Ecole Nationale d'Ingénieurs de Brest
Université de Bretagne Occidentale

Programmation par objets

— Le langage C++ —

J. Tisseau , S. Morvan

– 1994/1997 –

V. Rodin

– Version révisée novembre 2023 –

Table des matières

- Les langages à objets 3
- Du C au C++ : le premier + ... 37
- Les classes en C++ 81
- La surdéfinition des opérateurs 279
- L'héritage en C++ 226
- Type concret de données en C++ 272
- Standard Template Library (STL) 275
- Un peu de C++ Moderne 280
- Bibliographie 286

Exemples disponibles :

http://labsticc.univ-brest.fr/˜rodin/FTP/Enseignements/M1Info/C++

Les langages à objets

- **Les langages à objets**
  1. Caractéristiques générales
  2. Smalltalk
  3. Eiffel
  4. C++

Les points communs

Modélisation : classes et instances

Activation : messages et méthodes

Construction : héritage et composition

Les différences (1)

Langage : tout objet ou hybride ?

Typage : fort ou faible ?

Métaclasses : les classes sont–elles elles–mêmes des objets ?

Attributs : simples ou complexes ?

Héritage : simple ou multiple ?

Les différences (2)

Envoi de messages : dynamique ou statique ?

- **liaison dynamique** : le lien message/méthode est traité à l'exécution
- **liaison statique** : le lien message/méthode est traité à la compilation

Ramasse–miettes : automatique ou manuel ?

Le langage Smalltalk

Concepteur : .... **A. Kay**

Date de naissance : .... **Début des années 1970**

Lieu de naissance : .... **Xerox PARC**

But initial : .... **Outil personnel de gestion d'informations**

Particularités

- Tout est objet.
- Tout objet est instance d'une classe.
  Une classe est instance d'une autre classe : sa métaclasse.
- Un objet est uniquement accessible par envois de messages.

Le système Smalltalk–80

- A la fois langage interprété, système d'exploitation et environnement de programmation.
- Nombreuses classes prédéfinies.

Une classe en Smalltalk

```
"Definition de la classe Pile"
Object subClass: #Pile
   instanceVariableNames: 'contenu'

"Methodes de classe"
creer
   "retourne une instance de Pile"
   ^ (self new) initialiser
```


Une classe en Smalltalk (suite)

```
"Methodes d'instance"
initialiser
   contenu <- OrderedCollection new

empiler: unObjet
   contenu addLast: unObjet

depiler
   contenu removeLast

sommet
   ^ contenu last
```


Le langage Eiffel

Concepteur : .... **B. Meyer**

Date de naissance : .... **Fin des années 1980**

Lieu de naissance : .... **ISE – Santa Barbara**

But initial : .... **Langage pour le Génie Logiciel**

Particularités

- Classes et objets
- Héritages simple et multiple
- Polymorphisme et liaison dynamique
- Généricité
- Programmation par contrat
- Traitement des exceptions
- Environnement de programmation

Une classe en Eiffel

```
class Pile[T]
  vide, empiler, depiler, sommet
feature
  representation : TABLEAU[T];
  taille_max : INTEGER;
  nb_elements : INTEGER;

  Create(n : INTEGER) is
   do
    if n > 0 then taille_max := n end;
    representation.Create(1,taille_max)
   end; -- Create
```


Une classe en Eiffel (suite)

```
vide : BOOLEAN is
 do
  Result := (nb_elements = 0)
 end; -- vide
sommet : T is
 require
  not vide
 do
  Result := representation.entree(nb_elements)
 end; -- sommet
```


Une classe en Eiffel (suite)

```
  empiler(x : T) is
   do
    nb_elements := nb_elements + 1;
    representation.entre(nb_elements,x)
   ensure
    not vide;
    sommet := x;
    nb_elements = old nb_elements + 1
   end; -- empiler
```

Une classe en Eiffel (fin)

```
depiler is
 require
  not vide
 do
  nb_elements := nb_elements - 1
 ensure
  nb_elements := old nb_elements - 1
 end; -- depiler
```


Le langage C++

Concepteur : .... **B. Stroustrup** `http://www.research.att.com/~bs`

Date de naissance : .... **Début des années 1980**

Lieu de naissance : .... **Laboratoires Bell (ATT)**

But initial : .... **Programmation objet en C**

Le premier +

- Compatibilité avec le langage C
- Typage fort
- Références
- Surdéfinition des fonctions
- Généricité

Le deuxième +

- Classes et objets
- Héritages simple et multiple
- Polymorphisme et liaison dynamique

Mon premier programme C++

```
/* mon premier programme en C++ */

#include <iostream>

using namespace std;       // Accès à la bibliothèque standard

int main(void)                           // fonction principale
{
   cout << "Hello world !" << endl;
   return 0;
}
```

Remarque : Absence de `using namespace std;`
$\Longrightarrow$ `cout` $\rightarrow$ `std::cout` et `endl` $\rightarrow$ `std::endl`

Mon premier programme C++ : remarques

- **2 types de commentaires**
  `/* ... */` (sur plusieurs lignes)
  `// ...` (en fin de ligne)
- **directive du préprocesseur**
  `#include <iostream>`
- **Utilisation de l'espace de nommage `std`**
  $\Longrightarrow$ **Accès à bibliothèque standard**
  `using namespace std; // ici, pour l'accès à cout et endl`
- **en–tête de la fonction principale**
  `int main(void)`
- **corps de la fonction principale**
  `{ cout << "Hello world !" << endl; return 0 ;}`

Une classe C++ : interface

```
// Pile.h :  une pile d'entiers

#ifndef PILE_H
#define PILE_H

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

using namespace std;

const int MAX = 10;
```

à suivre ...

Une classe C++ : interface publique

```
class Pile {
// Injection dans un flot
  friend ostream& operator<<(ostream& s, const Pile& p);

 public:
// Allocations
          Pile(void);
          Pile(const Pile& p);
          Pile& operator=(const Pile& p);
  virtual ~Pile(void);
```

à suivre ...

