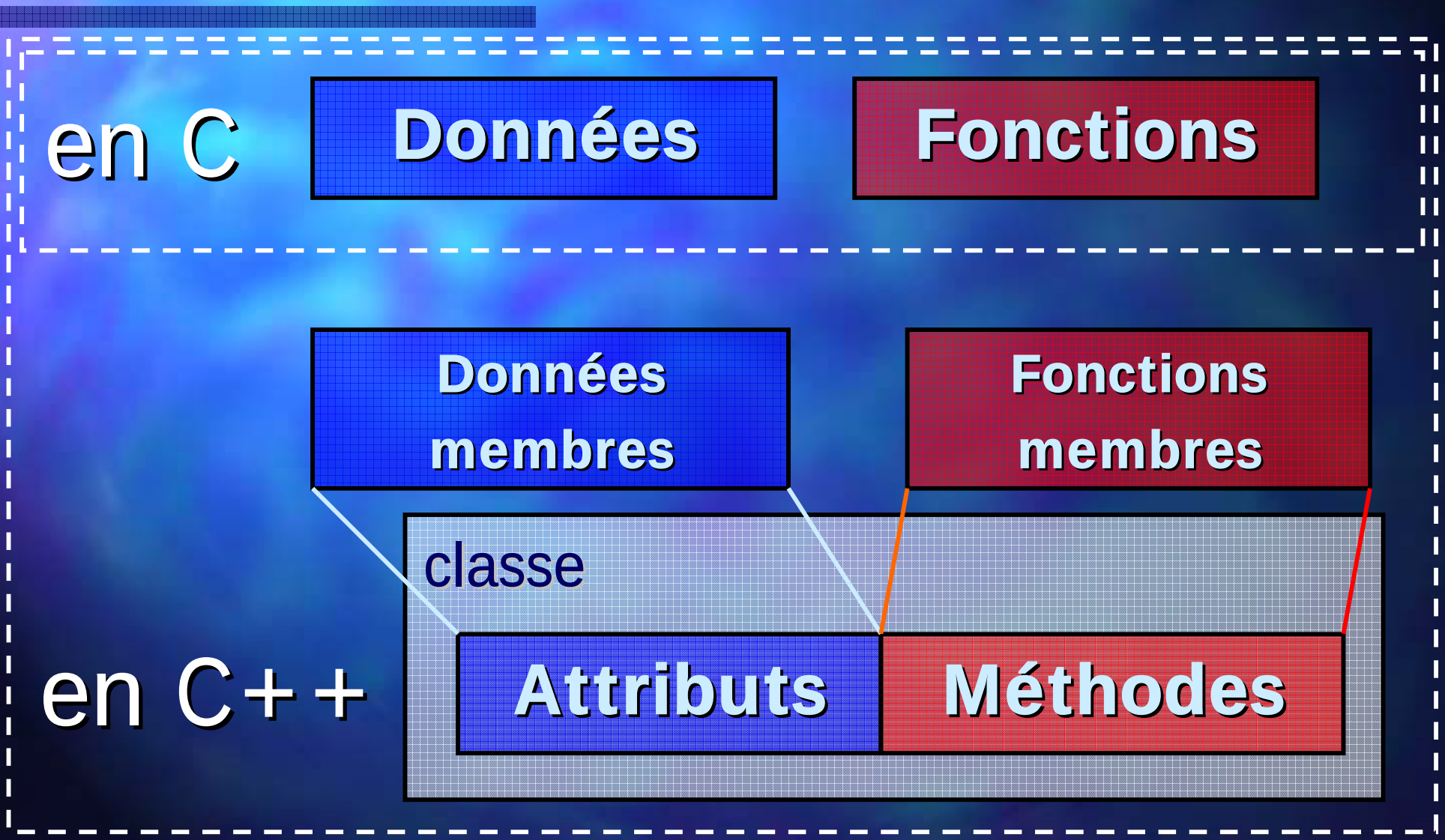
type en C++ :
StructType1

Déclaration des types en C++

- Mais les règles de bonne programmation nous imposent de déclarer séparément les types et les variables (le typedef n'est plus nécessaire) :

```
union UnionType
{
  // ...
};
struct StructType
{
  // ...
};
class ClassType
{
  // ...
};
```

Approche objet



Approche objet

- Classe
  - un type de données parmi les 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 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
- Appel d'un des constructeurs

Utilisation

- Lectures / écritures des attributs accessibles
- Appels des méthodes accessibles

Destruction

- Appel de l'unique destructeur
- Libération de la mémoire : sizeof(type_classe) octets

Approche objet

- En C++ toute donnée est un objet et inversement
- Au début de la vie : construction ⇨ appel d'un constructeur (même un qui ne fait rien)
  - tout type possède deux constructeurs par défaut :
  - un constructeur sans paramètres (qui ne fait rien)
  - un constructeur copie (qui copie le contenu)
- A la fin de la vie : destruction ⇨ appel d'un destructeur (même un qui ne fait rien)
  - tout type (natif) possède un destructeur par défaut qui ne fait rien

Approche objet

- Exemple : même le type natif int est une classe !

```
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.

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 1er 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(paramètres)
```

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

```
objet.NomMethode(paramètres)          // ou
ptrobjet->NomMethode(paramètres)    // idem à: (*ptrobjet).
```

Définitions de méthodes

- *In line*
  - déclaration suivie de définition à l'intérieur de la déclaration de classe
  - typiquement dans un fichier *.h

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

Définitions de méthodes

- *Out line*
  - déclaration à l'intérieur et définition à l'extérieur de la déclaration de classe
  - typiquement dans un fichier *.cpp

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

déclaration

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

définition

Exemple - CFract

- Modéliser une fraction entière :

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

public par défaut

- Elle a déjà deux constructeurs par défaut (sans paramètres et copie) et un destructeur par défaut

Exemple - CFract

- Modéliser une fraction entière :

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

private par défaut

- Elle a déjà deux constructeurs par défaut (sans paramètres et copie) et un destructeur par défaut

Exemple - CFract

- Sans faire aucun effort, le compilateur fournit le code suivant (en rouge):

```
class CFract
{
public:
  int a, b;
  void Afficher()
  {
    printf("Fraction : (%d/%d)\n",a,b);
  }
  CFract() {}
  CFract(const CFract& f) { a=f.a; b=f.b }
  ~CFract() {}
  CFract& operator= (const CFract& f)
    { a=f.a; b=f.b; return *this; }
};
```

adresse de l'objet pour lequel on travaille

Constructeurs et destructeur par défaut / explicites

- Tout constructeur explicite (sauf celui de copie) désactive la génération automatique du constructeur par défaut sans paramètres
- Tout constructeur de copie explicite désactive la génération automatique du constructeur par défaut de copie
- L'écriture d'un destructeur explicite arrête la génération automatique de celui par défaut

Exemple - CFract

- 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 copie pour faire que f3 ait le même contenu que f1.
- Même exercice avec une instance dynamique de la classe CFract pointé par pf4. Faire que cette variable ait le contenu de f2, ou inversement, que f2 ait son contenu.

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 Init ou par le constructeur CFract :

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

Exemple - CFract

- 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.

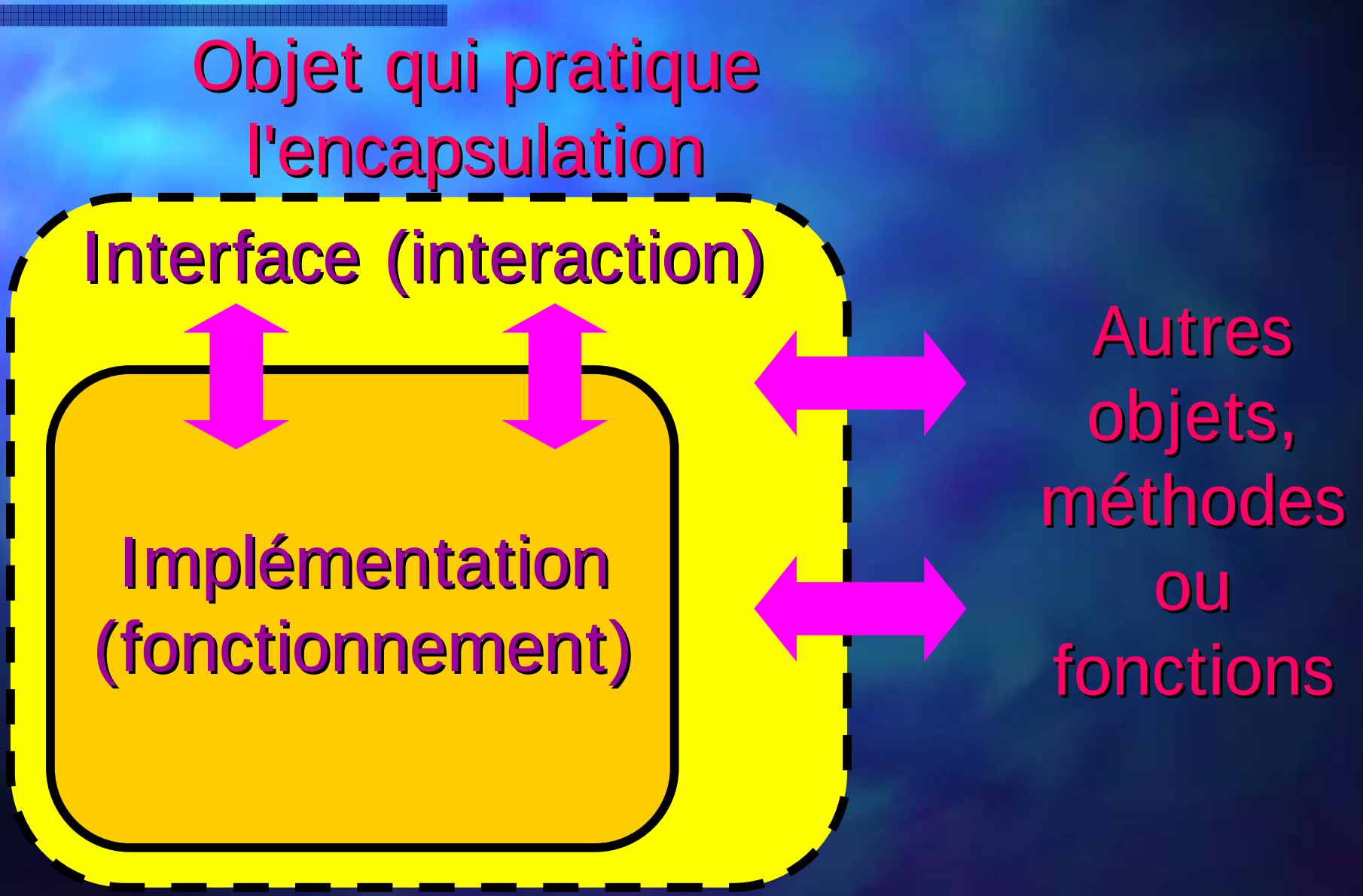
Méthodes
Exemple - CFract

- 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
- 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 :
  - des droits et d'espaces d'accès
  - d'une architecture pensée et validée par le concepteur
  - renforcement / relâchement des contraintes

Encapsulation



Encapsulation
Droits d'accès - Utilisation

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

Encapsulation
Droits d'accès - Signification

Accès	classe elle-même	classe dérivée	extérieur
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 (référencée) 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 d'utiliser **#define** :
  - accessible à tout le monde, conflit de nom possible
  - affecté par **#undef**
- Utiliser des constantes statiques et/ou **enum** :
  - à l'intérieur de la classe dans la zone appropriée
  - plongés dans le *namespace* de la classe
  - on peut initialiser *inline* les constantes statiques entières et les enum (typés ou anonymes) !
  - la taille des **enum** n'est pas toujours celle d'un **int**, le compilateur décide en fonction de la plus grosse valeur
  - on peut continuer à utiliser pour les constantes des noms "tout en majuscules".

Déclaration de constantes spécifiques pour une classe

```
class Palette
{
public:
   static const unsigned int SZ_MAX=256;
   enum {RED_COLOR=0x0000FF, WHITE_COLOR=0xFFFFFF};
   enum TypePal {GRAY_PAL=0, RAINBOW_PAL, HOT_PAL};
protected:
   DWORD pal[SZ_MAX];
public:
   Palette(TypePal type=GRAY_PAL)
   {
      switch(type) {
         case GRAY_PAL: pal[0]=WHITE_COLOR; ...
      }
   }
};
```

Relâchement de la protection par const

- On peut rajouter le qualificatif **mutable** aux attributs non-const et non-static pour permettre leur modification dans une méthode **const** de la classe. *Déconseillé !*
- Exemple :

```
class ClasseA
{
  mutable int nbShow;
  void Show() const
  {
    nbShow++; // ceci est permis car nbShow est mutable
    printf( /* ... */ );
  }
};
```

Relâchement des droits d'accès

- On peut déclarer des classes ou des fonctions amies à l'aide de **friend** : *Déconseillé !*

```
class ClasseA
{
  friend ClasseB;
  friend void FonctionC();
  //...
};
```

  - 📖 Le code des méthodes de ClasseB ou de la fonction FonctionC aura les mêmes droits d'accès que les méthodes de ClasseA qui les a déclarées
  - 📖 L'amitié n'est ni réciproque ni transitive !

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 :
    - droit d'accès private
    - méthode publique de lecture : type_attr GetAttribut()
  - filtrage d'écriture / modification :
    - droit d'accès private / protected
    - méthode publique d'écriture : bool SetAttribute(type_attr val)

📖 *Les méthodes de type GetXXX et SetXXX on les appelle des **accèsseurs**. 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 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 ça sert ?
  - écrire a+b à la place de Add(a,b) ou a.Add(b)
- Deux formes :
  1. méthodes
  2. fonctions
- Quels opérateurs ?
  - ❶ et ❷ : + - / * | || & && << >> < > ++ --
  - seulement ❶ : = -> [] ()
  - ni ❶ ni ❷ : . .* :: ::* ?: sizeof
  - spécifique : , = () [] new delete

Surcharge d'opérateur par une fonction

- Syntaxe de la déclaration :

  TypeResultat operator Symbol (Args);

- Syntaxe de la définition :

  TypeResultat operator Symbol (Args) {CorpFnct};

- Nb. d'arguments = nb. d'arg. de l'opérateur
  - pour un op. binaire : 2 arguments (gauche, droite)
  - pour un op. unaire : 1 argument

Surcharge d'opérateur par une méthode

- Syntaxe de la définition *in line* :

```
class ClassObj
{ ...
TypeResultat operator Symbol (Args) {CorpMeth}
}
```

- Syntaxe de la définition *out line* :

```
class ClassObj
{ Declaration }
TypeResultat ClassObj::operator Symbol (Args)
   {CorpMeth}
```

Surcharge d'un opérateur binaire "op"

class1 class2

obj1 op obj2

obj3

class3

Quand le compilateur rencontre une telle expression, il cherche une surcharge de l'opérateur "**op**" par une fonction ou par une méthode.

```
const class3& operator op(const class1& obj1, const class2& obj2)
{ ... }          // en fonction des const et &:  4x3x3 versions possibles
```

```
class class1 {
   const class3& operator op(const class2& obj2) const { ... }
... };           // en fonction des const et &: 4x3x2 versions possibles
```

Surcharge d'un opérateur unaire "op"

class1 **obj1 op** ou **op obj1** class1

obj2 class2 **obj2** class2

Quand le compilateur rencontre une telle expression, il cherche une surcharge de l'opérateur "**op**" par une fonction ou par une méthode.

```
const class2& operator op(const class1& obj1)
{ ... }      // en fonction des const et &: 4x3 versions possibles
```

```
class class1 {
   const class2& operator op( ) const { ... }
... };       // en fonction des const et &: 4x2 versions possibles
```

Exemple de surcharge d'opérateur 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);
}
```

Exemple de surcharge d'opérateur 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);
  }
};
```

Cas particuliers de surcharge : transtypage et *functors*

- Les opérateurs de transtypage (*typecast*) :
  - la syntaxe n'admet pas de type de retour (comme pour les constructeurs) car il est implicitement défini par le nom de l'opérateur de *typecast*
  - le code doit toujours retourner une valeur du type de l'opérateur en question (ou compatible)
  - exemple de surcharge (toujours méthode !) :

```
operator int() const { return expression_de_type_int; }
```

- La surcharge de l'opérateur parenthèses "()" peux avoir n'importe quel nombre de paramètres :

```
bool operator()(type1 arg1, type2 arg2, type3 arg3) { ... }
```

Cas particuliers de surcharge: pré / post incré / décrémentation

- Pour faire la différence entre la syntaxe de surcharge des opérateurs de post- et pré-incrémentation (++) ou post- et pré-décrémentation (--) (méthodes !):
  - l'opérateur de pré-incrémentation n'a pas d'arguments
  - attention : comme comportement classique, il est censé retourner la variable incrémentée comme l-value !
  - l'opérateur de post-incrémentation reçoit un argument de type int (qui normalement doit être ignoré)
  - attention : comme comportement classique, il est censé retourner une copie de la variable avant l'incrémentation (comme r-value)

Opérateurs pour CFract Exemple

- 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 pré-incrémentation (`++`), de comparaison(`==`)
  - d'un opérateur de conversion (*typecast*) vers un double
  - d'un opérateur d'addition avec un entier (nécessaire ?)
- Chaque opérateur doit avoir un comportement aussi proche que possible de la version "classique"
- Tester les opérateurs ainsi surchargés

Gestion des ressources système en C++

- Ressource : mémoire dynamique, fichiers, objets graphiques, canaux de communication etc.
- Règles de gestion : on alloue une ressource, on l'utilise et on la libère aussitôt que possible
- Il faut s'assurer que l'on oublie jamais de libérer (une seule fois) les ressources allouées
- En C++ la gestion est très simple si l'on respecte scrupuleusement les règles suivantes de "bonne gestion" de ressources