Une classe C++ : interface publique (suite)

```
 public:
// Inspecteurs
   bool estVide(void) const;
   bool estPleine(void) const;
   int sommet(void) const;
// Manipulateurs
   void empiler(int x);
   int depiler(void);
```

à suivre ...

Une classe C++ : interface privée

```
 private:
   int _pile[MAX];
   int _taille;
}; // fin de la classe Pile

#endif
```

fin Pile.h

Une classe C++ : implémentation

```
// Pile.cpp

#include "Pile.h"

// Allocations
Pile::Pile(void)
{
   _taille = 0;
}

Pile::Pile(const Pile& p)
{
   _taille = p._taille;
   for(int i = 0; i < _taille; i++) _pile[i] = p._pile[i];
}
```

à suivre ...

Une classe C++ : implémentation

```
Pile& Pile::operator=(const Pile& p)
{
   if(this != &p) {
      _taille = p._taille;
      for(int i = 0; i < _taille; i++) _pile[i] = p._pile[i];
   }
   return *this;
}

Pile::~Pile(void)
{
}
```

à suivre ...

Une classe C++ : implémentation (suite)

```
// Inspecteurs
bool Pile::estVide(void) const
{
  return _taille == 0;
}

bool Pile::estPleine(void) const
{
  return _taille == MAX;
}

int Pile::sommet(void) const
{
  assert(!estVide());
  return _pile[_taille - 1];
}
```

à suivre ...

Une classe C++ : implémentation (suite)

```
// Manipulateurs
void Pile::empiler(int x)
{
  assert(!estPleine());
  _pile[_taille++] = x;
}

int Pile::depiler(void)
{
  assert(!estVide());
  return _pile[--_taille];
}
```

à suivre ...

Une classe C++ : implémentation (suite)

```
// Injection dans un flot
ostream& operator<<(ostream& s, const Pile& p)
{
  s << '(';
  if(!p.estVide()) {
    for(int i = 0; i < p._taille - 1; i++) {
      s << p._pile[i] << ',';
    }
    s << p._pile[p._taille - 1];
  }
  s << ')';

  return s;
}
```

fin Pile.cpp

Mon deuxième programme C++

```
// main.cpp

#include <iostream>
#include "Pile.h"

using namespace std;

int main(void)
{
  Pile p;
  int x, y;

  cout << "Donner 2 entiers svp :  ";
  cin >> x >> y;
  cout << "x = " << x << " , y = " << y << endl;
```

à suivre ...

Mon deuxième programme C++ (suite)

```
   p.empiler(5);
   p.empiler(x);
   p.empiler(y);

   cout << p << endl;

   while(!p.estVide()) {
      cout << "sommet = " << p.depiler() << endl;
   }

   return 0;
}
```

fin `main.cpp`

Compilation d'un programme C++

```
$ ls
main.cpp Pile.cpp Pile.h
$ g++ -c Pile.cpp
$ ls
main.cpp Pile.cpp Pile.h Pile.o
$ g++ -c main.cpp
$ ls
main.cpp main.o Pile.cpp Pile.h Pile.o
$ g++ Pile.o main.o -o main
$ ls
main main.cpp main.o Pile.cpp Pile.h Pile.o
$
```


Fichier `makefile` correspondant

```
main:  Pile.o main.o
        g++ Pile.o main.o -o main

Pile.o:  Pile.h Pile.cpp
        g++ -c Pile.cpp

main.o:  Pile.h main.cpp
        g++ -c main.cpp

clean:
        rm -f *.o
        rm -f a.out main core
```


Exécution d'un programme C++

```
$ ls
main main.cpp main.o Pile.cpp Pile.h Pile.o
$
$ ./main
Donner 2 entiers svp :  8 7 ↩
x = 8 , y = 7
(5,8,7)
sommet = 7
sommet = 8
sommet = 5
$
```

Du C au C++ : le premier +

■ **Du C au C++ : le premier +**

1. **C et C++**
2. **Les variables en C++**
3. **Les fonctions en C++**
4. **La généricité en +**
5. **Les espaces de nommage (`namespace`) en +**

C et C++

■ **C et C++**

1. **Compatibilité C/C++**
2. **Un typage + fort**
3. **+ de mots réservés**
4. **Des entrées/sorties + simples**
5. **C + C++**

Compatibilité C/C++

- Un compilateur C++ peut compiler un code C écrit selon la norme ANSI C.

Un typage + fort

- Le langage C est un langage *faiblement typé*.
  Les compilateurs C sont “laxistes” et laissent beaucoup (trop) de liberté au programmeur.

- Le langage C++ introduit un *typage fort* beaucoup + strict.
  Les compilateurs font + de vérifications tout en produisant un code aussi efficace.

+ de mots réservés

asm	**delete**	if	return	**try**
auto	do	**inline**	short	typedef
break	double	int	signed	union
case	else	long	sizeof	unsigned
catch	enum	**new**	static	**virtual**
char	extern	**operator**	struct	void
class	float	**private**	switch	volatile
const	for	**protected**	**template**	while
continue	**friend**	**public**	**this**	
default	goto	register	**throw**	

Des sorties + simples

```
/* en C */

#include <stdio.h>


int main(void)
{
 int i = 5;
 char* str = "Hello";
 double x = 3.14;

 printf("%d%s%f\n",
        i, str, x);
 return 0;
}
```

```
// en C++

#include <iostream>
using namespace std;

int main(void)
{
 int i = 5;
 char* str = "Hello";
 double x = 3.14;

 cout << i << str << x
      << endl;
 return 0;
}
```


Des entrées + simples

```
/* en C */

#include <stdio.h>


int main(void)
{
 int i;
 char str[80];
 double x;

 scanf("%d%s%lf",
       &i, str, &x);
 return 0;
}
```

```
// en C++

#include <iostream>
using namespace std;

int main(void)
{
 int i;
 char str[80];
 double x;

 cin >> i >> str >> x;

 return 0;
}
```


C + C++

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

char* f(short, double);            // fonction compilée en C
char* g(const char*, int);         // fonction compilée en C