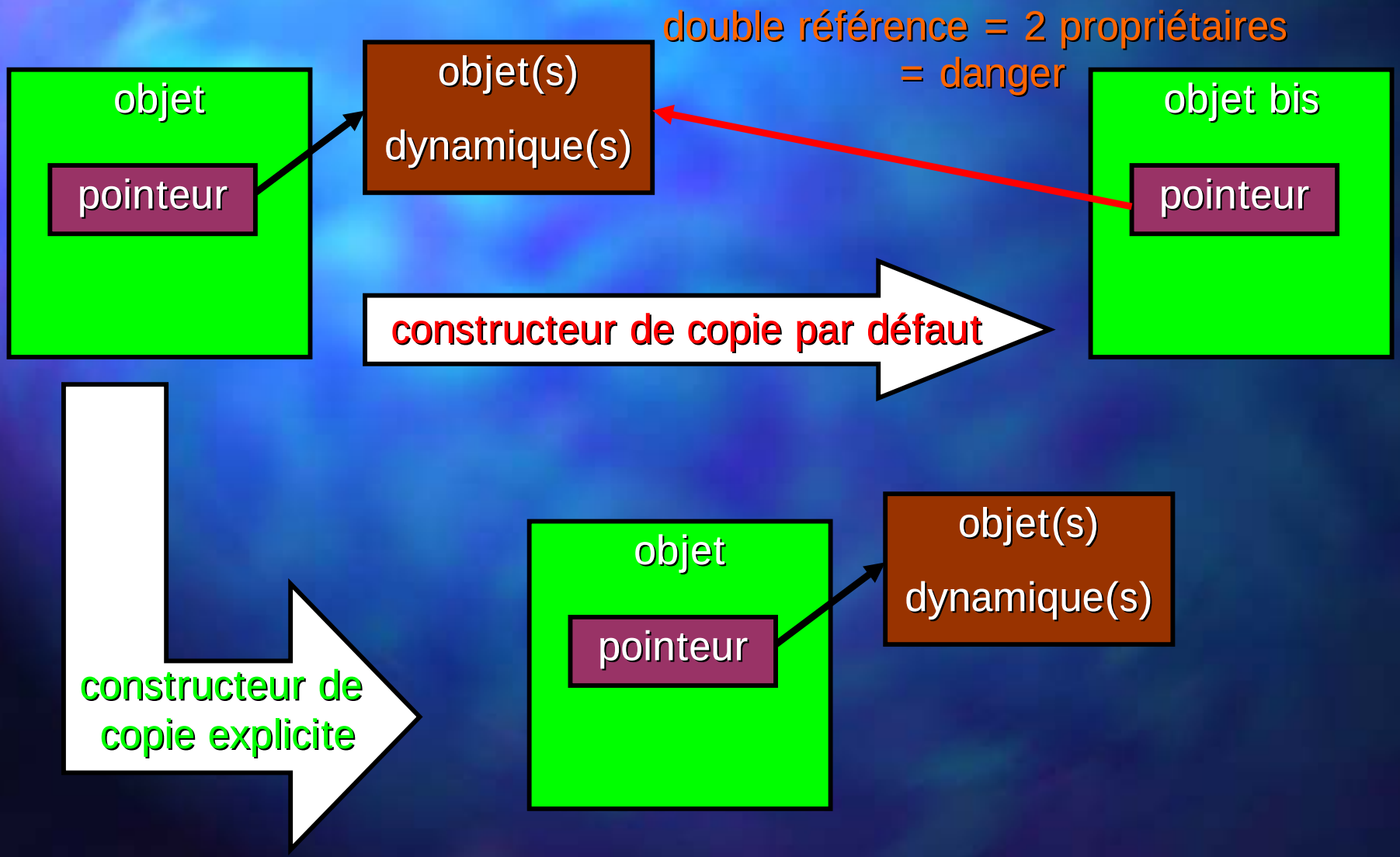
Gestion des ressources système en C++

- Règles "de bonne gestion" des ressources :
  - tout pointeur / handle vers la ressource sera l'attribut d'un objet qui sera son **unique** propriétaire
  - la valeur nulle de l'attribut signifie que la ressource n'est pas allouée actuellement
  - on alloue la ressource dans le constructeur ou dans les méthodes de l'objet si l'attribut est nul
  - on la libère dans des méthodes ou dans le destructeur si l'attribut n'est pas nul, puis on le remet à une valeur nulle
  - l'attribut est encapsulé (toute modification externe est interdite)
  - on peut procéder à des transferts "conservateurs"

Constructeur copie et opérateur d'attribution

- Tout classe qui manipule des données dynamiques ou ressources système doit avoir :
  - des constructeurs qui initialisent les pointeurs ou les *handles* à zéro ou avec les données dynamiques ou la ressource en question
  - un constructeur copie et un opérateur d'attribution explicites qui gèrent correctement les pointeurs et les *handles*
  - un destructeur explicite qui doit (en fonction des valeurs des pointeurs / *handles*) libérer les ressources allouées

Constructeur copie et opérateur d'attribution



Classe gestionnaire Exemple

- Modéliser une figure géométrique polygonale par une classe FigGeom
- Interface :
  - constructeur qui reçoit le nombre de sommets (positions aléatoires) et une couleur (32 bits non-signées)
  - tout autre constructeur/destructeur/opérateur nécessaire
  - possibilité de lire ou modifier le nombre de sommets
- Implémentation :
  - le nombre de sommets nbs, la couleur color
  - les coordonnées (de type float) des sommets (en nombre variable) seront mémorisés dans deux tableaux pointés par xs et ys

Objets globaux

- La construction des objets globaux se fait avant l'exécution de la première ligne de programme !
- Entre plusieurs objets globaux, l'ordre de leur construction n'est pas forcement définie
- Le constructeur d'un objet global peut contenir une quantité importante de code, voir tout le programme (le cas d'une application MFC).
- La destruction des objets globaux se fait après la dernière ligne du programme

Méthodes et attributs statiques

- Les méthodes et les attributs statiques existent en dehors des instances de classes
- Une méthode statique peut être appelée sans instance : `NomClasse::NomMethode(params)`
- Une méthode statique ne peut pas avoir accès aux attributs et méthodes non-statiques (car elle n'a pas de *this* !)
- Une méthode non-statique a accès aux attributs et méthodes statiques.

Méthodes et attributs statiques - Exemple

- Un attribut statique est partagé par toutes les instances d'une classe, il peut servir comme canal de communication entre les objets
- Faire en sorte que la classe FigGeom puisse fournir à tout instant le nombre d'instances crées ou existantes (tout type de variables)
- Rendre impossible toute altération de cette information par quelqu'un de l'extérieur de la classe

Méthodes et attributs statiques - Exemple

.h/.cpp

```
class FigGeom
{
   static int nbfigs;
   ...
public:
   ...
   static int GetNbFigs() {return nbfigs;}
};
```

*.cpp

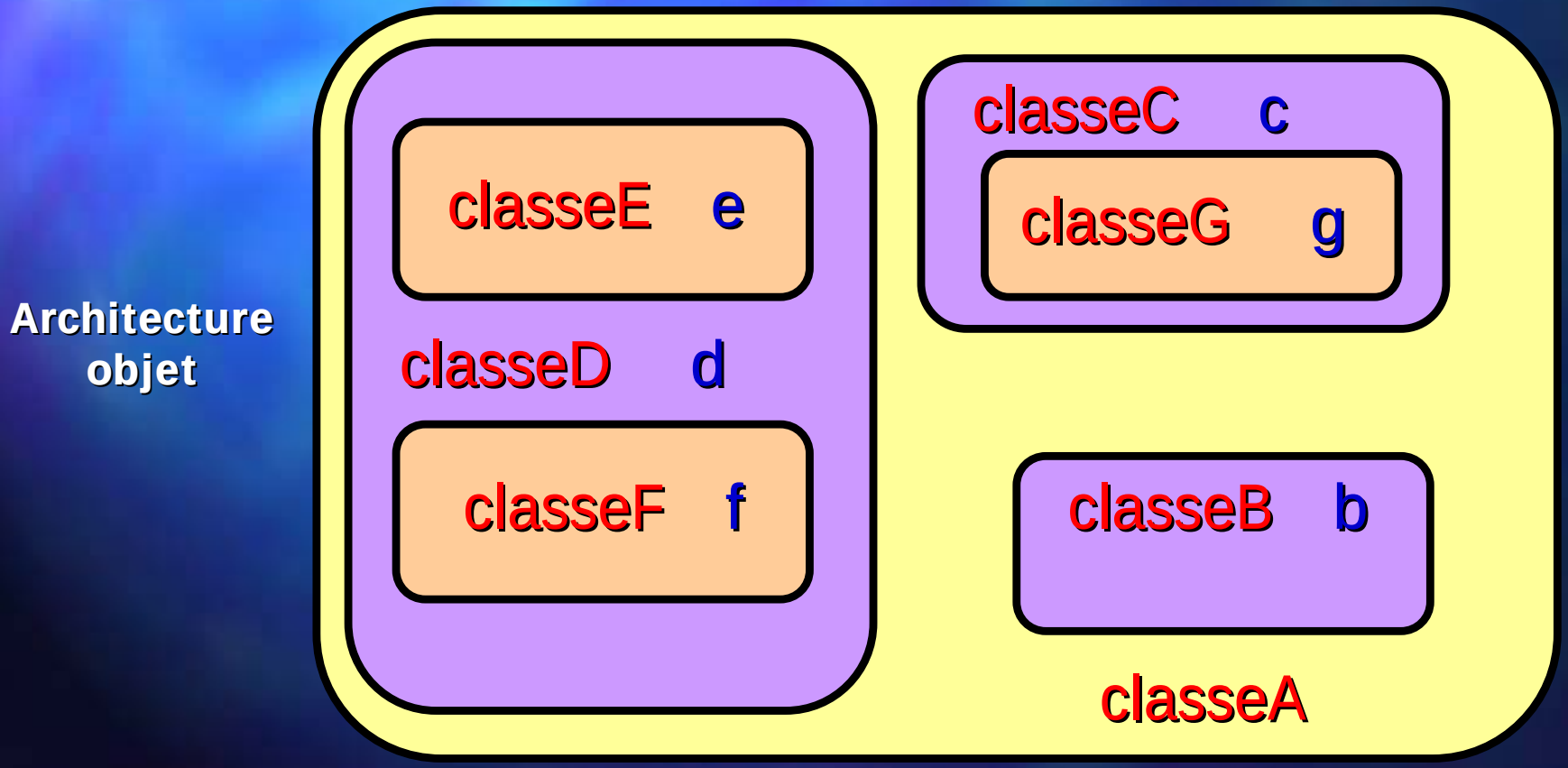
```
int FigGeom::nbfigs=0;
FigGeom::FigGeom(...) {nbfigs++; ...}
Figgeom::~FigGeom() {... nbfigs--;}
```

Objets complexes Construction

- Objet complexe : un objet qui a au moins 1 sous-objet obtenu par agrégation hybride (déclaré avec class ou struct)
- Sous-objet : un objet déclaré comme attribut ou obtenu par héritage
- Avant d'exécuter le constructeur de l'objet, on construit tout sous-objet qui le compose:
  - les objets attributs (déclarés dans la liste)
  - les objets héritées (éventuellement)
- A son tour, chaque sous-objet est construit de la même manière

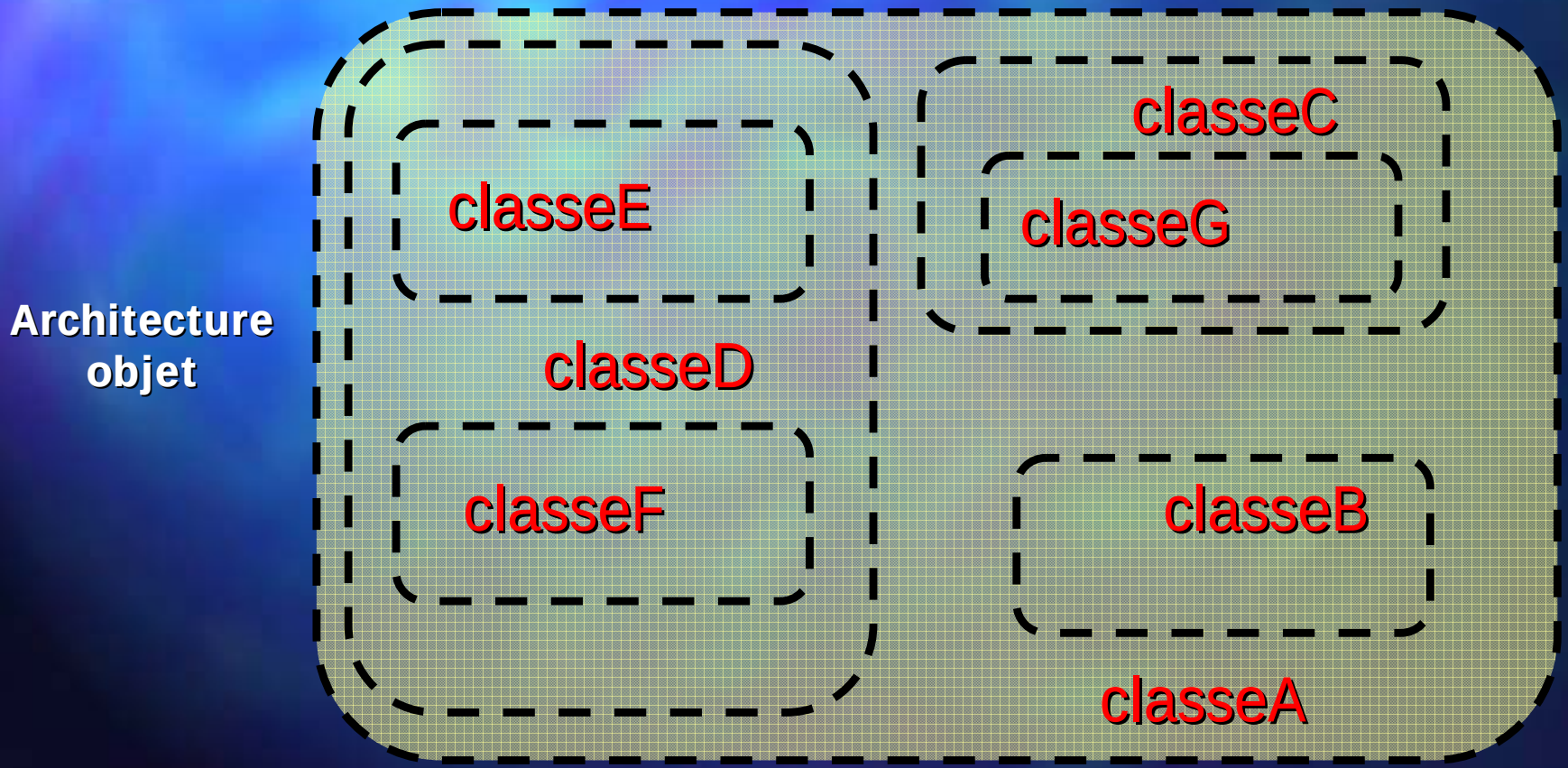
Objets complexes
Exemple de construction

Comment se passe la construction d'un objet complexe (qui contient d'autres sous-objets) ?



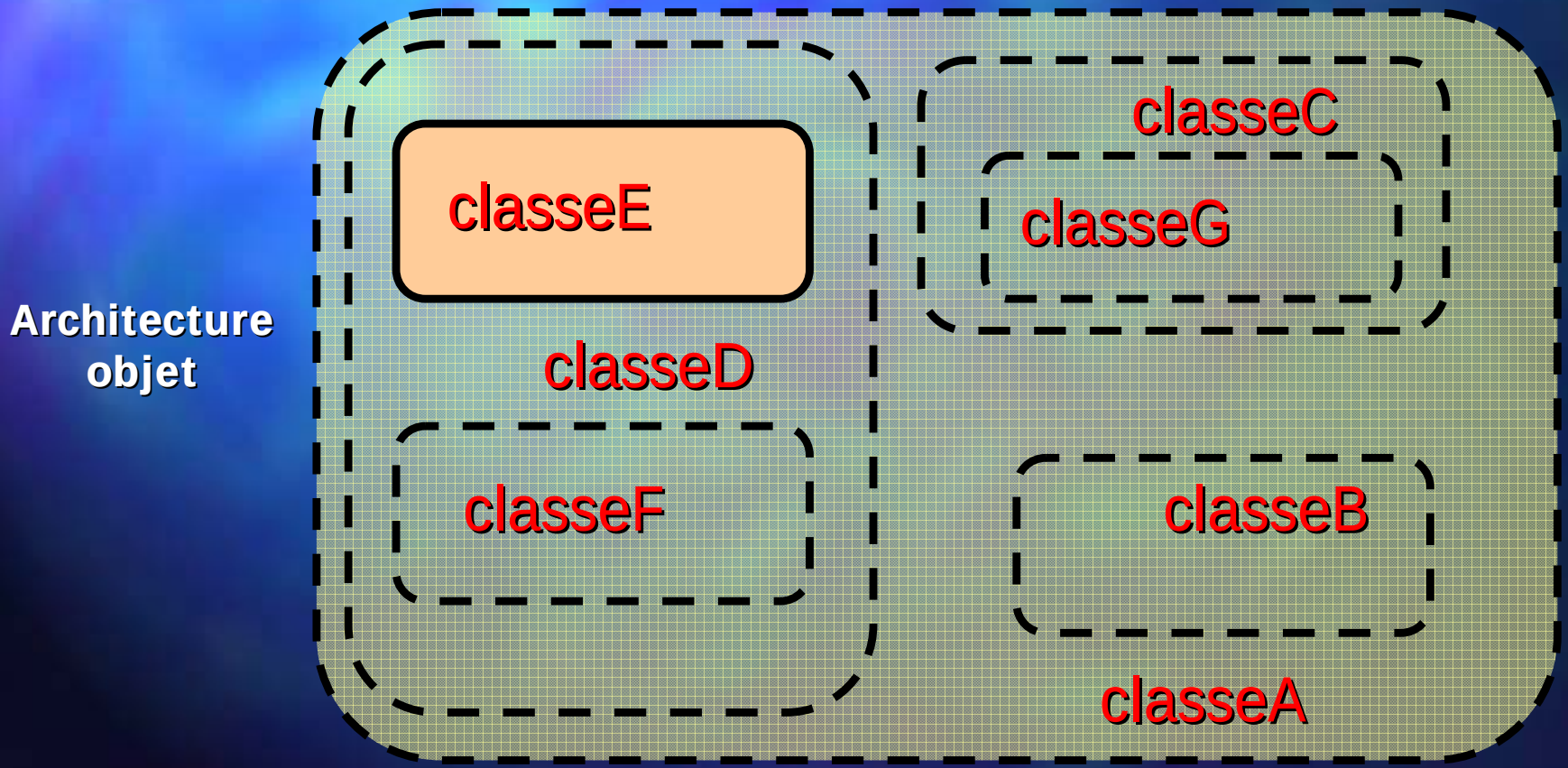
Objets complexes
Exemple de construction

Allocation de mémoire pour tout l'objet



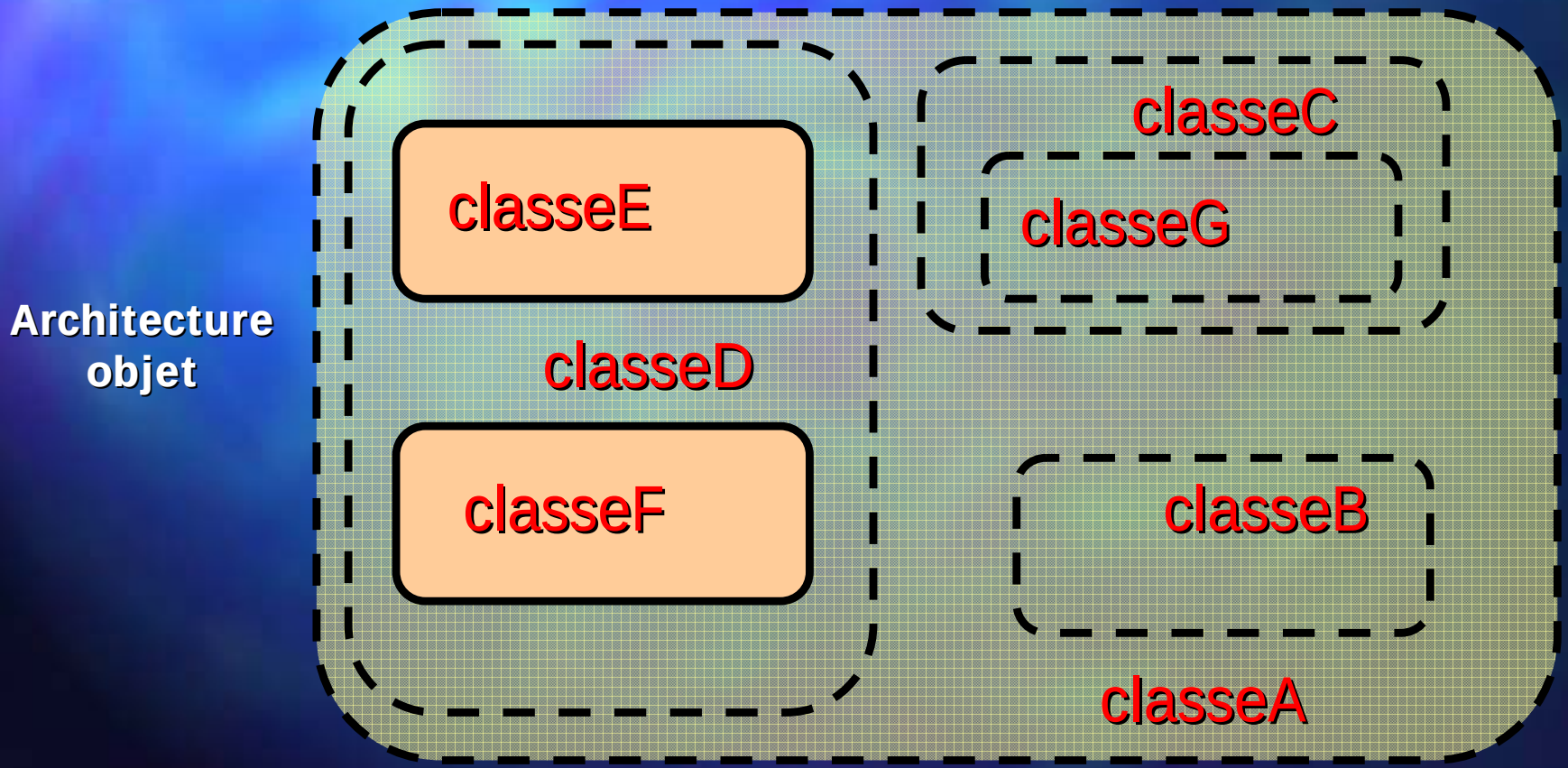
Objets complexes
Exemple de construction

Le compilateur commence par l'appel du constructeur des objets les plus petites (de type natif)



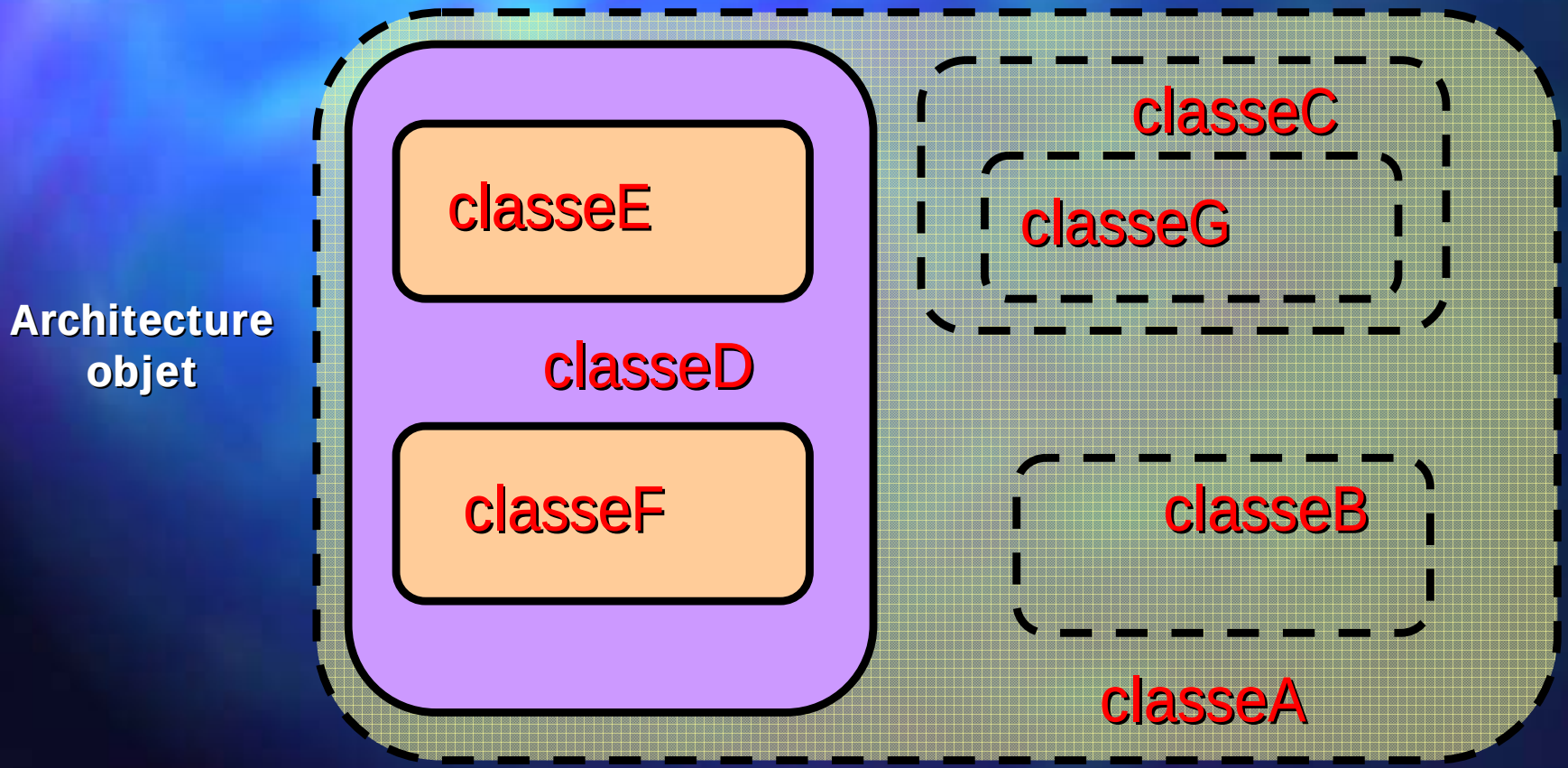
Objets complexes
Exemple de construction

Le compilateur commence par l'appel du constructeur des objets les plus petites (de type natif)



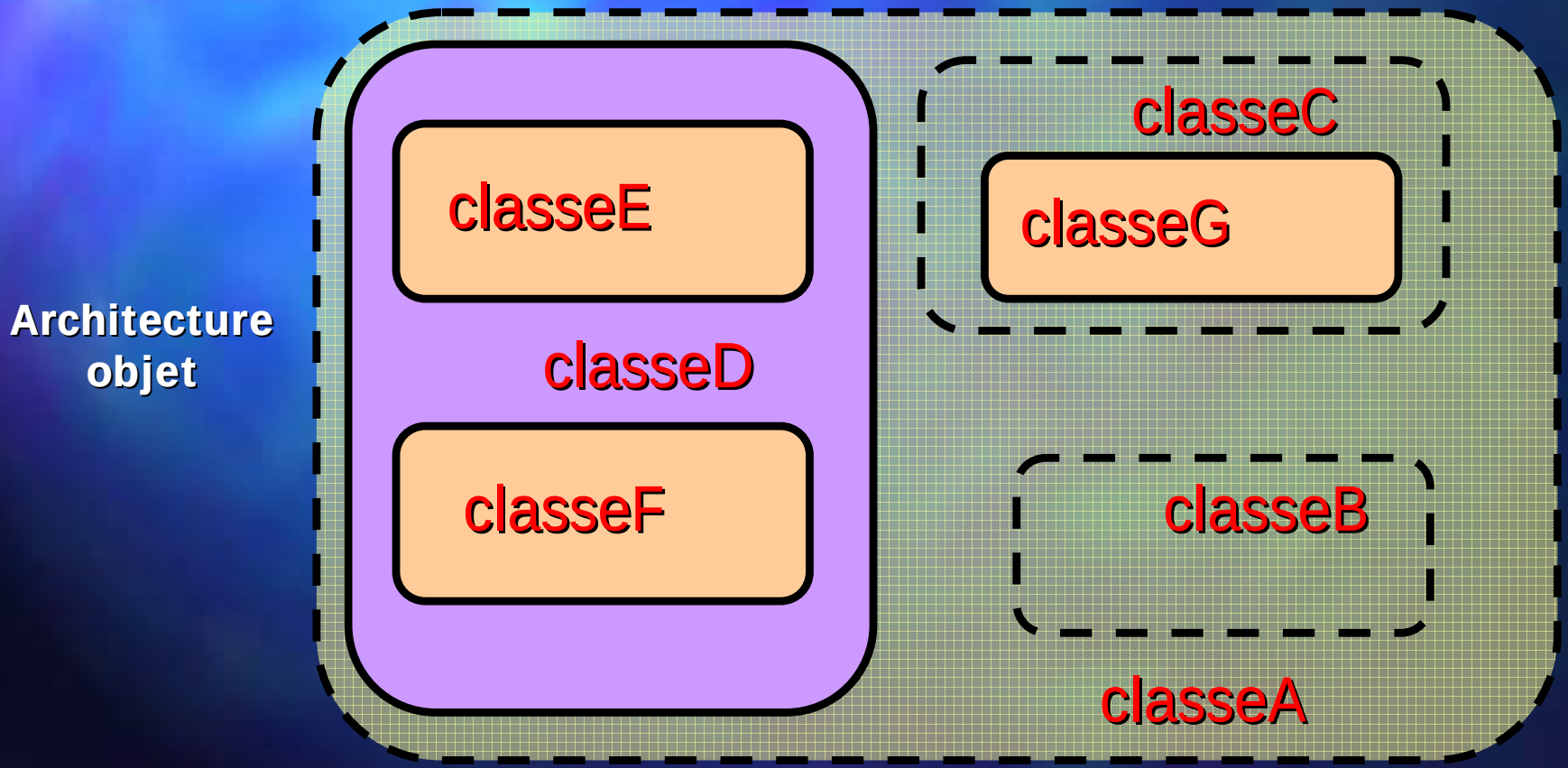
Objets complexes
Exemple de construction

Il continue par l'appel des constructeurs d'objets moyens



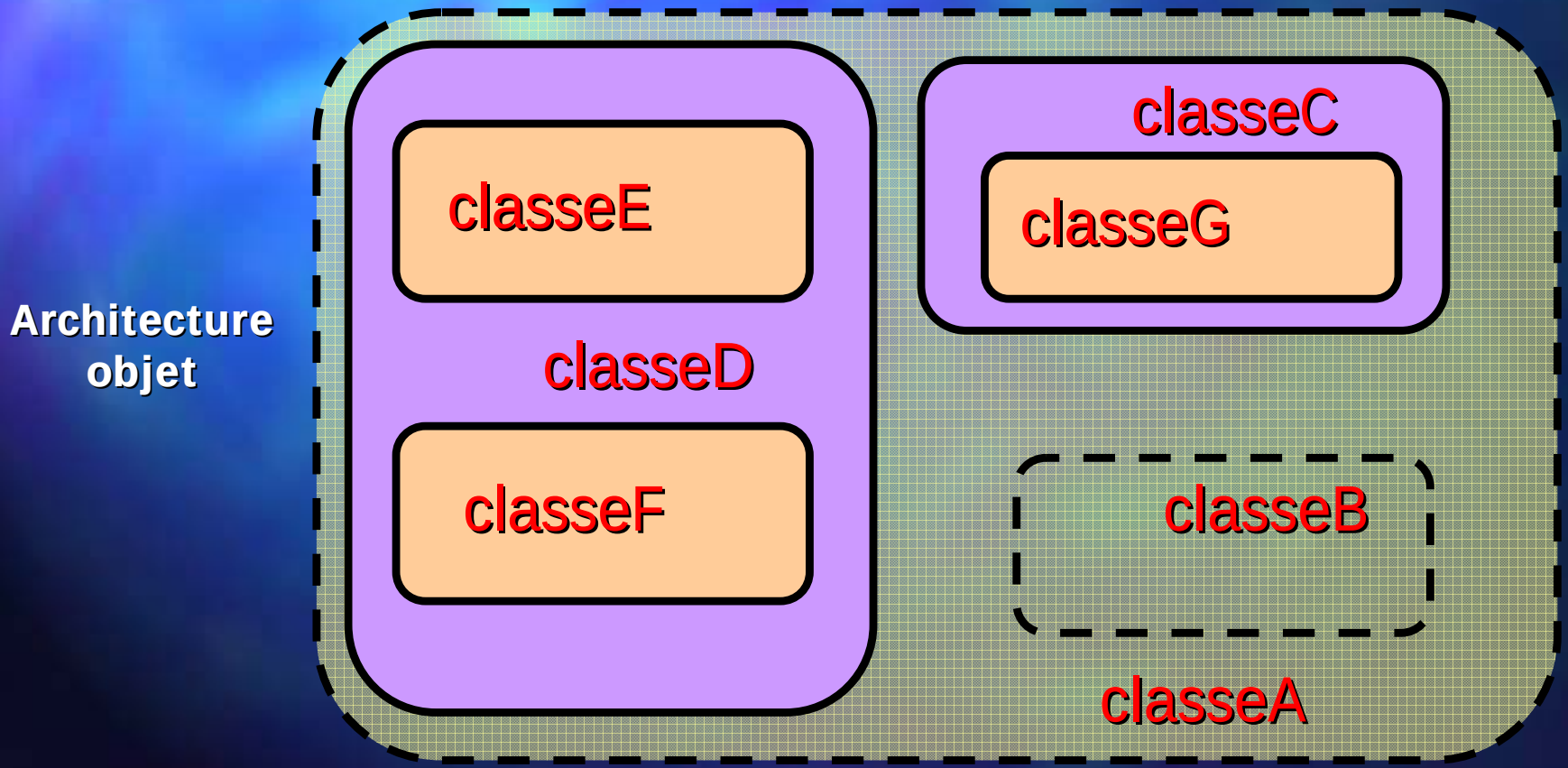
Objets complexes
Exemple de construction

Puis il poursuit de la même manière la construction d'un autre sous-objet avec ses sous-sous-objets



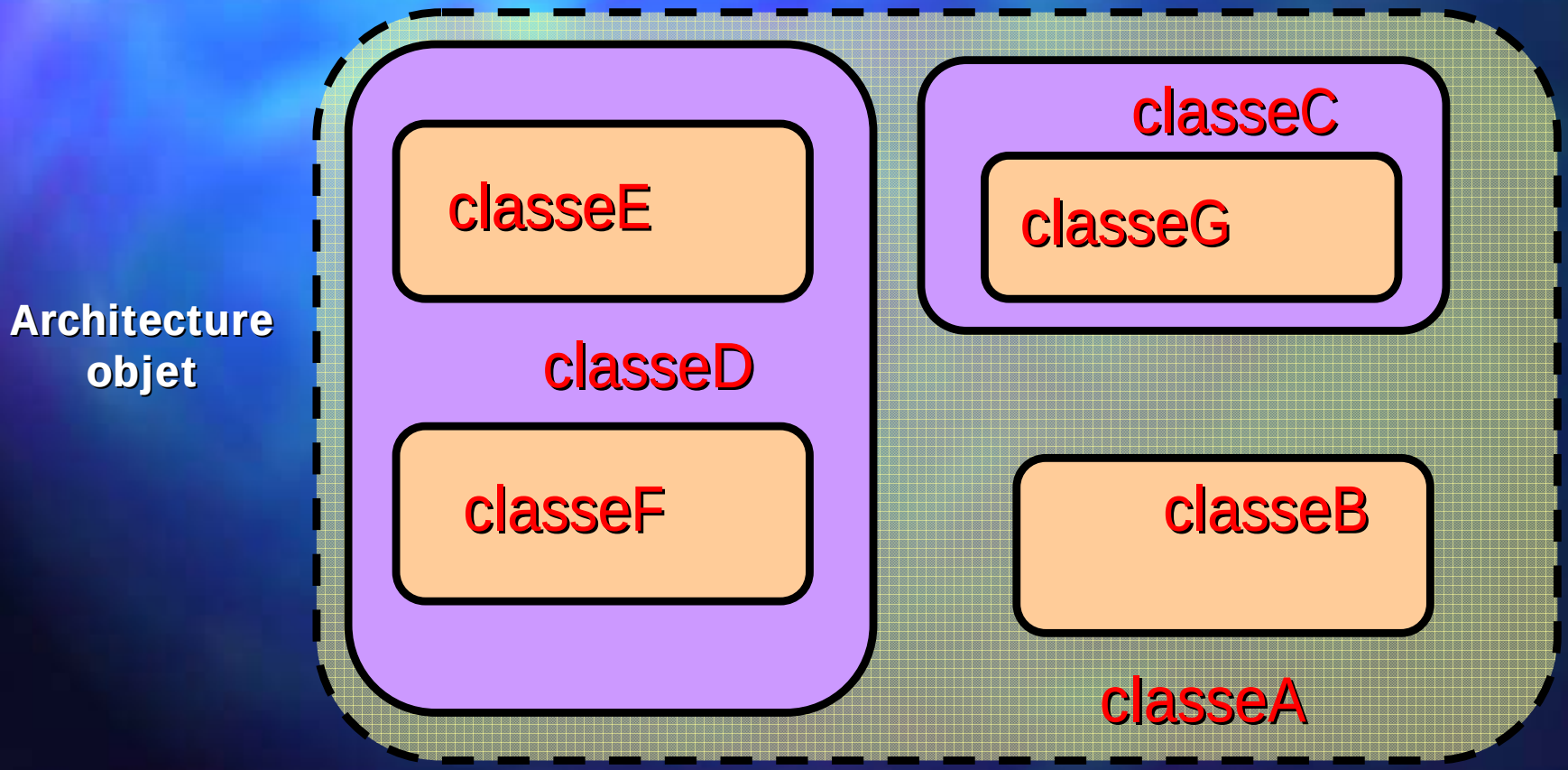
Objets complexes
Exemple de construction