#if __cplusplus
}
#endif
```

Les variables en C++

■ **Les variables en C++**

1. **Définition au + près**
2. **Définition des constantes**
3. **Les références en +**
4. **Résolution de portée**
5. **Allocation dynamique + simple**

Définition au + près

```
/* en C */
/* définition au loin */

{
   int i;
   ...
   for(i = 0; i < n; i++)
   {
      ...
   }
   ...
}
```

```
// en C++
// définition au plus près

{
   ...
   for(int i = 0; i < n; i++)
   {
      ...
   }
   ...
}
```


Définition des constantes

```
/* en C */
/* portée :  le fichier */

#define PI 3.14

static const int i = 5;
```

```
// en C++
// portée :  le fichier

const double PI = 3.14;

const int i = 5;
```

Rappel de C :

`static` ⇒ portée le fichier
pas `static` ⇒ portée le(s) fichier(s)

Les références en +

- On définit un type *référence* en faisant suivre le spécificateur de type par l'opérateur d'adressage.

  `type&`

- Une *référence*, de même qu'une constante, doit être initialisée.

  `int i; int& j = i;`

- Les *références* sont des *alias*.
  Elles sont utilisées comme alternative au nom de l'objet avec lequel elles ont été initialisées.

Exemple de références

```
int i;
int& ref = i;                              // uniquement en C++
int* p = &i;
(i == ref) && (*p == ref) && (p == &ref); // expression
                                          // booléenne vraie...!
```

Les *références* sont essentiellement utilisées comme arguments ou types de fonctions.

Résolution de portée

```
#include <iostream>                          // uniquement en C++
using namespace std;

int i = 0;                                                  // ::i

int main(void)
{
  int i = 1;
  cout << "i = " << i << endl;                           // i = 1
  cout << "::i = " << ::i << endl;                     // ::i = 0
  {
    int i = 2;
    cout << "i = " << i << endl;                         // i = 2
    cout << "::i = " << ::i << endl;                   // ::i = 0
  }
  return 0;
}
```


Allocation dynamique + simple

```
/* en C */
#include <stdlib.h>
int main(void)
{
 int* i;
 char* str;

 i = (int*)
         malloc(sizeof(int));
 str = (char*)
    malloc(sizeof(char)*200);

 free(i); /* Pas de Ramasse- */
 free(str);      /* miettes */
 return 0;
}
```

```
// en C++

int main(void)
{
 int* i;
 char* str;

 i = new int;

 str = new char[200];


 delete i;     // Pas de Ramasse-
 delete [] str;       // miettes
 return 0;
}
```


Les fonctions en C++

- **Les fonctions en C++**
  1. **Prototypes de fonctions**
  2. **Arguments par défaut**
  3. **Définition des fonctions**
  4. **Fonctions en ligne**
  5. **Surdéfinition des fonctions**
  6. **Passage des arguments**

Prototypes de fonctions

```
/* en C */

f1();                    /* 1 */
int f1();                /* 2 */
int f1(double);          /* 3 */
int f1(double d);        /* 4 */
```

```
// en C++

int f1(double);          // 1
int f1(double);          // 2
int f1(double);          // 3
int f1(double d);        // 4
```

prototype ≡ `type identificateur(signature)`

Arguments par défaut

```
// uniquement en C++
// uniquement dans le prototype

void fct(const char* s="", int i=5, float f=0.0);

void fct(const char* s, int i, float f)
{ /* corps de la fonction */ }
```

Appels possibles:

```
fct(str,n,r);
fct(str,n);                       // fct(str,n,0.0)
fct(str);                         // fct(str,5,0.0)
fct();                            //  fct("",5,0.0)
```

Les arguments concernés doivent être les derniers de la liste d'arguments.

Définition des fonctions

```
/* en C : K & R */
char* f(x,y)
int x; char* y;
{ /* corps de la fonction */ }

/* en C : ANSI C */
char* f(int x, char* y)
{ /* corps de la fonction */ }
```

```
// en C++
char* f(int x, char* y)
{ /* corps de la fonction */ }
```


Fonctions en ligne

```
/* en C */

#define ABS(x) \
     ((x) > 0 ?  (x) :  -(x))
```

Appel :

```
y = sqrt(ABS(x));
```

```
// en C++
// à définir dans le .h

inline double ABS(double x)
{
   return (x > 0 ?  x :  -x);
}
```

Appel :

```
y = sqrt(ABS(x));
```

Une fonction inline C++ est aussi efficace qu'une macro–instruction C.

Surdéfinition des fonctions

```
// uniquement en C++

int max(int a, int b)                    { ...  le code ...  }
int max(const int tab[], int nbElems)    { ...  le code ...  }
int max(const Liste l)                   { ...  le code ...  }
double max(double a, double b)           { ...  le code ...  }

int j, k; int t[512]; Liste l; double x, y;

// ...  Ici, initialisations de j, k, t, l, x et y ...
```

Appels possibles :

```
max(j, k);                                   // max(int, int)
max(t, 512);                         // max(const int[], int)
max(l);                                   // max(const Liste)
max(x,y);                              // max(double, double)
```


Passage des arguments

Passage par valeur : La fonction n'a jamais accès aux arguments effectifs de l'appel : les valeurs que la fonction manipule sont des copies locales.

Le contenu des arguments effectifs de l'appel n'est pas modifié.

Passage des arguments (suite)

Problèmes :

1. quand un objet volumineux est passé à l'appel
2. quand le contenu des arguments effectifs doit être modifié

Solutions :

- en C : passer des pointeurs !
- en C++ : passer des pointeurs ou des références ! ... et des références de préférence !

Passage par valeurs

```
#include <iostream>                              // Fichier swap.cpp
using namespace std;

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

int main(void) {
  int x = 10, y = 20;
  cout << x << ',' << y << endl;
  swap(x,y);
  cout << x << ',' << y << endl;
  return 0;
}
```

```
$ g++ swap.cpp
$ ./a.out
10,20
10,20
```

Passage par pointeurs

```
#include <iostream>                          // Fichier swap.cpp
using namespace std;

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

int main(void) {
   int x = 10, y = 20;
   cout << x << ',' << y << endl;
   swap(&x, &y);
   cout << x << ',' << y << endl;
   return 0;
}
```

```
$ g++ swap.cpp
$ ./a.out
10,20
20,10
```


Passage par références

```
#include <iostream>                          // Fichier swap.cpp
using namespace std;

void swap(int& a, int& b) { int tmp = b; b = a; a = tmp; }

int main(void) {
   int x = 10, y = 20;
   cout << x << ',' << y << '\n';
   swap(x,y);
   cout << x << ',' << y << '\n';
   return 0;
}
```

```
$ g++ swap.cpp
$ ./a.out
10,20
20,10
```


La généricité en +

- **La généricité en +**
  1. **Les fonctions génériques**
  2. **Les structures génériques**

Fonctions génériques, exemple : `minimum` (1)

```
// uniquement en C++                  // Fichier testMinimun.cpp

#include <iostream>
using namespace std;

template <class T>                               // T : type formel
T minimum(T a, T b) { return a < b ?  a :  b; }

template <class T>                               // T : type formel
T minimum(const T t[], int n) {
   T minVal = t[0];
   for(int i = 1; i < n; i++)
      if(t[i] < minVal) minVal = t[i];
   return minVal;
}
```

à suivre ...

Fonctions génériques, exemple : `minimum` (2)

```
int main(void)
{
 int iT[] = {1, 9, 3, 0};
 double dT[] = {1.3, 9.2, 3.4, 0.7};

 cout << "minimum(3,2) => " << minimum(3,2) << endl;         // 2
 cout << "minimum(3.9,2.3) => " << minimum(3.9,2.3) << endl; // 2.3

 cout << "minimum(iT) => " << minimum(iT,4) << endl;         // 0
 cout << "minimum(dT) => " << minimum(dT,4) << endl;         // 0.7

 return 0;
}
```

fin `testMinimum.cpp`

```
$ g++ testMinimum.cpp
$ ./a.out
  ...                                        # Sorties écran
```


Exemple de fonction générique : `accumulate` (1)

```
int accumulate(int t[], int n, int init) {
  for(int i = 0; i < n; i++) init = init + t[i];
  return init;
}
```

Utilisation :

```
int array[] = { 1, 3, 5, 7, 9, 11, 2, 6 };

cout << accumulate(array, 8, 0) << endl;                // 44
```


Exemple de fonction générique : `accumulate` (2)

```
template <class T>
T accumulate(T t[], int n, T init) {
  for(int i = 0; i < n; i++) init = init + t[i];
  return init;
}
```

Utilisation :

```
int array[] = { 1, 3, 5, 7, 9, 11, 2, 6 };

cout << accumulate(array, 8, 0) << endl;                // 44
```


Exemple de fonction générique : `accumulate` (3)

```
template <class T, class BinaryOperation>
T accumulate(T t[], int n, T init, BinaryOperation op) {
  for(int i = 0; i < n; i++) init = op(init,t[i]);
  return init;
}
```

Utilisation :

```
int add(int x, int y)  { return x + y; }
int mult(int x, int y) { return x * y; }

int array[] = { 1, 3, 5, 7, 9, 11, 2, 6 };

cout << accumulate(array, 8, 0, add)  << endl;          // 44
cout << accumulate(array, 8, 1, mult) << endl;      // 124740
```

Exemple de fonction générique : `accumulate` (4)

```
template <class T, class BinaryOperation>
T accumulate(T* first, T* last, T init, BinaryOperation op) {
   while(first != last) init = op(init,*first++);
   return init;
}
```

Utilisation :

```
int add(int x, int y)  { return x + y; }
int mult(int x, int y) { return x * y; }

int array[] = { 1, 3, 5, 7, 9, 11, 2, 6 };

cout << accumulate(array, array+8, 0, add)  << endl;      // 44
cout << accumulate(array, array+8, 1, mult) << endl;  // 124740
```


Structures génériques

```
// uniquement en C++

template <class T1, class T2>
struct Pair {
   T1 first; T2 second;
};
```

Utilisation :

```
                               // Ici, en C++, on peut aussi écrire
struct Pair<double,int> p;  // simplement:  Pair<double,int> p;
p.first  = 3.14;
p.second = 3;
```


Exemple de structure générique : (1)

```
#include <assert.h>                 // Fichier structStack.cpp
#include <iostream>
using namespace std;

const int MAX = 10;

template <class T>
struct stack {
   T t[MAX];
   int n;
};
                          // Ici, en C++, on peut aussi écrire
struct stack<int> s1;     // simplement:  stack<int> s1;

stack< pair<int,char> > s2;    // De base, pair existe en C++
```

à suivre ...

Exemple de structure générique : (2)

```
template <class T>
bool isEmpty(const stack<T>& s) { return s.n == 0; }

template <class T>
bool isFull(const stack<T>& s) { return s.n == MAX; }

template <class T>
const T& top(const stack<T>& s) {
   assert(!isEmpty(s));
   return s.t[s.n - 1];
}
```

à suivre ...

Exemple de structure générique : (3)

```
template <class T>
void push(stack<T>& s, const T& t) {
   assert(!isFull(s));
   s.t[s.n++] = t;
}

template <class T>
T pop(stack<T>& s) {
   assert(!isEmpty(s));
   return s.t[--s.n];
}
```

à suivre ...

Exemple de structure générique : (4)

```
int main(void)
{
 pair<int,char> p;
 stack< pair<int,char> > s;
 s.n = 0; // Au départ la pile est vide
 for(int i = 0; i < MAX; i++) {
  p.first = i;
  p.second = 'r' + i; push(s,p);
 }
 while(!isEmpty(s)) {
  cout << pop(s).second << endl;
 }
 return 0;
}
```