… et il continue de la même manière la construction



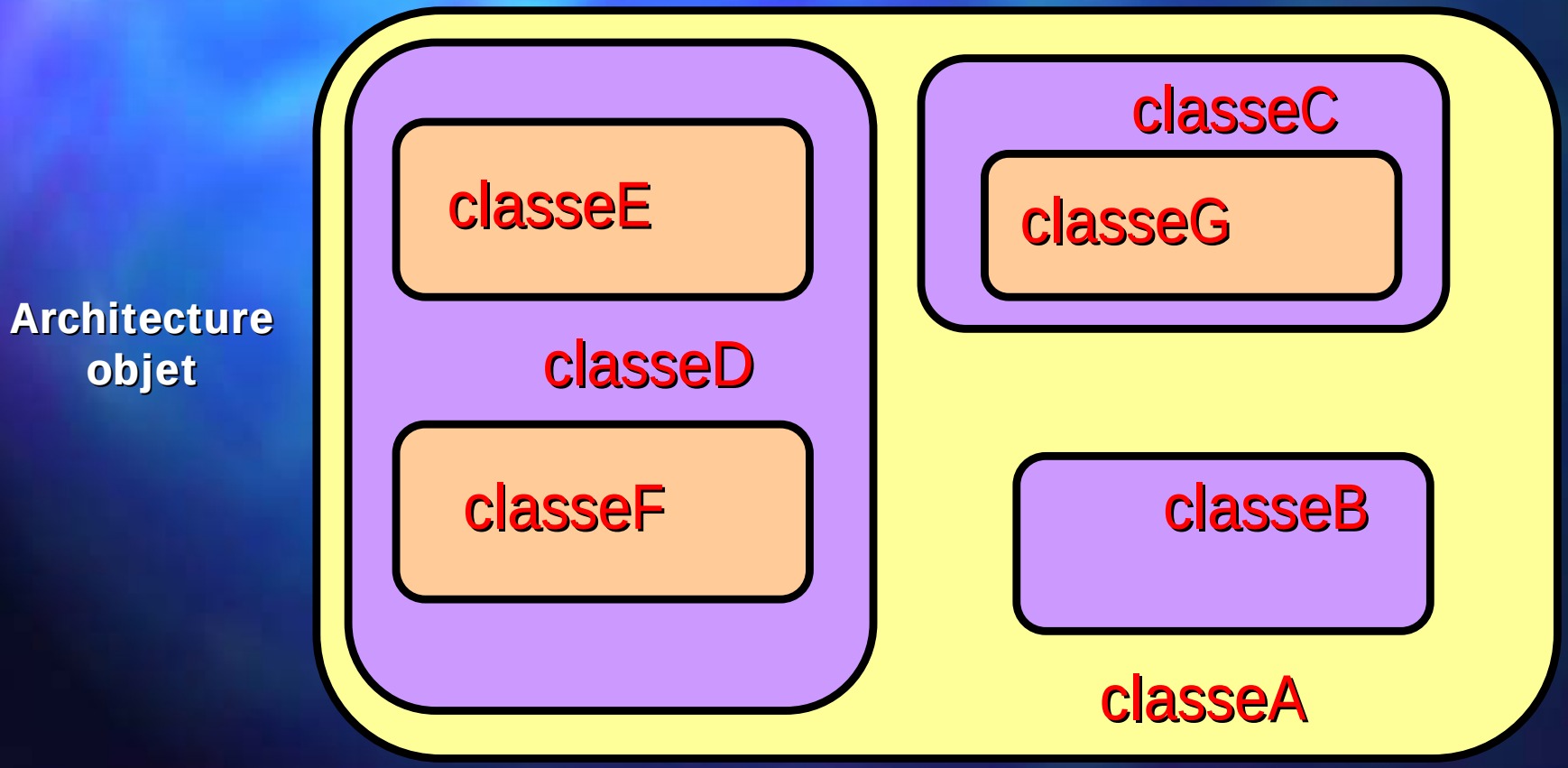
Objets complexes
Exemple de construction

… et il continue de la même manière la construction



Objets complexes
Exemple de construction

... en terminant par l'exécution du corps du constructeur de la classe classeA



Objets complexes
Exemple de construction

- Comment le compilateur sait quel constructeur appeler ?
  - soit l'appel au constructeur du sous-objet figure entre le prototype et le corps du constructeur
  - soit il n'y a pas d'appel explicite et alors le compilateur appelle pour le sous-objet en question le constructeur sans paramètres. S'il ne trouve pas de constructeur sans paramètres il affiche une erreur de compilation.
  - les mêmes règles s'appliquent d'une manière itérative pour l'appel des constructeurs des sous-sous-objets …
- Ne jamais préjuger sur l'ordre d'appel des constructeurs des sous-objets.

Objets complexes
Constructeurs

```
class ClasseA
{
  ClasseB b;
  ClasseC c;
  ClasseD d;
public:
  ClasseA(parametres): b(parametres), d(parametres)
  { // corps du constructeur
  }
};
```

appel implicite du constructeur

liste d'appels explicites de constructeurs

- Appel explicite obligatoire : si le sous-objet n'a pas de constructeur sans paramètres.
- Appel explicite optionnel : si le sous-objet a un constructeur sans paramètres
- Appel explicite déconseillé / interdit : d'un constructeur d'un sous-sous-objet ou grand-parent

Objets complexes Constructeurs / Copie

- Le même principe d'appel itératif des constructeurs (de l'objet le plus simple jusqu'à l'objet le plus complexe) s'applique pour :
  - le constructeur de copie
  - l'opérateur de copie (d'attribution)
- Mais il n'y a plus d'ambiguïté possible pour les appels donc le compilateur le fait d'une manière implicite
- Attention : un seul ctor ou opérateur de copie inaccessible (privé ou autre) peut empêcher la copie du "grand" objet

Destructeurs

- A la destruction l'ordre est inversé
- Après l'exécution du destructeur de l'objet, le compilateur appelle les destructeurs de tous les sous-objets qui le compose:
  - les objets attributs (déclarés dans la liste)
  - les objets héritées (éventuellement)
- Il n'y a pas d'ambiguïté car les destructeurs sont uniques

Conversions implicites

- Qui le provoque ? Vers quel type (classe) ?
  - une attribution ou une opération suivie d'attribution
    - vers le type de l'opérande gauche (conteneur)
  - un champ d'instruction (for, while, if etc.)
    - vers le type bool
  - un appel de fonction / méthode / constructeur
    - vers le type du paramètre déclaré dans le prototype
  - une instruction de retour
    - vers le type de retour de la fonction / méthode
  - un opérateur non défini pour le type en question
    - vers le type pour lequel l'opérateur est défini

Conversions implicites

- Quelles règles applique le compilateur pour convertir le type class1 vers le type class2 ?
  - règles de conversion implicite des types natifs
  - constructeurs de class2 avec 1 paramètre class1
    - on peut l'interdire en mettant le qualificatif **explicite** devant la déclaration du constructeur
  - opérateurs surchargés de *typecast* de class1 vers class2.
  - extraction de l'ancêtre si class2 est un ancêtre de class1.
- S'il n'y a pas de règle : erreur de conversion
- S'il y a une seule règle : conversion automatique
- S'il y a plusieurs règles : erreur d'ambiguïté

Flots d'entrée / sortie

- C++ prévoit des entrées / sorties à l'aide des objets, appelés flots, vers trois média différents:
  - la console / le clavier (standard)
    - déclarations dans <iostream.h>
  - les fichiers
    - déclarations dans <fstream.h>
  - la mémoire (buffers)
    - déclarations dans <strstream.h>
- Pour toute classe type flot de données:
  - l'entrée est faite par l'opérateur polymorphe >>
  - la sortie est faite par l'opérateur polymorphe <<

Flots d'entrée / sortie

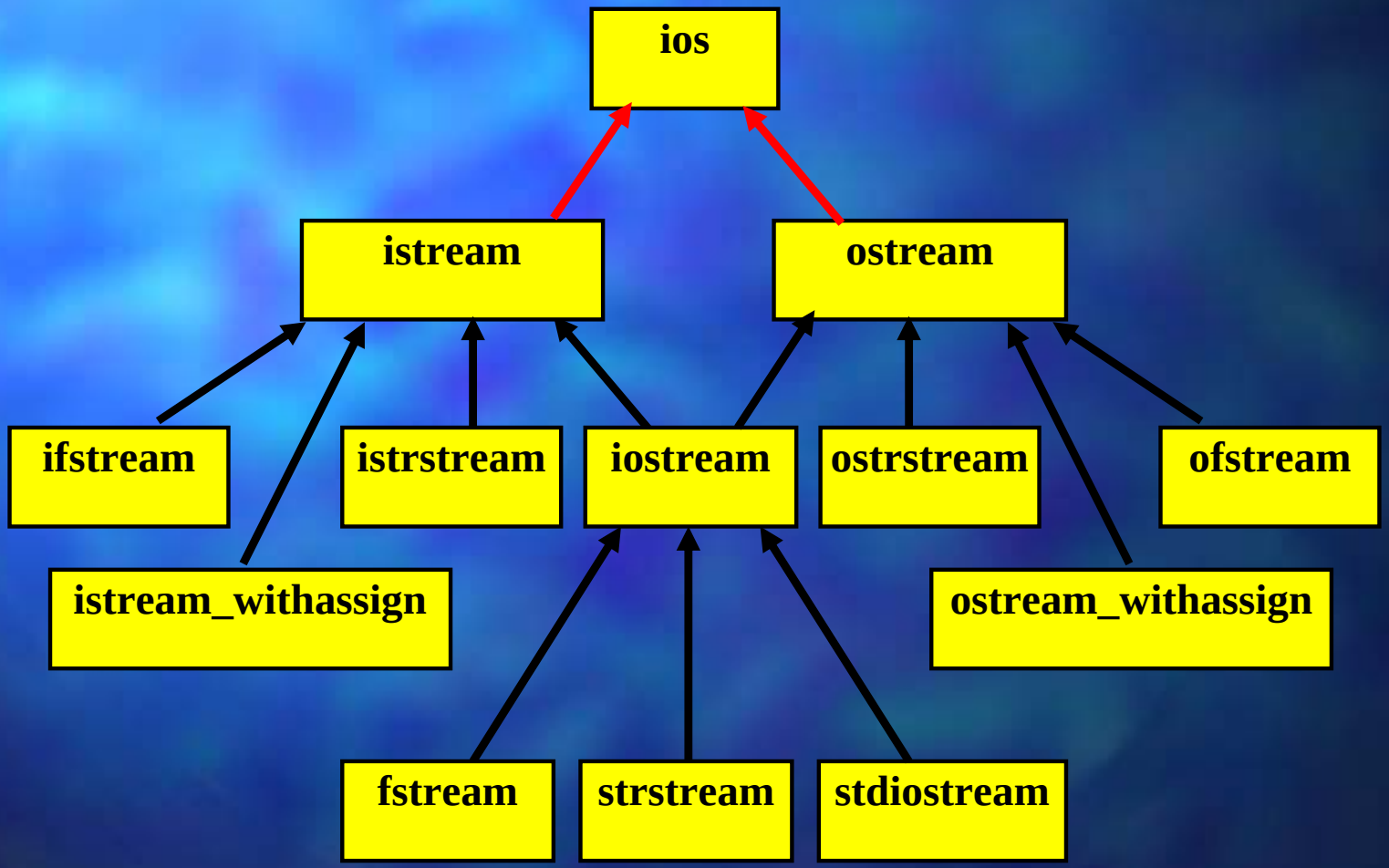
- Version obsolète (VS 6.0): *IOStream Library*
  - accessible par inclusion :

```
#include <iostream.h>
void main()
{ cout<<"Coucou !\n"; }
```

- Version ANSI/ISO : *Standard Template Library*
  - accessible par inclusion :

```
#include <iostream>
using namespace std;
void main()
{ cout<<"Coucou !\n"; }
```

Architecture de classes de *IOStream*



Flots d'entrée / sortie

- Les trois familles partagent une base virtuelle unique appelée ios
- Quatre flots standard dans <iostream.h>:
  - cout pour les sorties vers la console
    - instance globale de ostream_withassign
  - cin pour les entrées du clavier
    - instance globale de istream_withassign
  - cerr pour les messages d'erreur (=cout)
    - instance globale de ostream_withassign
  - clog pour les messages d'activité
    - instance globale de ostream_withassign

Flots d'entrée / sortie

- Les opérateurs << et >> sont surchargés pour tous les types natifs (char, short, int, long etc.)

```
istream& istream::operator>>(type_de_base& variable)
   {/* ... */}
ostream& ostream::operator<<(type_de_base variable)
   {/* ... */}
```

- Manipulateurs :
  - ils s'insèrent comme les variables et modifient le comportement des flux
  - pour les manipulateurs avec paramètre il faut inclure iomanip.h ou iomanip

Flots d'entrée / sortie

Manipulateur	Sens	Compatibilité	Signification
`oct`	E/S	IOStream, STL	affichage / lecture octale
`dec`	E/S	IOStream, STL	affichage / lecture décimale
`hex`	E/S	IOStream, STL	affichage / lecture hexadécimale
`flush`	S	IOStream, STL	vide le buffer de sortie
`endl`	S	IOStream, STL	insère un saut de ligne et vide le buffer
`ends`	S	IOStream, STL	insère un caractère nul (`'\0'`)
`setbase(int base)`	E/S	STL	change la base (seulement entre 8, 10 et 16)
`setprecision(int nb)`	E/S	IOStream, STL	définit le nombre de chiffres après la virgule
`setw(int nbcar)`	E/S	IOStream, STL	définit la taille du champ suivant
`setfill(int car)`	E/S	IOStream, STL	définit le caractère de remplissage (par défaut `' '`)
`setiosflags(long f)`	E/S	IOStream, STL	modifications avancées de toutes les options
`resetiosflags(long f)`	E/S	IOStream, STL	restauration par défaut de toutes les options

Flags (fmtflags) pour [re]setiosflags()

- Plongés dans le sous-espace **ios_base** :

fixed	virgule fixe
scientific	format scientifique
left	alignement à gauche
right	alignement à droite
showpoint	affichage virgule
showpos	affichage du "+" si positif
dec / hex / oct	base 10, 16, 8
showbase	affichage préfixe base (0x ou 0)
boolalpha	affichage bool comme "true/false"

Pourquoi les flots d'E/S ?

- Plus sûr vis-à-vis des fautes de codage :

```
void main()
{
  int i1, i2;
  double d1;
  // double erreur a l'exécution !!!
  scanf("%d%d%lf", &i1, &d1, i2);
  // appel automatique du bon opérateur
  cin>>i1>>d1>>i2;
}
```

- Il ne faut jamais mélanger les flots C (stdio.h) avec les flots C++ (iostream) !

Extensions aux nouvelles classes

- Extension à n'importe quel objet (surcharge par fonction car opérande droit) :

```
class CCercle
{
  float x, y, r;
  friend
    ostream& operator<<(ostream& os, const CCercle& c);
};

ostream& operator<<(ostream& os, const CCercle& c)
{
  return os<<"Cercle de rayon "<<c.r
    <<" et centre (" << c.x <<","<< c.y <<")\n";
}
```

Extensions aux nouvelles classes

- Si l'on veut éviter la déclaration d'amitié :

```
class CCercle
{
  float x, y, r;
public:
  ostream& Affiche(ostream& os) const
  {
    return os<<"Cercle de rayon "<< r
    <<" et centre (" << x <<","<< y <<")\n";
  }
};

ostream& operator<<(ostream& os, const CCercle& c)
{ return c.Affiche(os); }
```

Utilisation des flots vers des fichiers

- Les flots "fichiers" fstream sont directement compatibles avec les ostream et istream. Il faut juste créer un objet de type fstream.

```
fstream( const char* szName,
         int nMode,
         int nProt = filebuf::openprot );
```

nom du fichier

modes d'ouverture:

- **ios::in ou ios::out**
- **rien ou ios::bin**
- **ios::trunc ou ios::app**
- **ios::noreplace,**
- **ios::nocreate**

protection du fichier:

- **filebuf::sh_none**
- **filebuf::sh_read**
- **filebuf::sh_write**

Utilisation des flots vers des fichiers

- Il n'y a pas de différence à l'utilisation entre cout et un fichier-*stream* ouvert en mode "sortie" :

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

bibliothèque classique IOStream (obsolète sauf VS 6.0)

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

bibliothèque STL

```
void main()
{
  CCercle c(1.0f, 3.0f, 4.0f);
  cout << c;
  fstream fs("cercle.txt",ios::out);
  fs << c;
}
```

Tableau d'équivalences

Flux C	Flux C++
`#include <stdio.h>`	`#include <iostream>`
`stdin`	`cin`
`stdout`	`cout`
`printf("Text %d\n",i)`	`cout<<"Text "<<i<<endl`
`fprintf(pf,"Text %d\n",i)`	`os<<"Text "<<i<<endl`
`FILE* pf=fopen("...","wt")`	`fstream os("...",ios::out)`
`function(FILE* pf, ...)`	`function(ostream& os, ...)`

`FILE* operator<<(`
`  FILE* pf,`
`  const CClass& obj)`
	`ostream& operator<<(`
	`  ostream& os,`
	`  const CClass& obj)`

`fclose(pf); pf=0;`	`}`

Héritage

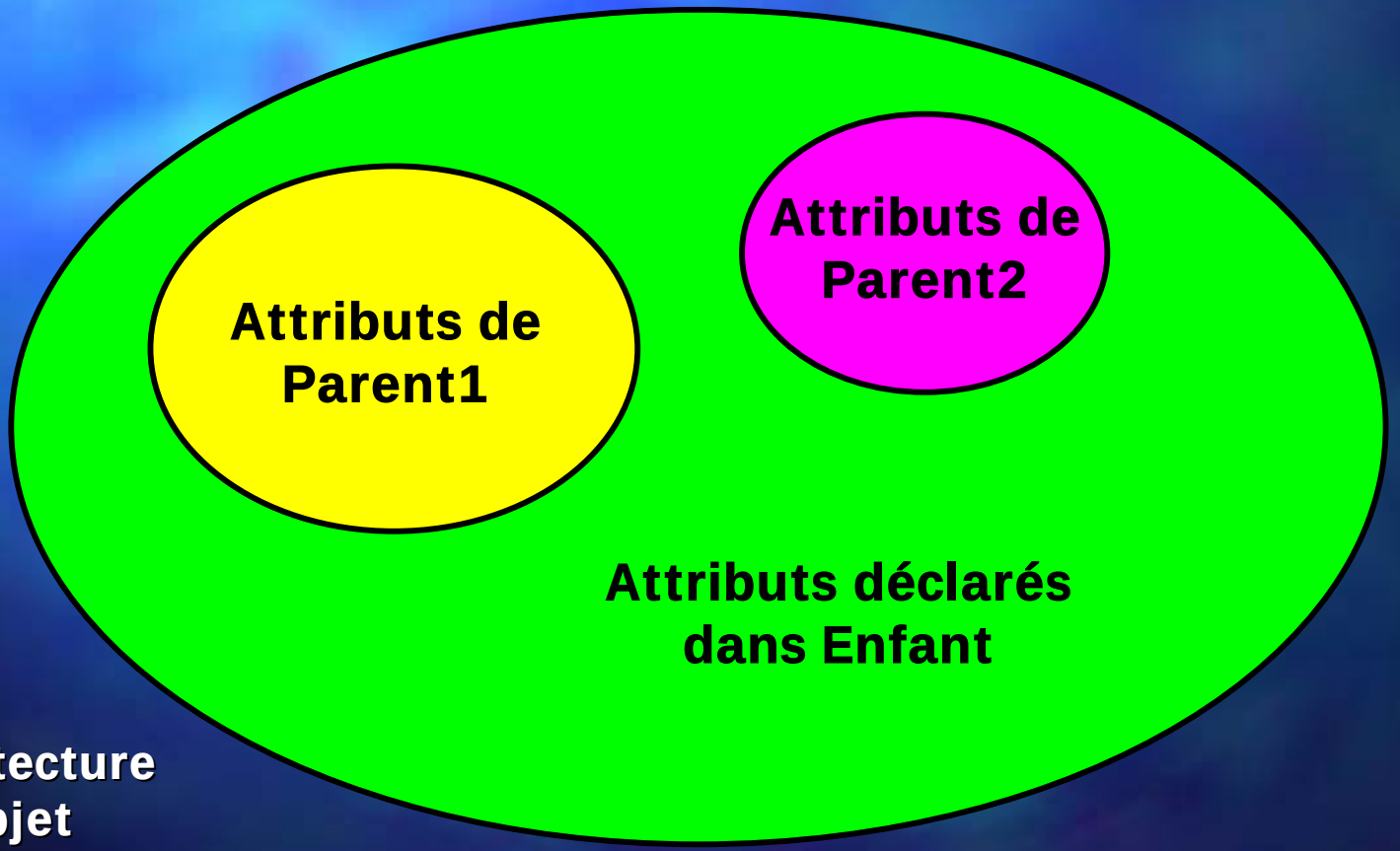
- La déclaration d'une classe peut comprendre une clause d'héritage d'une ou plusieurs autres classes :

```
class Enfant : public Parent1, protected Parent2
{
   ...
};
```

- La classe enfant hérite tous les membres des classes parents. La syntaxe d'accès de l'intérieur ou de l'extérieur reste la même
- Il est interdit d'hériter de soi-même

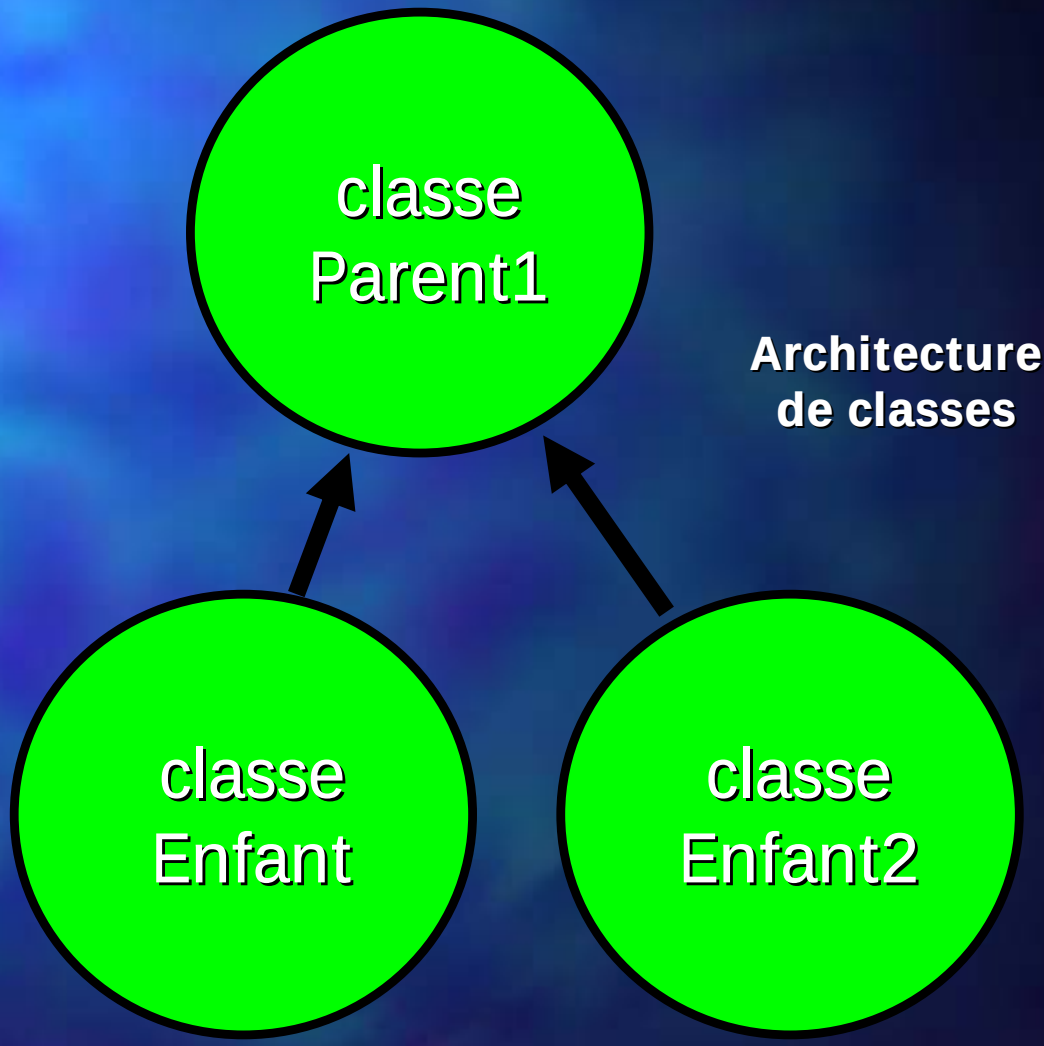
Héritage

Image en mémoire d'un objet de type Enfant et les sous-objets de type Parent1 et Parent2



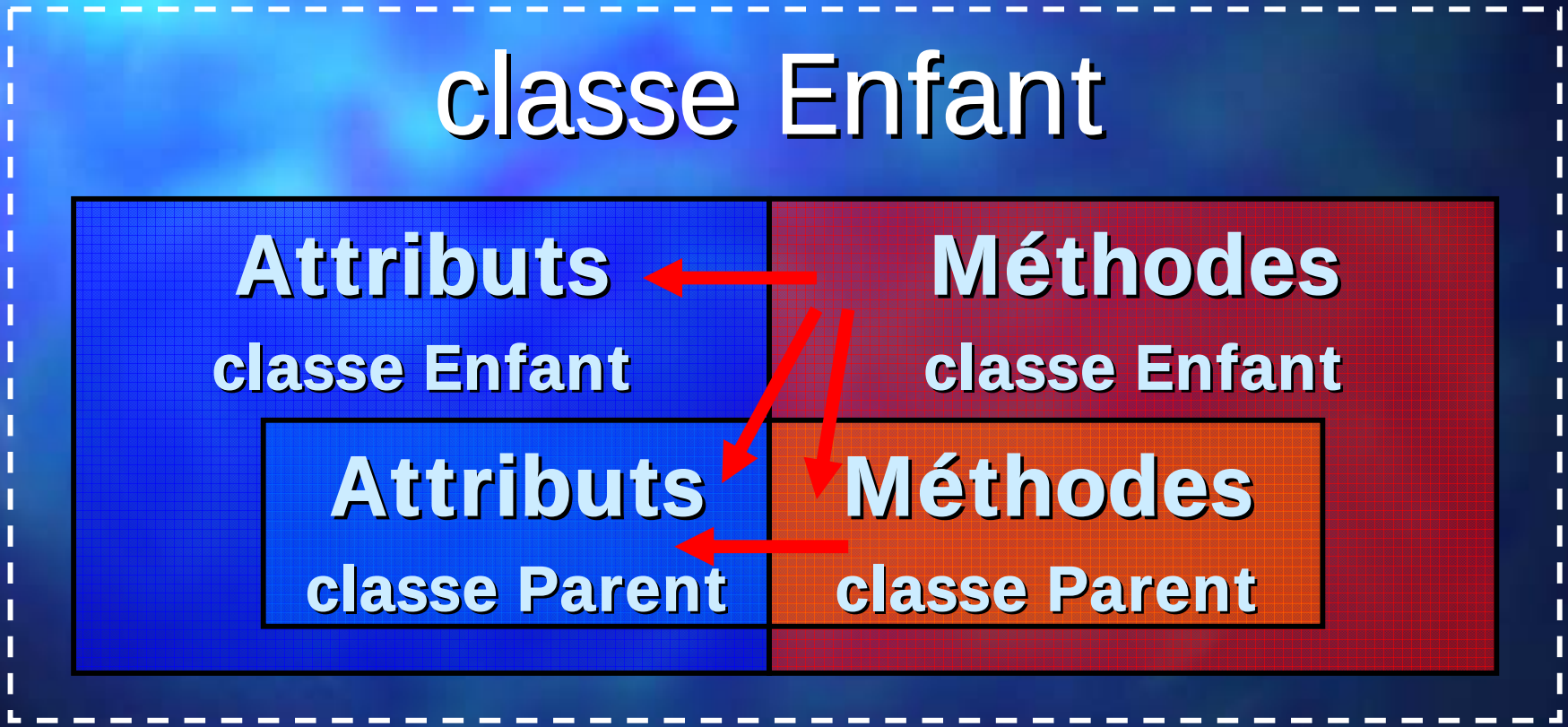
Héritage mono-parental

```
class Parent1
{ ... };

class Enfant :
   public Parent1
{ ... };

class Enfant2 :
   public Parent1
{ ... };
```



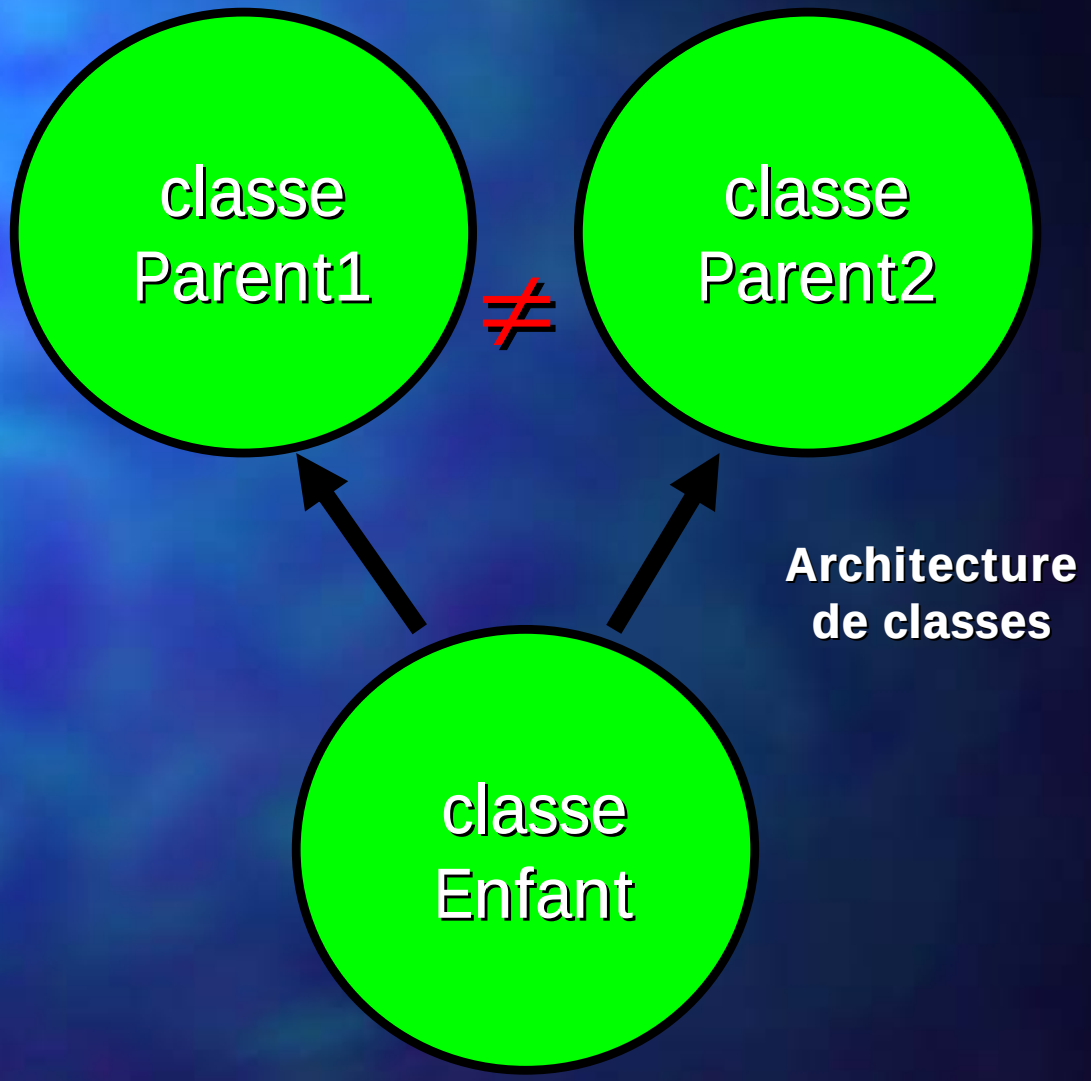
Architecture de classes

Héritage mono-parental

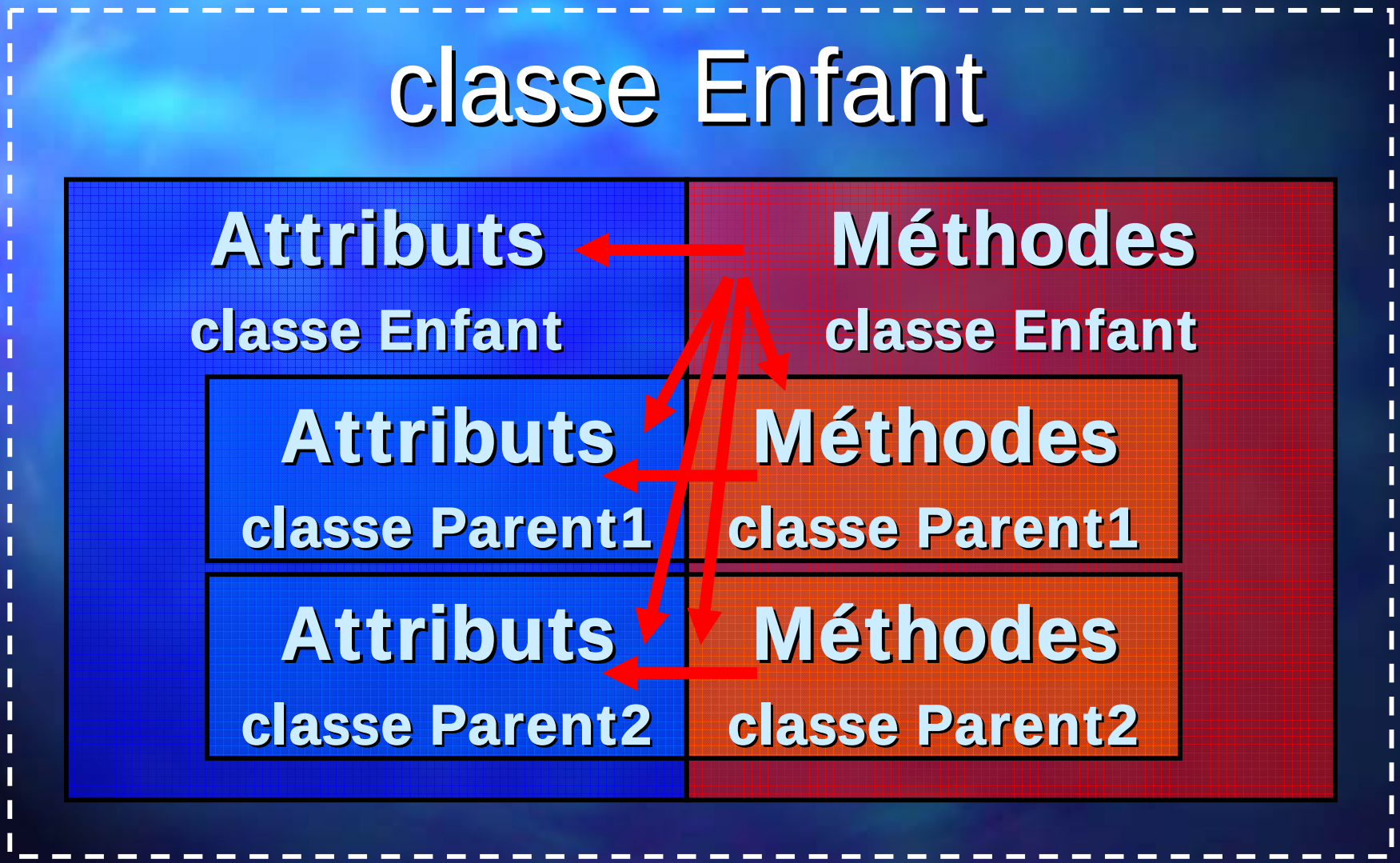


Héritage multi-parental

```
class Parent1
{ ... };

class Parent2
{ ... };

class Enfant :
   public Parent1,
   public Parent2
{ ... };
```