```
$ g++ structStack.cpp
$ ./a.out
{
z
y
x
w
v
u
t
s
r
$
```

fin `structStack.cpp`

Les espaces de nommage en +

■ Les espaces de nommage en + (`namespace`)

1. **Intérêts des `namespace`**
2. **Exemple 1**
3. **Le mot clef `using`**
4. **Exemple 2**
5. **Exemple 3**

Intérêts des `namespace`

Un `namespace` permet de **regrouper** des entités comme :

- des objets,
- des fonctions,
- des classes

sous un même nom.

Exemple de `namespace` :

```
namespace aNameSpace
{
   int a,b;
}
```

Dans l'espace de nommage `a` et `b` sont accessibles directement.

En dehors, il faut préciser :
`aNameSpace::a` et
`aNameSpace::b`.

Exemple 1

```
#include <iostream>                                  // nameSpace1.cpp
using namespace std;
namespace first
{
 int var = 5;
}
namespace second
{
 double var = 3.14;
}
int main(void)
{
 cout << first::var  << endl; // 5
 cout << second::var << endl; // 3.14
 return 0;
}
```


Le mot clef `using`

Le mot clef `using` sert à introduire :

- un nom présent dans un espace de nommage afin qu'il soit directement accessible
  Exemple : `using first::var;`
- tous les noms présents dans un espace de nommage afin qu'ils soient directement accessibles
  Exemple : `using namespace first;`

Remarque : `using` peut être utilisé dans un bloc ou
en dehors de tout bloc (en global).

Exemple 2

```
#include <iostream>                                  // nameSpace2.cpp
using namespace std;                              // Validité :  le fichier
namespace first
{
 int var = 5;
}
namespace second
{
 double var = 3.14;
}
int main(void)
{
 using first::var;             // Validité (uniquement first::var): le bloc
 cout << var         << endl; // 5
 cout << second::var << endl; // 3.14
 return 0;
}
```


Exemple 3

```
#include <iostream>                                  // nameSpace3.cpp
using namespace std;                              // Validité :  le fichier
namespace first
{
 int var = 5;
}
namespace second
{
 double var = 3.14;
}
int main(void)
{
 using namespace first;       // Validité (de tout le namespace): le bloc
 cout << var         << endl; // 5
 cout << second::var << endl; // 3.14
 return 0;
}
```

Les classes en C++

- **Les classes en C++**
  1. C++ : le deuxième +
  2. Les classes
  3. Les objets
  4. Les fonctions membres
  5. Les fonctions amies
  6. Les classes génériques

C++ : le deuxième +

- **C++ : le deuxième +**
  1. C++ et programmation par objets
  2. Des structures aux classes

C++ et programmation par objets

- Le langage C++ introduit les notions de *classe* et d'*instance* de la programmation par objets.
- Un mécanisme de protection très élaboré des attributs et des méthodes permet une *encapsulation* efficace.
- En C++, l'*héritage* est simple ou multiple.
- Le typage est statique.
- Les fonctions virtuelles permettent le *polymorphisme* et la *liaison dynamique*.

Des structures aux classes

```
/* en C et en C++ */

#include <math.h>

struct X {
  int i;
  double (*f)(double);
};
```

Utilisation :

```
struct X x; double v;
x.i = 2;
x.f = &sin;
v = x.f(3.14);
```

```
// en C++

#include <math.h>

class X {
 public :
  int i;
  double f(double);
};

double X::f(double a)
{ return sin(a); }
```

Utilisation :

```
X x; double v;
x.i = 2;
v = x.f(3.14);
```

Les classes

- **Les classes**
  1. **Classification**
  2. **Encapsulation**
  3. **Classe ≡ composant logiciel**
  4. **Compilation séparée en C++**

Classification

- La classification regroupe des objets ayant la même structure (*attributs*) et le même comportement (*opérations* ou *méthodes*) au sein d'une même classe.
- Une classe est une *abstraction* qui décrit un ensemble, éventuellement infini, d'objets individuels.
- Chaque objet est dit *instance* de sa classe.

Exemple de classe en C++

```
class Point {
 public:
  // Attributes
   int x;
   int y;
  // Operations
   void display(void) const;      // const : méthode qui ne modifie pas l'objet
   void move(int dx, int dy);
};
```

Utilisation :

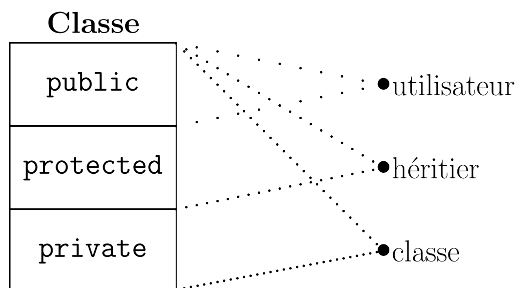
```
Point p; p.x = 12; p.y = 5;      // p est une instance de la classe Point
p.display(); p.move(2,-1);
```


Encapsulation

- L'encapsulation (ou *masquage d'information*) consiste à séparer les aspects externes d'un objet, accessibles par les autres objets, des détails de son implémentation interne, rendus invisibles aux autres objets.
- L'implémentation d'un objet peut être modifiée sans affecter les applications qui emploient cet objet.

Masquage d'information en C++

`private` , `protected` , `public`



Syntaxe C++

```
class ClassName {
 public :
  // Attributs et opérations publiques
 protected:
  // Attributs et opérations protégés
 private:
  // Attributs et opérations privés
};
```

Utilisation :

```
ClassName instance1, instance2;
```


Exemple de classe avec encapsulation

```
class Point {
 public:
  // Operations
  int  getx(void) const;
  int  gety(void) const;
  void display(void) const;
  void set(int x, int y);
  void move(int dx, int dy);
 private:
  // Attributes
  int _x;
  int _y;
};
```


Classe ≡ composant logiciel

Une classe constitue une unité de compilation indépendante.

classe ≡ type ≡ module

- Le fichier en–tête (`ClassName.h`) contient la déclaration de la classe. Il joue le rôle d'interface de la classe (*spécification*).
- Le fichier source (`ClassName.cpp`) contient la définition des fonctions membres (*implémentation*).

Interface et Implémentation