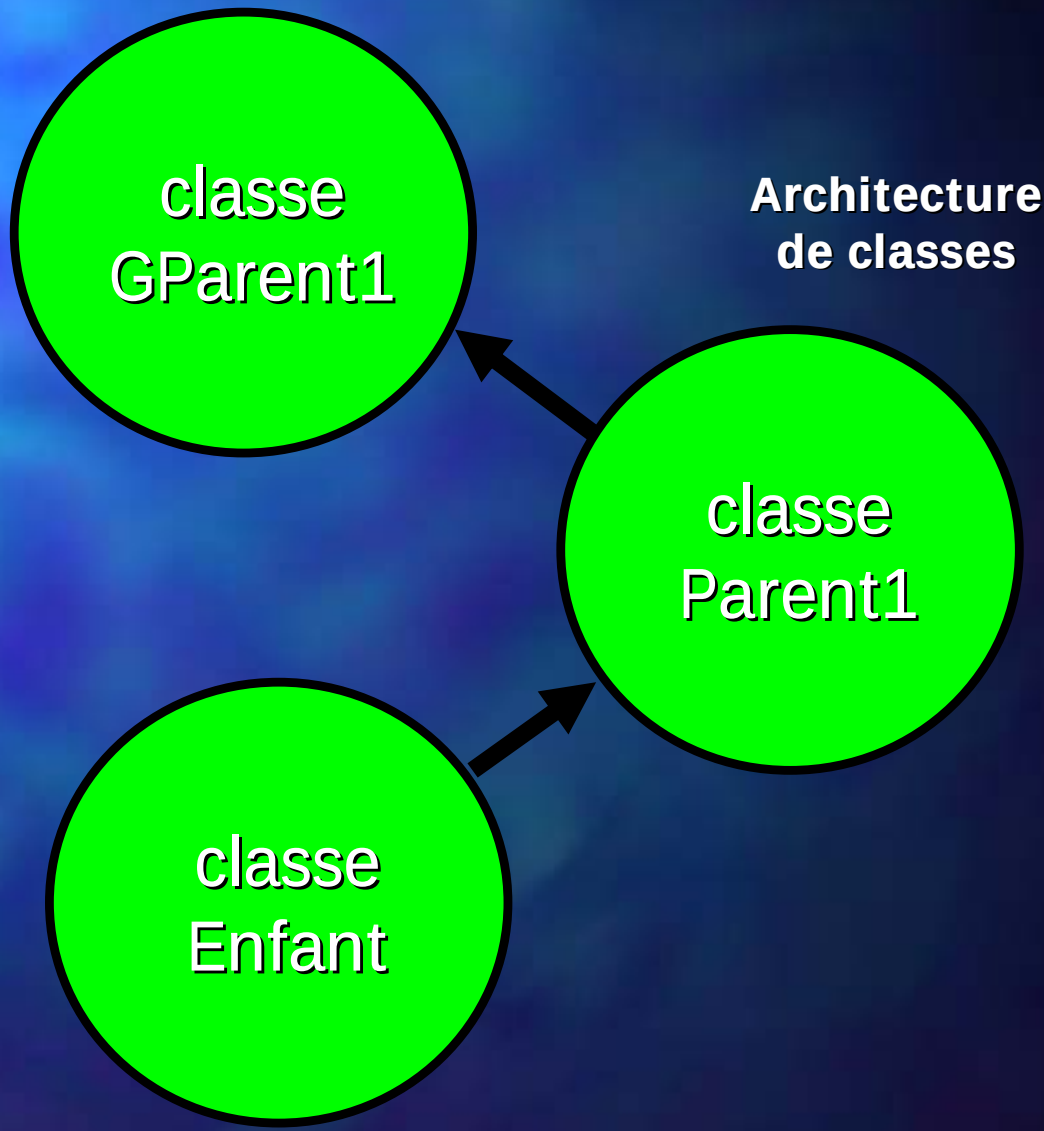


Héritage multi-parental

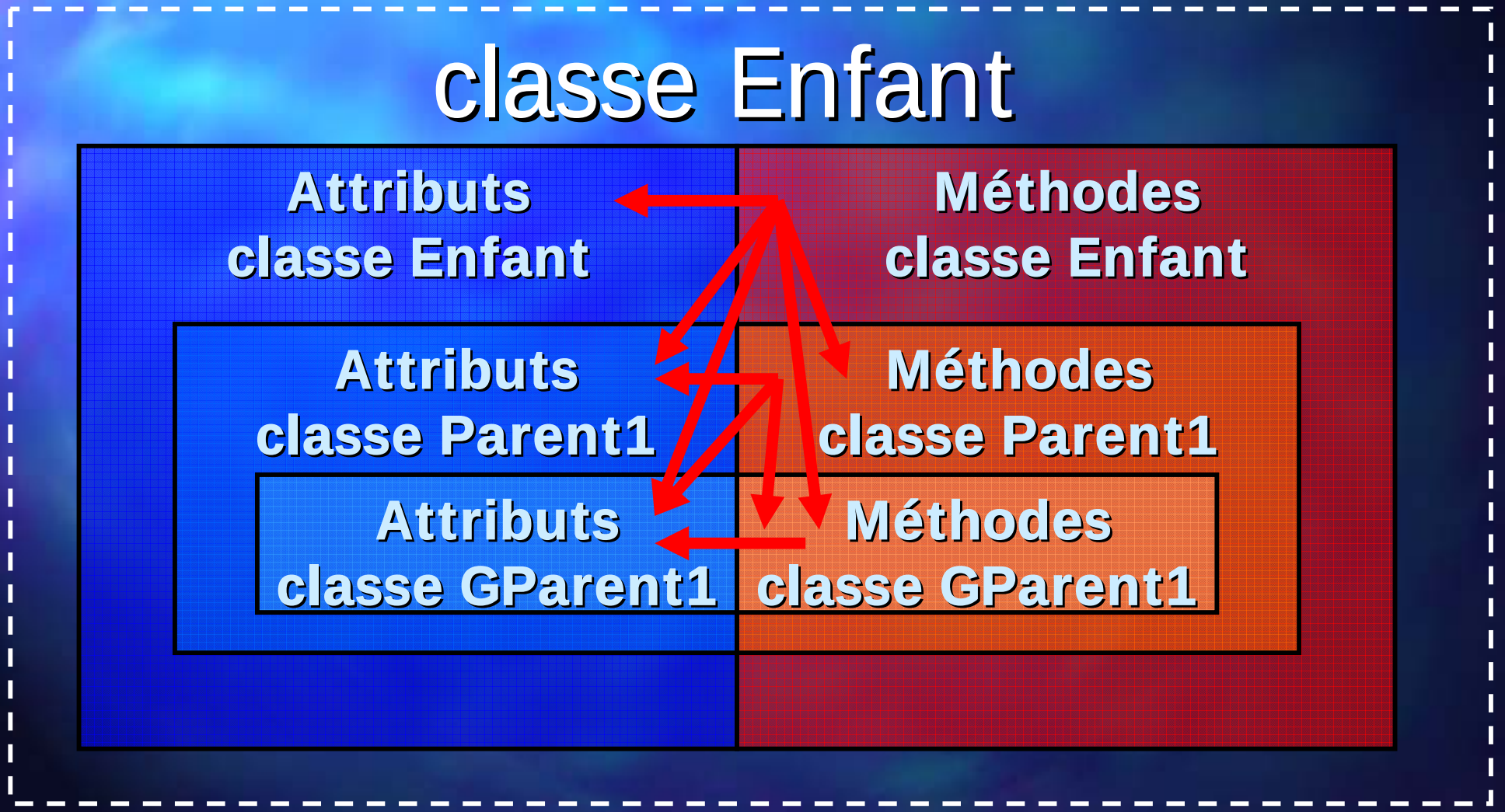


Héritage multi-parental

```
class GParent1
{ ... };

class Parent1 :
   public GParent1
{ ... };

class Enfant :
   public Parent1
{ ... };
```

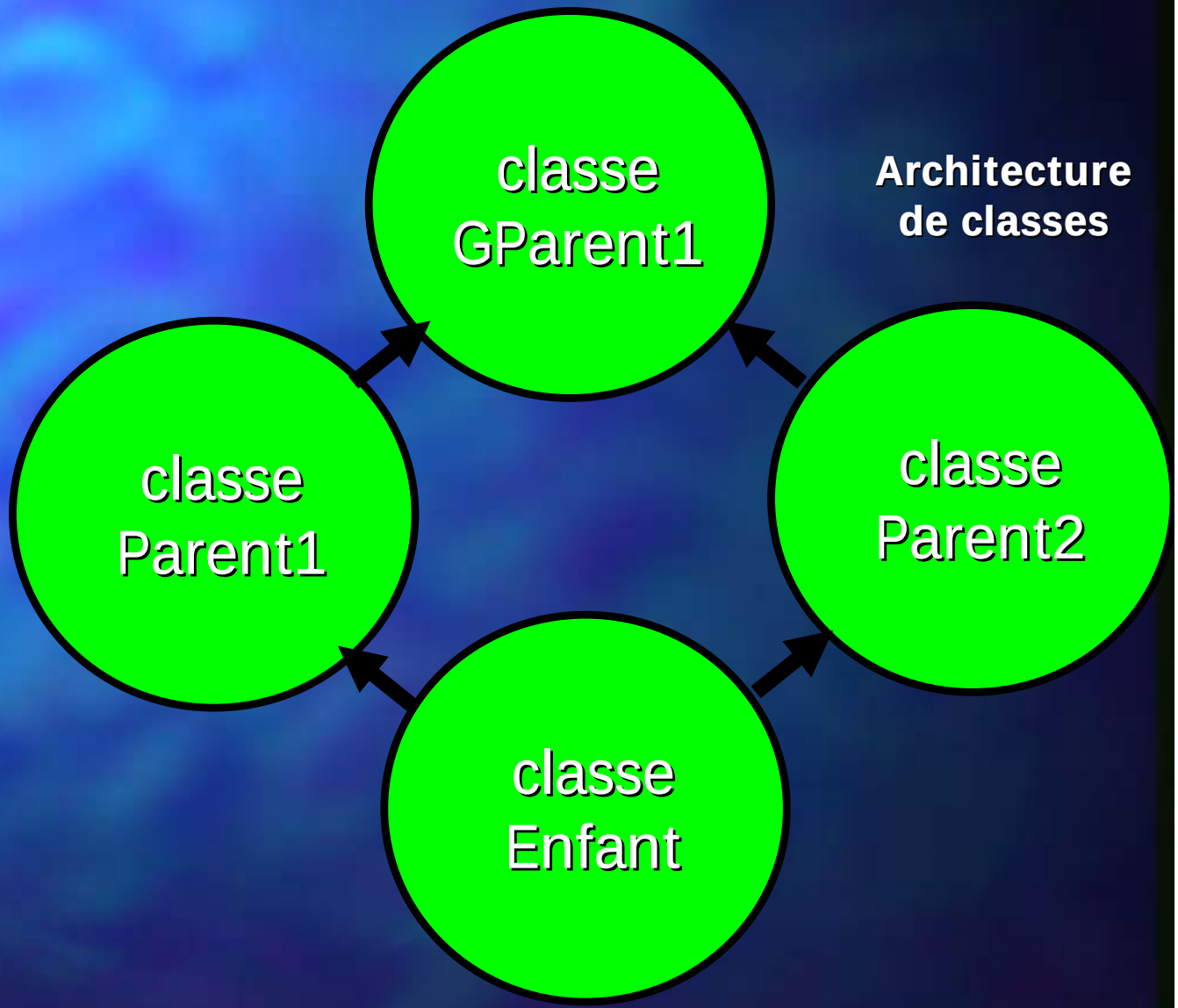


Héritage multiple

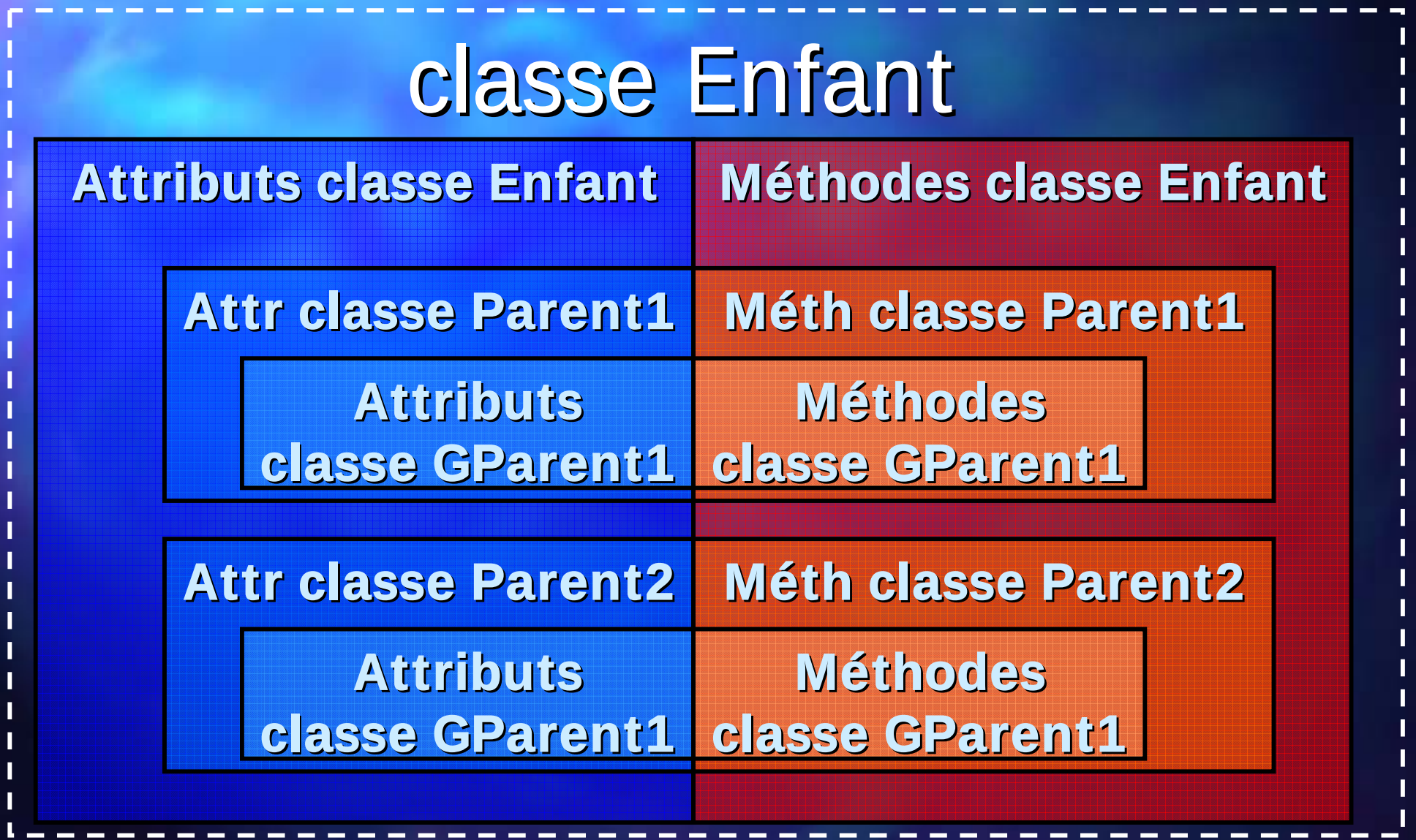


Héritage multiple

```
class GParent1
{ ... };
class Parent1 :
   public GParent1
{ ... };
class Parent2 :
   public GParent1
{ ... };
class Enfant :
   public Parent1,
   public Parent2
{ ... };
```

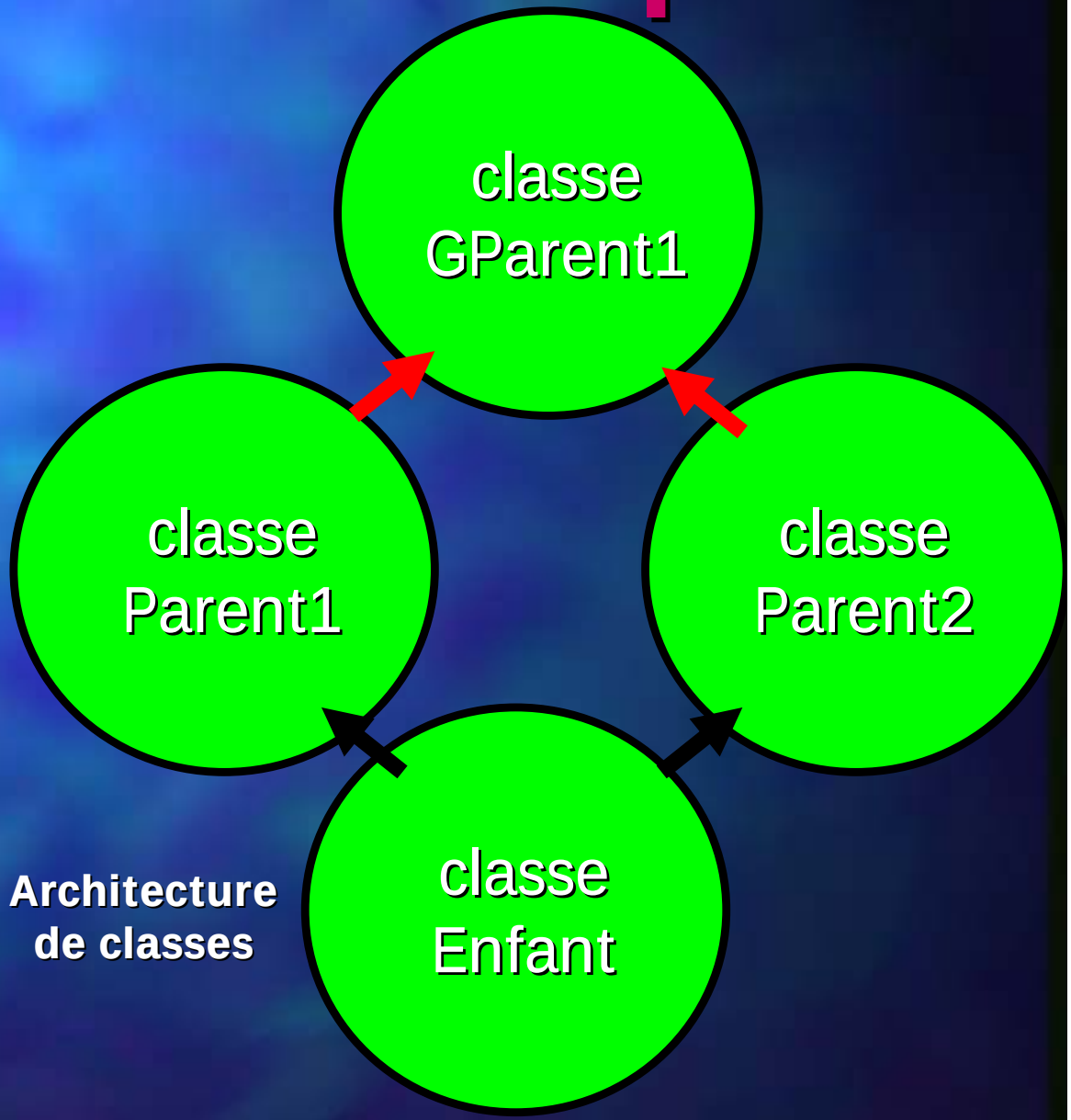


Héritage multiple



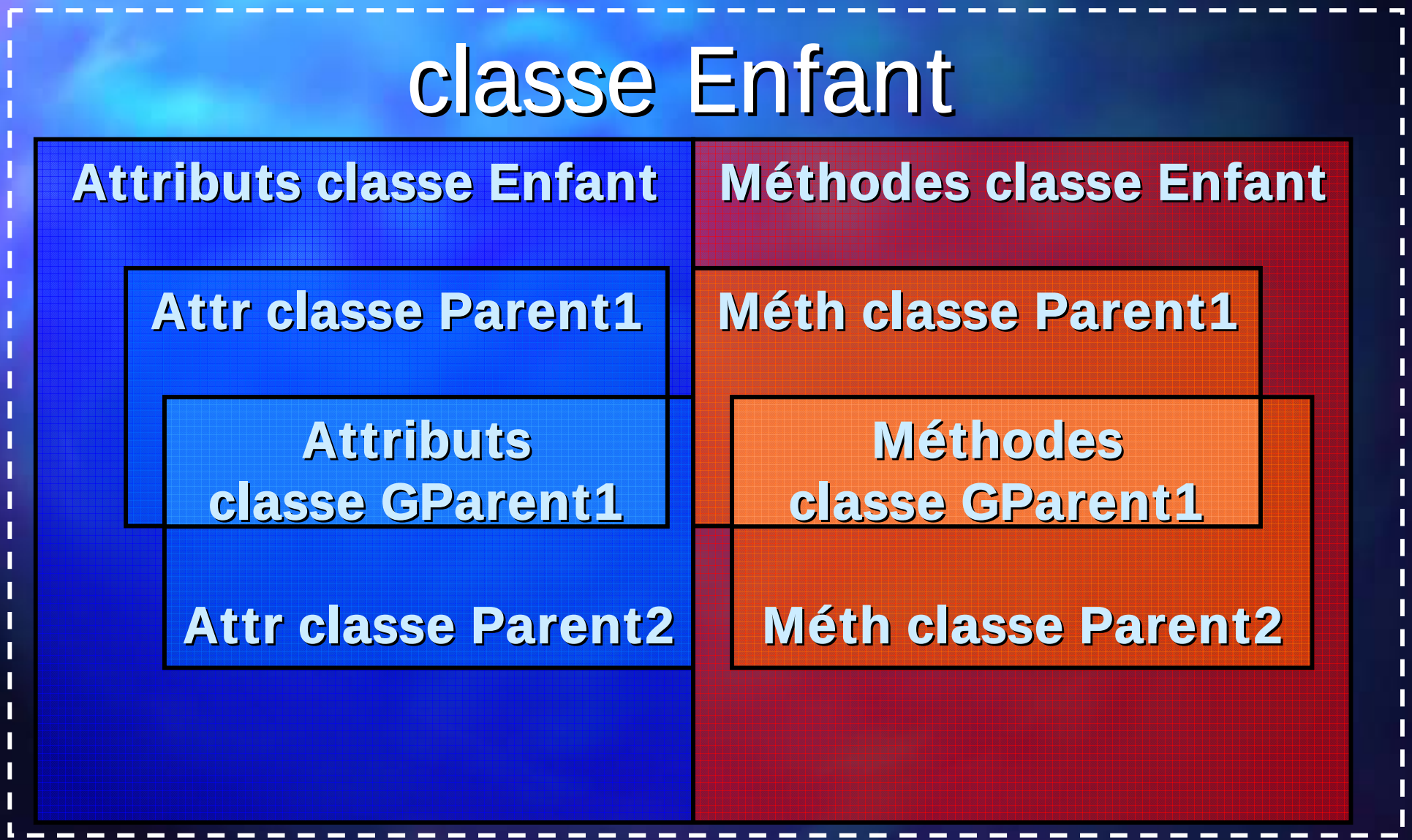
Héritage virtuel multiple

```
class GParent1
{ ... };
class Parent1 :
   virtual public GParent1
{ ... };
class Parent2 :
   virtual public GParent1
{ ... };
class Enfant :
   public Parent1,
   public Parent2
{ ... };
```