```
// Point.h
#ifndef POINT_H
#define POINT_H

class Point {
  public:
    // Operations
    int  getx(void) const;
    int  gety(void) const;
    void display(void) const;
    void set(int x, int y);
    void move(int dx, int dy);
  private:
    // Attributes
    int _x;
    int _y;
};
#endif
```

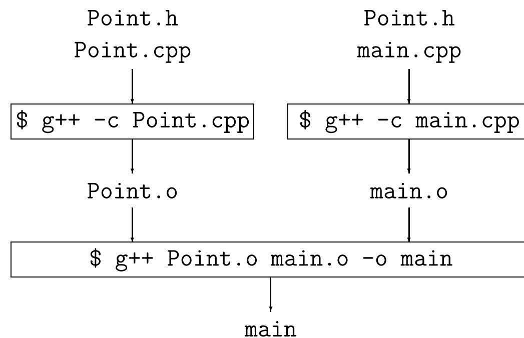
```
// Point.cpp

#include <iostream>
#include "Point.h"

using namespace std;

int Point::getx(void) const { return _x; }

int Point::gety(void) const { return _y; }

void Point::display(void) const
{ cout << '(' << _x << ',' << _y << ')'; }

void Point::set(int x, int y)
{ _x = x; _y = y; }

void Point::move(int dx, int dy)
{ _x += dx; _y += dy; }
```


Utilisation

```
// main.cpp

#include <iostream>
#include "Point.h"

using namespace std;

int main(void) {
    Point p;
    p.set(12,5);
    p.display(); cout << endl;
    p.move(2,-1);
    p.display(); cout << endl;
    return 0;
}
```

```
$ ls
main.cpp Point.cpp Point.h
$ g++ Point.cpp Main.cpp -o main
$ ls
main.cpp Point.cpp Point.h main
$ ./main
(12,5)
(14,4)
$
```


Compilation séparée en C++



Le fichier `makefile`

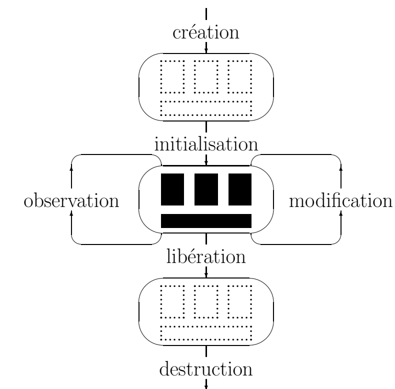
```
# makefile

main:  Point.o main.o
       g++ Point.o main.o -o main

Point.o:  Point.h Point.cpp
       g++ -c Point.cpp

main.o:  Point.h main.cpp
       g++ -c main.cpp
```

```
$ ls
makefile main.cpp Point.cpp Point.h
$ make
        g++ -c Point.cpp
        g++ -c main.cpp
        g++ Point.o main.o -o main
$ ls
makefile main.cpp Main.o
Point.cpp Point.h Point.o main
$ ./main
(12,5)
(14,4)
$
```

Les objets

- **Les objets**
  1. **La vie d'un objet**
  2. **Dynamique des objets**
  3. **Auto–référence**
  4. **Variables de classe**

La vie d'un objet



Création et initialisation

Création : C++ fournit un constructeur par défaut permettant la création de nouveaux objets.
On pourra définir ses propres constructeurs.

Initialisation : L'initialisation doit être prévue par le concepteur de la classe.

Adressage : Par défaut, C++ permet l'accès aux objets par l'intermédiaire de l'opérateur `&`.

Affectation

- Par défaut, l'affectation de deux objets de même type correspond à une simple recopie des valeurs des données, membres à membres, publiques, protégées et privées.
- Par défaut, les objets comportant une partie d'allocation dynamique n'obtiendront qu'une simple recopie des pointeurs sur les zones dynamiques.
- Pour une affectation complète, une surdéfinition de l'opérateur d'affectation (=) est nécessaire.

Opérations par défaut

```
class Point;                 // déclaration de la classe Point

Point point1, point2;        // instanciations des objets avec
                             // le constructeur par défaut...

Point* ptr = (Point*)0;                  // pointeur sur objet

ptr = &point1;                                   // adressage

Point& ref = point2;                             // référence

point1 = ref;                                  // affectation
```


Dynamique des objets : objets statiques

Objets statiques : variables globales, ou objets déclarés `static`, ils sont définis à la compilation.

```
int n = 5;                    // Variable globale

void f(void) {
  static int n = 0;     // Variable static propre à la fonction
  cout << n++ << endl; // => conservée entre 2 appels
}

class C {
  static int n;  // Variable static propre à la classe
};              // => partagée par tous les objets de la classe
```


Dynamique des objets : objets dynamiques

Objets dynamiques : définis par un pointeur sur une zone mémoire allouée par l'opérateur `new`.

Ces objets peuvent être détruits par l'opérateur `delete`.

```
int* i = new int;
int* t = new int[5];

delete i;
delete [] t;
```


Dynamique des objets : objets automatiques

Objets automatiques : variables locales ayant une durée de vie correspondant à l'activation du bloc concerné.

$\Longrightarrow$ création à l'entrée du bloc, destruction à la sortie !

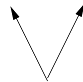
```
{
  int i = 1;
  cout << i << endl;
  {
    int i = 2;
    cout << i << endl;
  }
}
```

Dynamique des objets : objets temporaires

Objets temporaires : ils sont créés par le compilateur pour effectuer des calculs intermédiaires.

Pour contrôler la création de ces objets temporaires, il sera nécessaire de définir un constructeur par recopie.

```
int f(int x) { return x * x; }

cout << f(2) + f(4) << endl;
```



Création éventuelle d'objets temporaires

Exemples d'objets

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

const int MAX = 5;

int square(int i) { return (i * i); }

int main(void)
{
 int i = 0;
 int* array = new int[MAX];

 for(i = 0; i < MAX; i++) array[i] = square(i) + square(i);

 for(i = 0; i < MAX; i++) cout << array[i] << endl;

 delete [] array;

 return 0;
}
```


Auto–référence

- Le mot réservé `this` désigne un pointeur, implicitement déclaré, sur l'objet lui–même (auto–référence).
- Il peut être utilisé dans n'importe quelle fonction membre et constitue un alias de l'objet.
- Implicitement, chaque fonction membre possède comme premier argument le pointeur `this`, adresse de l'objet.

```
class C {
 public:
    int f(double);    // int f(C* this, double);
};
```

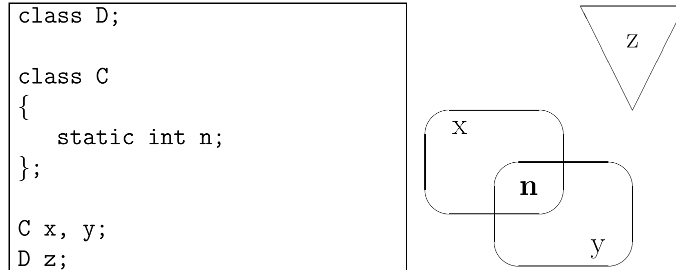

Le pointeur `this`

```
// Point.h
#ifndef POINT_H
#define POINT_H

class Point {
  public:
    // Operations
    int  getx(void) const;
    int  gety(void) const;
    void display(void) const;
    void set(int x, int y);
    void move(int dx, int dy);
  private:
    // Attributes
    int _x;
    int _y;
};
#endif
```

```
// Point.cpp
#include <iostream>
#include "Point.h"

using namespace std;

int Point::getx(void) const
{ return this->_x; }          // return _x;

int Point::gety(void) const
{ return this->_y; }          // return _y;

void Point::display(void) const
{ cout << '(' << _x << ',' << _y << ')'; }

void Point::set(int x, int y)
{ (*this)._x = x; (*this)._y = y; }

void Point::move(int dx, int dy)
{ this->_x += dx; this->_y += dy; }
```

Variables de classe

Les membres déclarés `static` sont des membres partagés par tous les objets de la classe et stockés en un seul endroit.



```
class D;

class C
{
   static int n;
};

C x, y;
D z;
```

x n y z

Variables de classe

- Par défaut, les variables `static` sont initialisées à 0, et ne peuvent pas être initialisées à la définition.
- Une donnée membre `static` peut apparaître comme argument par défaut d'une fonction membre.
- Une donnée membre `static` peut prendre pour valeur un objet de la classe.

Exemple de variable de classe

// Dans le `.h` de la classe

```
#ifndef POINT_H                                         =>  Fichier Point.h
#define POINT_H
class Point {
 public:
⇒  static const char* className;          // Variable de classe
   int  getx(void) const;
   int  gety(void) const;
   void display(void) const;
   void set(int x, int y);
   void move(int dx, int dy);
 private:
   int _x;
   int _y;
};
#endif
```


Initialisation d'une variable de classe

// Dans le `.cpp` de la classe

```
#include <iostream>                                    =>  Fichier Point.cpp
#include "Point.h"
using namespace std;

⇒ const char* Point::className = "Point";      // Initialisation

int Point::getx(void) const
{ return _x; }
int Point::gety(void) const
{ return _y; }
void Point::display(void) const
{ cout << '(' << _x << ',' << _y << ')'; }
void Point::set(int x, int y)
{ _x = x; _y = y; }
void Point::move(int dx, int dy)
{ _x += dx; _y += dy; }
```

Utilisation d'une variable de classe

// Dans le programme principal (un .cpp)

```
#include <iostream>                                    => Fichier main.cpp
#include "Point.h"

using namespace std;

int main(void)
{
⇒ cout << Point::className << endl;              // Utilisation
  return 0;
}
```

```
$ g++ Point.cpp main.cpp                    # un makefile, c'est mieux !
$ ./a.out
Point
```


Les fonctions membres

■ **Les fonctions membres**

1. **Prototype d'une fonction membre**
2. **Fonctions membres** `const`
3. **Constructeurs**
4. **Destructeur**
5. **Ecriture condensée**
6. **Fonctions membres** `inline`
7. **Pointeur sur fonction membre**
8. **Fonctions membres statiques**
9. **Bonnes pratiques**

Prototype d'une fonction membre

Une fonction membre est caractérisée par :

- son nom (identificateur)
- sa signature (liste des arguments)
- son type
- sa classe

```
type classe::nom(signature);
```

- Eventuellement le mot clef `const` (voir page 117)

```
type classe::nom(signature) const;
```


Exemples de prototypes de fonctions membres

type	classe :: nom (signature)
char*	String :: fct (int, const char*)
int	String :: fct (void)
String&	String :: fct (const String&)
String	String :: fct (void) const

Fonctions membres `const`

- Une fonction membre peut être autorisée à consulter mais pas à modifier l'objet sur lequel elle est appelée.

  La garantie que l'objet invoqué (`*this`) ne sera pas modifié est donnée par le mot réservé `const` suffixé à la liste des arguments.

  ```
  type classe::nom(signature) const;
  ```

- Une fonction membre `const` peut être invoquée sur un objet `const`; une fonction membre ordinaire ne le peut pas.

Exemple de fonction membre `const`

```
class C {
 public:
   void f(void);
   void g(void) const;
};
```

Utilisation :

```
const C a;

a.f();                    // erreur
a.g();                        // ok
```


Les constructeurs

- Un constructeur est une fonction membre dont le nom est le même que celui de la classe. Un constructeur n'a pas de type.
- Une classe comporte généralement plusieurs constructeurs (surdéfinition de fonctions) offrant à l'utilisateur un choix pour l'instanciation de nouveaux objets.

  En fonction du nombre et du type des arguments passés à l'appel, le constructeur adapté sera choisi.
- Si au moins un constructeur existe, l'utilisation du constructeur par défaut fournit par C++ n'est plus autorisée.