Architecture de classes

Héritage virtuel multiple

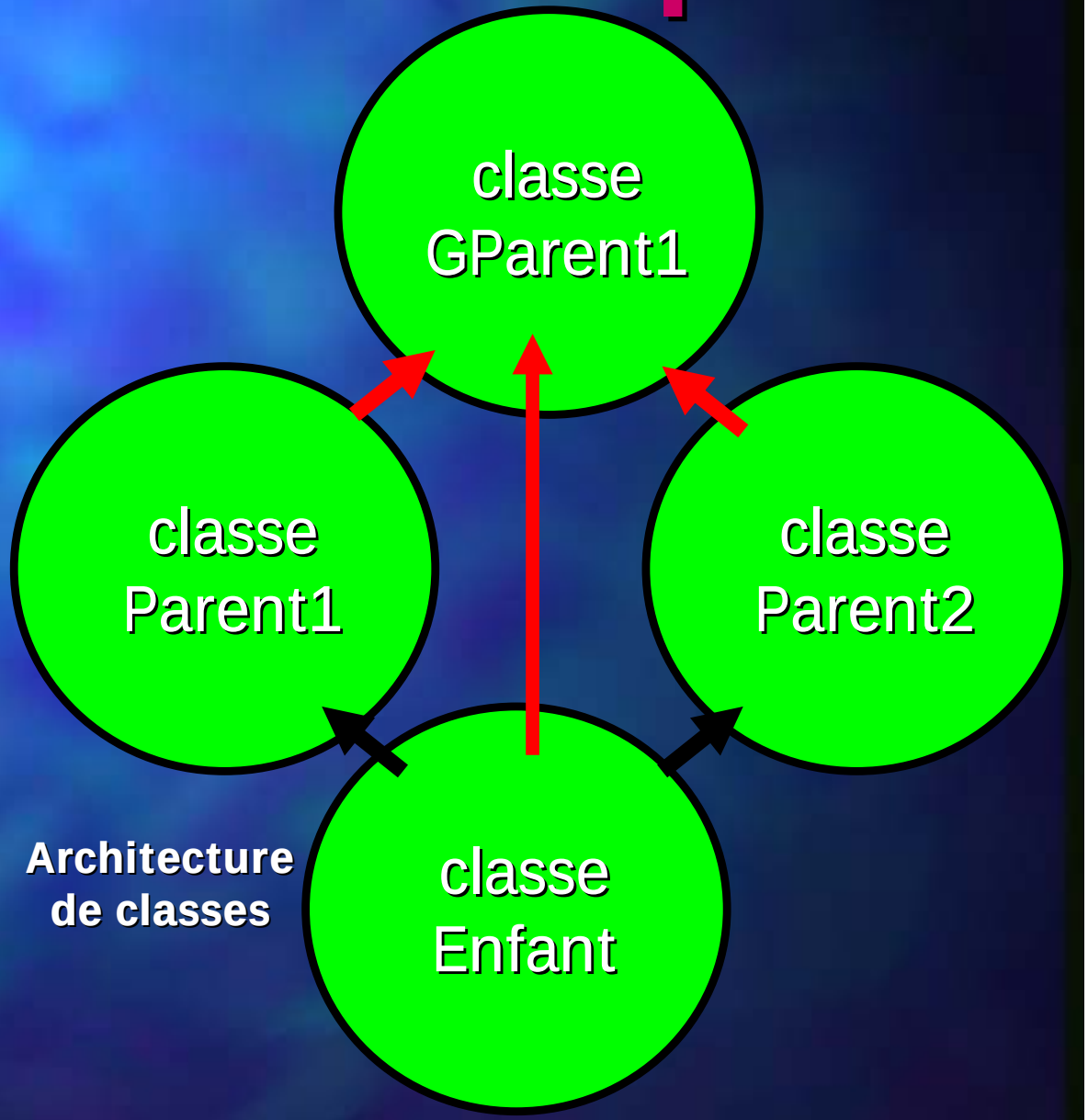


Comment se construit un grand-parent virtuel ?

- Si le grand-parent est hérité plus d'une fois d'une manière virtuelle alors:
  - tout éventuel appel explicite du constructeur du grand-parent au niveau des parents est ignoré par le compilateur
  - il appelle automatiquement le constructeur sans paramètres du grand-père (s'il n'existe pas : erreur !)
- Éviter cette contrainte :
  - on déclare en plus un héritage virtuel direct du grand-parent (qui devient aussi parent)
  - maintenant on peut appeler explicitement son constructeur (avec paramètres)

Héritage virtuel multiple

```
class GParent1
{ ... };
class Parent1 :
   virtual public GParent1
{ ... };
class Parent2 :
   virtual public GParent1
{ ... };
class Enfant :
   public Parent1,
   public Parent2,
   virtual public GParent1
{ ... };
```



Architecture de classes

Constructeurs - Héritage

- Syntaxe d'appel des constructeurs des classes hérités : le même que pour les attributs
- La classe Enfant peut appeler explicitement (sans être obligée) tout constructeur non-privé des classes parents (1 seul par classe héritée)
- Pour le reste des parents le compilateur appelle par défaut leur constructeur sans paramètres
- L'appel des constructeurs des grand-parents : interdit ou dangereux (en fonction de compilateur)
- L'ordre d'appel effectif = l'ordre de déclaration

Héritage
Construction / Destruction

- Construction d'un sous-objet hérité en mode virtuel
  - Une classe Enfant ne peut pas appeler les constructeurs des classes grands-parents héritées en mode virtuel
  - Si l'héritage virtuel est multiple le compilateur ignore tous les appels des parents aux constructeurs de la classe en question et appelle automatiquement le constructeur sans paramètres (qui doit exister !)
  - Pour éviter cela : hériter encore une fois directement
- Destruction: le compilateur appelle, après le corps du destructeur, tous les destructeurs des classes héritées

Visibilité des membres dans un héritage multiple

- Par défaut tous les attributs et les méthodes hérités sont visibles à la classe enfant (en respectant les droits d'accès). Mais on peut avoir :
  - visibilité cachée par un membre de l'enfant
  - ambiguïté entre deux membres portant le même nom mais se trouvant dans deux sous-objets différents
- Solutions ?
  - opérateur de résolution de portée :: vers l'espace du sous-objet (conseillé)
  - opérateur de *typecast* () vers le type du sous-objet (à éviter, il provoque des copies)

Héritage - Accès

- L'accès aux membres hérités est soumis aux droits d'accès résultant de l'héritage
- Sans le préciser, par défaut, l'héritage est de type private
- L'héritage ne fait que renforcer les contraintes d'accès (voir le tableau des droits)
- Exception: les opérateurs membres sont toujours hérités en mode public (il est impossible de restreindre leur accès)

Droits d'accès - Héritage

- Exemple :

```
class D : public A, protected B, private C
{
   //...
};
```

- L'héritage ne fait que restreindre les droits d'accès des membres hérités
- Les statuts de virtual ou static des membres restent inchangés, indépendamment du type d'héritage.

Droits d'accès - Héritage

Accès initial	Type d'héritage		
	public	protected	private
public	public	protected	private
protected	protected	protected	private
private	pas d'accès	pas d'accès	pas d'accès

Droits d'accès - Relâchement

- Classes amies :

```
class ClasseA : public classeB
{ friend ClasseC; ... };
```

  - 📖Toute instance de ClasseC peut accéder aux membres de ClasseA et membres hérités de ClasseB comme toute méthode de ClasseA

- Transfert de l'amitié par héritage :

```
class ClasseD : public classeC
{ ... };
```

  - 📖Uniquement les méthodes héritées de ClasseC ont le droit d'accéder les membres protégés de ClasseA.

Droits d'accès - Relâchement

- Fonctions amies :

```
class Cls1
{ friend ostream & operator <<(ostream&,const Cls1 &);
  friend istream & operator >>(istream&, Cls1 &);
  //...
};
ostream & operator <<(ostream &out, const Cls1& obj)
{
  return out<< ... <<endl;
}
```

📖La fonction amie peut accéder aux membres de obj et tout membre hérité comme toute méthode de Cls1

Tableau récapitulatif

Application des droits d'accès			classe A			héritière de public A			extérieur		
			méthodes const	méthodes	méthodes statiques	méthodes const	méthodes	méthodes statiques	fonctions et classes amies	autres	vers const A
classe A	attributs	public	L	L/E		L	L/E		L/E	L/E	L
		protected	L	L/E		L	L/E		L/E		
		private	L	L/E					L/E		
	méthodes const	public	A	A		A	A		A	A	A
		protected	A	A		A	A		A		
		private	A	A					A		
	méthodes	public		A			A		A	A	
		protected		A			A		A		
		private		A					A		
	attributs statiques	public	L/E	L/E	L/E	L/E	L/E	L/E	L/E	L/E	L/E
		protected	L/E	L/E	L/E	L/E	L/E	L/E	L/E		
		private	L/E	L/E	L/E				L/E		
	méthodes statiques	public	A	A	A	A	A	A	A	A	A
		protected	A	A	A	A	A	A	A		
		private	A	A	A				A		

Surcharge / polymorphisme

- En C++ le polymorphisme est la capacité à donner le même nom à des fonctions / méthodes / opérateurs qui ont un comportement diffèrent en fonction des paramètres, des objets et du contexte dans lequel on les emploie.
- Exemples en C "classique" :
  - `+ - / * = < >`
  - comportement différent réel / entier / signé etc.

Types de polymorphisme

- Polymorphisme statique de surcharge
  - surcharge d'une fonction globale ou méthode
  - appel en fonction du nombre et type de paramètres
- Polymorphisme d'héritage
  - on peut utiliser un objet enfant à la place du parent parce qu'il le contient
  - deux versions: statique et dynamique

Statique = comportement connu et déterminé à la compilation

Dynamique = comportement inconnu à la compilation et déterminé à l'exécution, par l'objet

Polymorphisme d'héritage statique

- Une classe enfant peut surcharger (redéfinir) le corps d'une méthode du parent à la condition de respecter le même prototype (même signature)
- Les nouvelles méthodes de l'enfant appelleront par défaut la méthode surchargée alors que les méthodes héritées appelleront la méthode initiale du parent

Polymorphisme d'héritage statique

```
class Parent
{ // ...
  const char* GetName() {return "Parent";}
  void AffName() {printf("%s\n",GetName());}
};

class Enfant : public Parent
{ // ...
  const char* GetName() {return "Enfant";}
  void Aff2Name() {printf("%s\n",GetName());}
  void Aff3Name() {printf("%s\n",Parent::GetName());}
  void Aff4Name() {printf("%s\n",AffName());}
};
```

Polymorphisme d'héritage statique

```
void main()
{ Enfant enf1, *penf2=new Enfant;
  Parent par1;
  printf("%s\n",enf1.GetName());                  // "Enfant"
  printf("%s\n",penf2->GetName());                // "Enfant"
  printf("%s\n",(Parent*)penf2->GetName());       // "Parent"
  printf("%s\n",par1.GetName());                  // "Parent"
  printf("%s\n",((Parent*)&enf1)->GetName());// "Parent"
  enf1.AffName();                                 // "Parent"
  penf2->Aff2Name();                              // "Enfant"
  enf1.Aff3Name();                                // "Parent"
  ((Parent)enf1).AffName();                       // "Parent"
  enf1.Aff4Name();                                // "Parent"
};
```

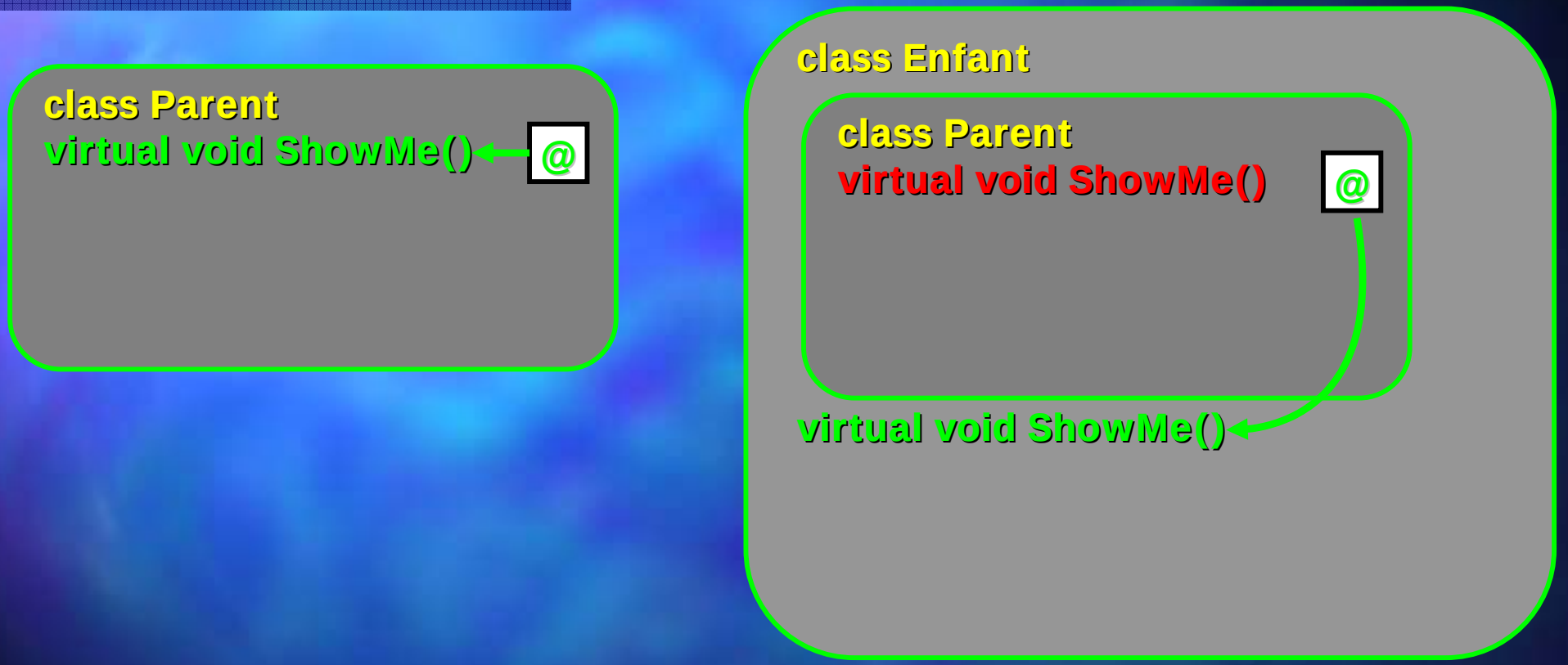
Polymorphisme d'héritage dynamique

- Le PHD n'est actif que pour les appels directs, sans opérateur de résolution de porté
- En utilisant le qualificatif virtuel dans la déclaration de prototype d'une méthode du parent, la surcharge devient effective aussi pour les méthodes du parent héritées par l'enfant
- La redéfinition remplace la méthode des parents
- Le caractère virtuel d'une méthode est défini dans la classe parent et il ne change plus à travers les héritages

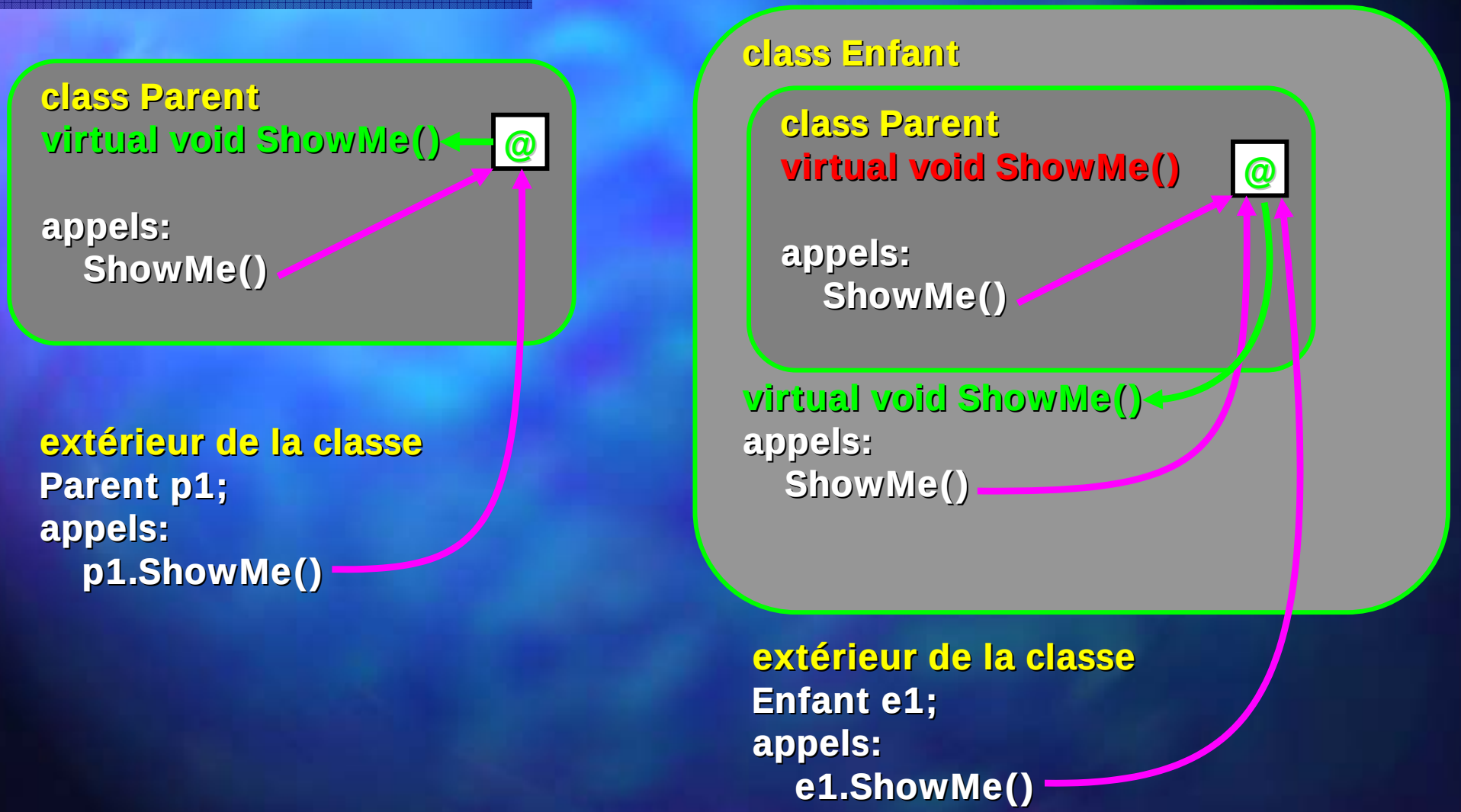
Polymorphisme d'héritage dynamique

- La version initiale de la méthode virtuelle surchargée reste visible aux parents et aux enfants seulement par l'intermédiaire de l'opérateur de portée Parent::Methode()
- Un petit-enfant peut ainsi appeler celle du parent Parent::Methode() ou celle du grand-parent GParent::Methode()
- L'adresse d'une méthode virtuelle est un attribut statique caché (tables virtuelles) de la classe, tout appel direct passe par cette adresse

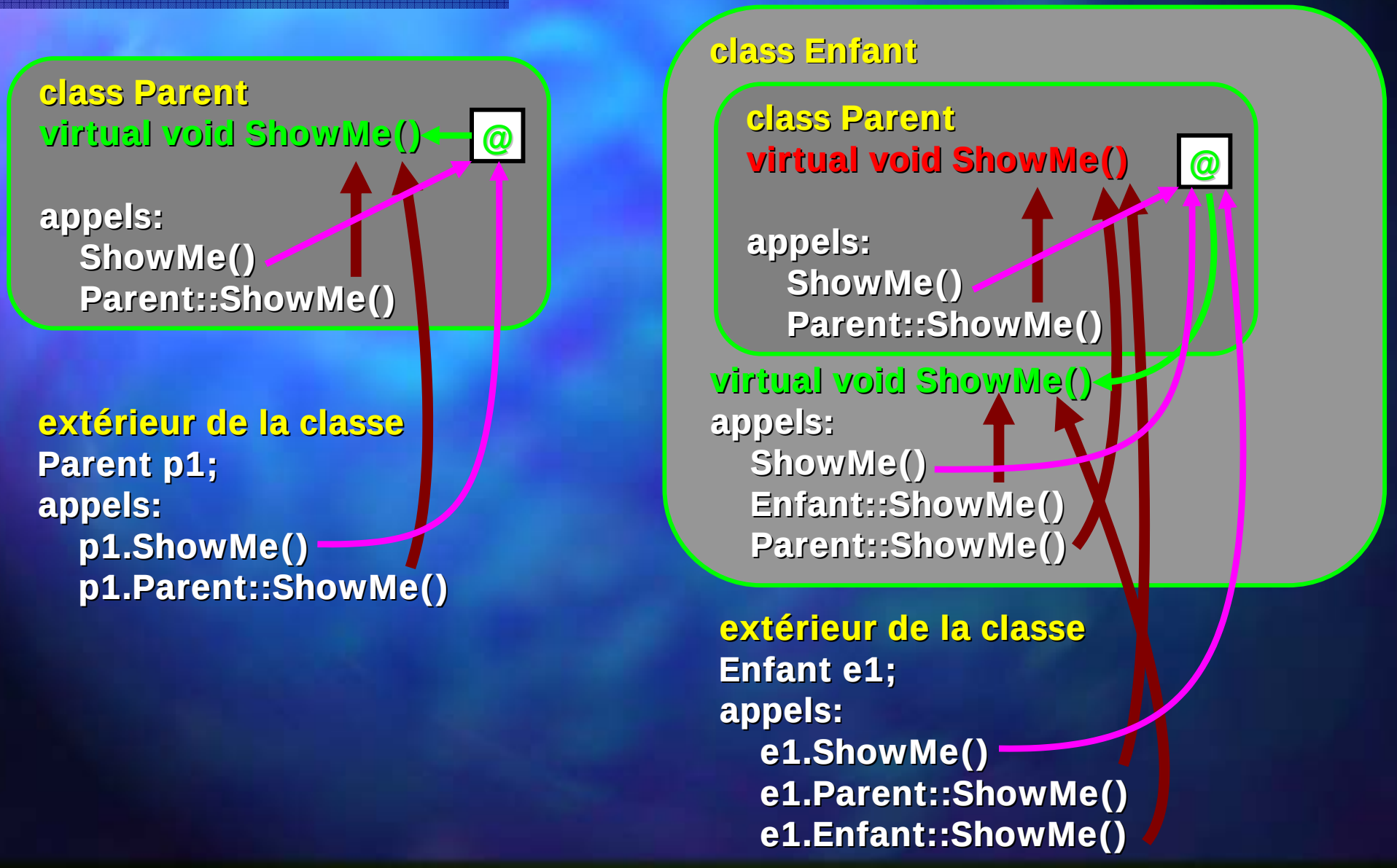
PHD : surcharge virtuelle



PHD : appels directs



PHD : appels par résolution de portée



Polymorphisme d'héritage dynamique

```
class Parent
{ // ...
  virtual const char* GetName() {return "Parent";}
  void AffName() {printf("%s\n",GetName());}
};

class Enfant : public Parent
{ // ...
  const char* GetName() {return "Enfant";}
  void Aff2Name() {printf("%s\n",GetName());}
  void Aff3Name() {printf("%s\n",Parent::GetName());
  void Aff4Name() {printf("%s\n",AffName());}
};
```

Polymorphisme d'héritage dynamique

```
void main()
{ Enfant enf1, *penf2=new Enfant;
  Parent par1;
  printf("%s\n",enf1.GetName());                    // "Enfant"
  printf("%s\n",penf2->GetName());                  // "Enfant"
  printf("%s\n",(Parent*)penf2->GetName());         // "Enfant"
  printf("%s\n",par1.GetName());                    // "Parent"
  printf("%s\n",((Parent*)&enf1)->GetName());       // "Enfant"
  enf1.AffName();                                   // "Enfant"
  penf2->Aff2Name();                                // "Enfant"
  enf1.Aff3Name();                                  // "Parent"
  ((Parent)enf1).AffName();                         // "Parent"
  enf1.Aff4Name();                                  // "Enfant"
};
```

Polymorphisme d'héritage dynamique

- La classe parent peut ne pas définir la méthode virtuelle : alors elle devient une méthode virtuelle pure

```
class Parent { // ...
  virtual const char* GetName()=0;
};
```

- Concrètement, l'adresse de la méthode (dans la table virtuelle) est initialisée à zéro
- Les héritiers ont le devoir de définir cette méthode, en respectant le prototype

Base abstraite

- Une classe qui a au moins une méthode virtuelle pure = base (classe) abstraite
- Elle ne peut pas avoir d'instances
- Les autres méthodes (virtuelles ou pas) peuvent appeler la méthode virtuelle pure (car il n'y a pas de danger).
- Un héritier pourra s'instancier seulement s'il a défini toutes les méthodes virtuelles pures héritées

Base abstraite

- Une base abstraite ne sert que pour être héritée: c'est le point de départ d'une famille de classes, par exemple
- Elle impose les prototypes des méthodes communes entre les héritiers, sans les définir
- Dans un programme:
  - on ne peut jamais avoir une instance d'une base abstraite mais
  - on peut avoir des pointeurs vers des bases abstraites (qui sont en réalité des enfants et qui ressemblent par polymorphisme à leur base abstraite)

Polymorphisme et RTTI (*run-time type information*)

- *Typecast* enfant ➔ parent : sûr donc automatique
- *Typecast* parent ➔ enfant : pas sûr donc explicite
- Solution C++ : l'information de type à l'exécution
  - il faut l'activer comme option de compilation
  - objets de taille plus grande (infos cachées)
  - retrouver le type de la classe :
    - opérateur `typeid` qui retourne l'info cachée : `const type_info&`

```
#include <typeinfo.h>
cout << typeid(*this).name( );
```

  - typecast C++ (intelligent) : dynamic_cast
    - static_cast<type>(expr) reinterpret_cast<type>(expr)
    - dynamic_cast<type>(expr) const_cast<type>(expr)

Classes et fonctions génériques

- Il serait bien d'avoir la possibilité de pouvoir choisir au dernier moment le type dans lequel on mémorise les coordonnées des sommets.
- Solution C "classique" :

```
typedef float Coord;
class FigGeom
{ ...
  Coord *xsommet, *ysommet; ...
};
```

  - Avant la compilation il suffit de remplacer `float` par le type désiré.
- Solution C++ : utilisation des *templates*.

Classes et fonctions génériques : déclaration

- La déclaration d'un *template* (modèle, patron) de fonction ou de classe admet une liste de types et de non-types (dans n'importe quel ordre):

```
template<int NT1, int NT2, class T1, typename T2>
```

  - Les types génériques sont précédés par les mots réservés équivalents `typename` ou `class`
  - Les paramètres non-types sont des paramètres constants à la compilation de type : entier, énumération, pointeur, référence (pas de constantes réelles ni des chaînes de caractères, ni d'objets, ni de types)
  - Les *templates* de classe admettent des valeurs par défaut pour les types et les non-types (à partir de la fin de la liste vers le début)

Classes et fonctions génériques : utilisation

- L'instanciation d'un *template* se fait d'une manière explicite ou implicite (que pour les fonctions) :
  `nom_classe_ou_function<N1, N2, T1, T2>`
- Les paramètres de type sont remplacés à la compilation par de types connus
- Les paramètres non-type sont remplacés par des valeurs constantes à la compilation
  - Attention : la vérification complète du code des *templates* se fait seulement à l'utilisation !
  - Attention : en VS 6.0 les *templates* des fonctions doivent avoir des arguments de type générique pour assurer leur signature différente (plus de contraintes pour VS >8.0)

Templates (patrons) de fonctions

- Permet de déclarer / définir une seule fois une fonction et de l'utiliser sur des types différents.
- Exemple de syntaxe :

```
template <class TypeVar> TypeVar carre(TypeVar v)
{
  return v*v;
}
```

- Appel explicite :

```
float fc=carre<float>(2.5f);
```

- Appel implicite :

```
int i1=10;
int i2=carre(i1);
```

Templates (patrons) de fonctions

- Spécialisation : permet de définir un cas particulier à comportement exceptionnel (que faire pour un type donné ...)
- Exemples de syntaxe :

```
template <> char carre<char>(char v)
{
  return '?';
}

template <> bool carre<bool>(bool v)
{
  return v;
}
```

Templates (patrons) de classes

- Permet de déclarer / définir une seule fois un modèle de classe et d'avoir à l'utilisation une série de classes qui travaillent sur des types différents.
- Syntaxe :

```
template <class TypeVar> class NomClasse
{ //...
  TypeVar membre;
  NomClasse(TypeVar val): membre(val) {}; // inline
  TypeVar GetVal(); // definition outline
};
// definition d'une méthode outline
template <class TypeVar>
  TypeVar NomClasse<TypeVar>::GetVal() {return membre;}
```

Templates (patrons) de classes

- Utilisation :

```
NomClasse<float> obj_float(2.5f);
NomClasse<char> obj_char('A');
NomClasse a(43); // erreur: pas de création implicite!
```

- Spécialisation du *template* NomClass :

```
template <> class NomClass<int>
{ /* ... */ };
```

- Syntaxe d'une fonction/opérateur qui travaille avec un *template* de classe NomClass :

```
template <class TypeVar>
  ostream& operator >>(const NomClass<TypeVar>& cls,
                       ostream& os) { /* ... */ }
```

Templates - Utilisation

- Les *templates* permettent l'utilisation des bibliothèques comme STL ou Boost. Contenu :
  - flux C++ (à utiliser avec les *templates* !)
  - encapsulation/manipulation de chaînes de caractères
  - conteneurs : vecteurs, listes, queues, piles, maps, sets accessibles par des itérateurs.
  - algorithmes d'ordonnancement, tri, parcours d'arbres, mélange aléatoire etc.
  - functors, smart-pointers
  - manipulation de mémoire (allocation, new, etc.)
  - fonctions mathématiques, logiques, limites numériques

Les exceptions

- Les exceptions permettent la mise en place d'un modèle de programmation différent du modèle "classique"
  - le modèle classique doit prévoir chaque type d'erreur possible et assurer le sortie (le dépilement) des fonctions appelées d'une manière explicite avec un code de retour adéquat
  - les exceptions permettent de se concentrer sur l'algorithme à coder, de générer ponctuellement des exceptions en cas d'erreur et de laisser le système et le compilateur remonter (dépiler les appels) jusqu'au niveau où l'on veut traiter ses erreurs / exceptions

Les exceptions

- Les exceptions sont de 2 types: matérielles et C++
  - soit de nature matérielle (système) en C/C++, le déclenchement est implicite et asynchrone :
    - la division par zéro
    - l'exécution d'une instruction invalide
    - l'accès à une zone mémoire invalide
    - le défaut de page (mémoire paginée)
  - soit définies par le langage C++ : déclenchées explicitement par l'instruction throw() suivie par l'instance d'une classe/type qui joue le rôle de l'exception (dans des fonctions/méthodes écrites par nous ou dans des bibliothèques C++)

Gestion des exceptions matérielles

- Les exceptions (des 2 types) non-gérées provoquent l'arrêt brutal du programme !
- Les exceptions matérielles sont gérées en utilisant un support spécifique API Win32 (en C) fourni par des extensions Visual C++ <excpt.h>:

```
__try { bloc à surveiller }
__except(EXCEPTION_EXECUTE_HANDLER)
   { code de gestion }
```

- En fonction des options de compilation, on peut mapper ou non, les exceptions matérielles sur des exceptions C++

Gestion des exceptions C++

- Les exceptions C++ sont gérées de la même manière par la succession surveillance - traitement
  - Le programmeur a la possibilité de surveiller le déclenchement dans un bloc de code à l'aide de l'instruction try
  - Le programmeur a la possibilité de répondre à une exception surveillée en plaçant après try{} une ou plusieurs instructions catch, chacune prévue pour un type précis d'exception
  - catch(...) placée en dernier, intercepte toute autre exception; l'emploi est optionnel

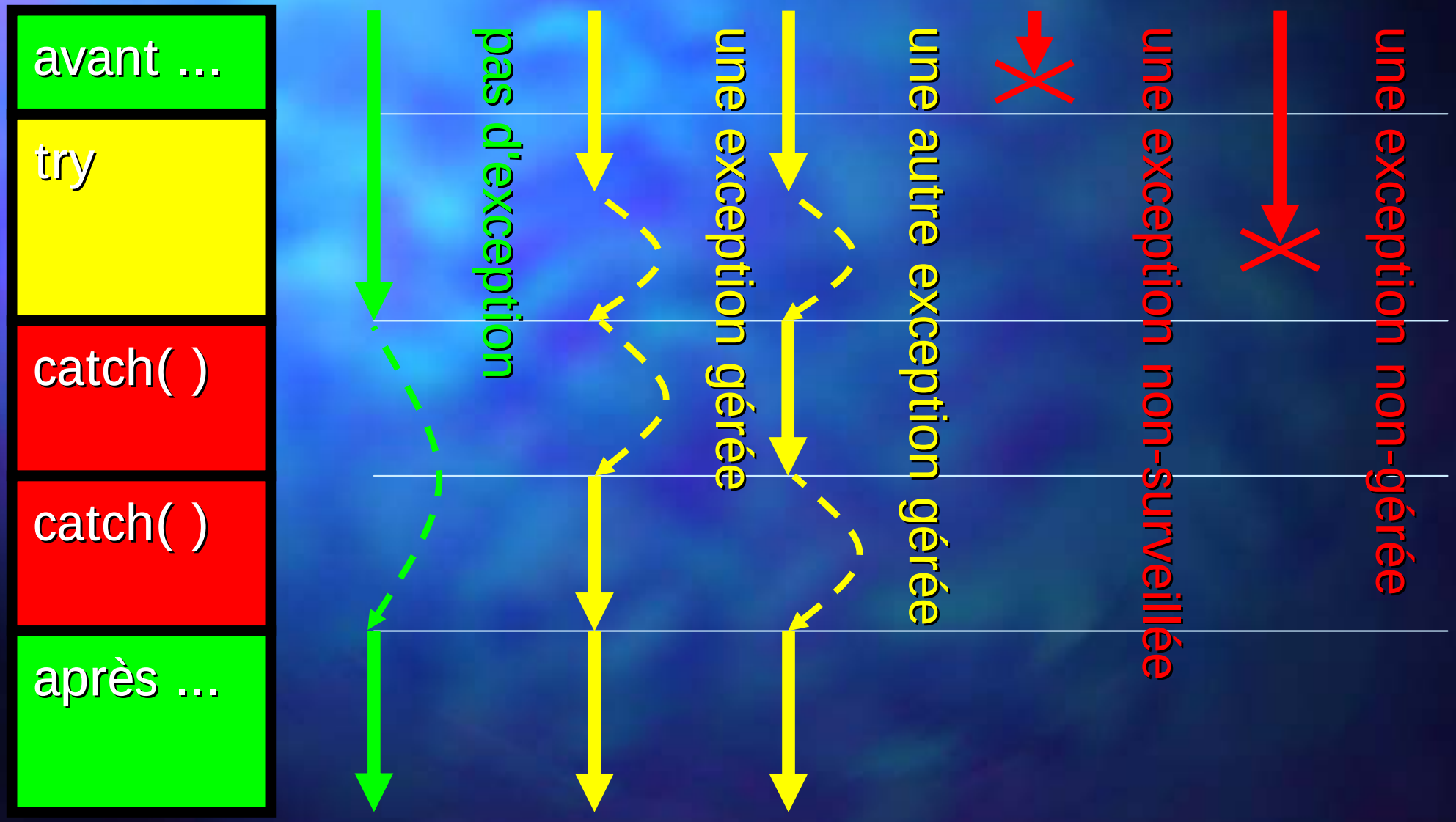
Gestion des exceptions

- Si le type d'exception générée par throw correspond à un catch qui suit le bloc try, alors il y a un transfert de l'exécution de l'endroit où l'exception a été générée vers le début du bloc catch concernée
- Sinon, le programme s'arrête avec le message "Exception non-gérée"
- Même chose si l'exception arrive dans une zone non-surveillée
- Par sa nature, la construction / destruction d'un objet global n'est pas surveillée

Gestion des exceptions

```
try
{ // bloc à surveiller
   throw(TypeM(/* parametres*/));
}
// pas d'instruction entre try et catch
catch(Type1 e1) // il en faut au moins une !
{/* bloc qui gère l'exception de type TypeN */ }
// ... autres catch, pas d'autres instructions
catch(TypeN eN)
{ /* bloc qui gère l'exception de type TypeN */ }
catch(...) // gestion par défaut, optionnel
{ /* gère toute autre exception non-traitée */ }

// ... suite normale du programme
```

Gestion des exceptions



Imbrication des exceptions

- Une exception cherche son gestionnaire parmi les catch les plus imbriquées
- Si elle ne le trouve pas, alors elle le cherche parmi les catch suivant le bloc try de niveau supérieur etc.
- Pas de gestionnaire = fin du programme