Exemples de constructeurs (1)

```
#ifndef C_H                                          // Fichier C.h
#define C_H

class C {
 public :
   C(void);                     // constructeur par défaut
   C(int n);               // constructeur avec initialisation
   C(const C& c);     // constructeur par recopie :  clônage
 private:
   int _n;
};

#endif // C_H
```

Exemples de constructeurs (2)

```
#include "C.h"                              // Fichier C.cpp

C::C(void)                         // constructeur par défaut
{
 _n = 0;
}
C::C(int n)                // constructeur avec initialisation
{
 _n = n;
}
C::C(const C& c)        // constructeur par recopie :  clônage
{
 _n = c._n;
}
```


Exemples de constructeurs (3): Utilisation

```
#include "C.h"                                   // mainC.cpp

int main(void)
{
 C c1;
 C c2(5);
 C c3(c2);

 ...

 return 0;
}
```

```
$ g++ C.cpp mainC.cpp                  # un makefile, c'est mieux !
$ ./a.out
```


Rôle des constructeurs

Le constructeur est appelé *après* la création de l'objet.
Il a pour rôle l'instanciation des objets, c'est–à–dire l'initialisation des attributs et l'allocation éventuelle de mémoire pour ces attributs.

Initialisation des objets dans les constructeurs (1)

Il est possible de transmettre une liste d'arguments au constructeur, cette liste apparaît dans l'en–tête de la fonction constructeur, précédée de l'opérateur `:` .

Initialisation des objets dans les constructeurs(2)

```
#include "C.h"                    // Fichier C.cpp (version 2)

C::C(void) :  _n(0)                // constructeur par défaut
{
}
C::C(int n) :  _n(n)      // constructeur avec initialisation
{
}
C::C(const C& c) :  _n(c._n)    // constructeur par recopie :
{                                // clônage
}
```

Le destructeur

Le destructeur est une fonction membre dont le nom est le même que le nom de la classe précédé d'un tilda (~).
Il n'a pas de type, ni d'argument.

Rôle du destructeur

Le destructeur est appelé *avant* la destruction de l'objet.
Il a pour rôle de libérer la mémoire des attributs créés par allocation dynamique.

Exemples de destructeurs (1)

```
#ifndef D_H                                  // Fichier D.h
#define D_H

class D {
 public :
            D(int n = 10);  // Valeur par défaut :  10
   virtual ~D(void);
 private:
   int* _array;
   int _n;
};

#endif // D_H
```


Exemples de destructeurs (2)

```
#include "D.h"                          // Fichier D.cpp

D::D(int n) :  _array(new int[n]), _n(n)
{
  // Rien dans le constructeur car,
  // _array = new int[n]; et _n = n; déjà fait !
}

D::~D(void)
{
 delete [] _array;
}
```


Exemples de destructeur (3): Utilisation

```
#include "D.h"                                        // mainD.cpp

int main(void)
{
 D d(2);
 D* dd = new D;     // Valeur par défaut, comme :  D* dd = new D(10);
 ...
 delete dd;      // Destructeur appelé pour la destruction de
                // l'objet pointé par dd (LORS DU 'delete')
 return 0;
}       // Destructeur appelé pour la destruction de l'objet d
      // (AUTOMATIQUE A LA SORTIE DU BLOC)
```

```
$ g++ D.cpp mainD.cpp                    # un makefile, c'est mieux !
$ ./a.out
```

Ecriture condensée (1)

Dans le fichier d'en–tête .h,
il est possible de mettre à l'intérieur de

```
class ClassName
{
 ...
};
```

le code de fonctions membres.

Attention,
le code de ces fonctions ne doit être présent qu'une seule fois !

Ecriture condensée (2) : le .h

```
#ifndef D_H                              // Fichier Dreduit.h
#define D_H

class D {
 public :
           D(int n = 10) :  _array(new int[n]), _n(n) {}
   virtual ~D(void) { delete [] _array; }
 private:
   int* _array;
   int _n;
};

#endif // D_H
```


Ecriture condensée (3) : Utilisation

```
#include "Dreduit.h"                       // mainDreduit.cpp

int main(void)
{
 D d(2);

 ...

 return 0;
}
```

```
$ g++ mainDreduit.cpp              # Ici, pas de Dreduit.cpp !
$ ./a.out                        # Car, tout le code dans Dreduit.h
```


Fonctions membres `inline` (1)

Dans le fichier d'en–tête .h, il est possible de
mettre le code de fonctions membres **`inline`** (voir page 56).

Ainsi, après

```
class ClassName
{
 ...
};
```

0n peut mettre le code des méthodes précédé de **`inline`**.

Attention,
le code de ces fonctions ne doit être présent qu'une seule fois !

Fonctions membres inline (2) : le .h

```
#ifndef D_H                              // Fichier Dinline.h
#define D_H

class D {
 public :
          D(int n = 10);
   virtual ~D(void);
 private:
   int* _array;
   int _n;
};

inline D::D(int n) :  _array(new int[n]), _n(n) {}
inline D::~D(void) { delete [] array; }

#endif // D_H
```


Fonctions membres inline (3) : Utilisation

```
#include "Dinline.h"                              // mainDinline.cpp

int main(void)
{
 D d(2);

 ...

 return 0;
}
```

```
$ g++ mainDinline.cpp              # Ici, pas de Dinline.cpp !
$ ./a.out                     # Car, tout le code dans Dinline.h
```


Exemple : la classe Point (1)

```
// Fichier Point.h
#ifndef POINT_H
#define POINT_H
class Point {
 public:
          Point(int x = 0, int y = 0) :  _x(x), _y(y) {}
          Point(const Point& p) :  _x(p._x), _y(p._y) {}
  virtual ~Point(void) {}
  int  getx(void) const;
  int  gety(void) const;
  void display(void) const;
  void set(int x, int y);
  void move(int dx, int dy);
 private:
  int _x;
  int _y;
};
#endif
```


Exemple : la classe Point (2)

```
// Fichier Point.cpp
#include <iostream>
#include "Point.h"
using namespace std;

int Point::getx(void) const
{ return _x; }

int Point::gety(void) const
{ return _y; }

void Point::display(void) const
{ cout << '(' << _x << ',' << _y << ')'; }

void Point::set(int x, int y)
{ _x = x; _y = y; }

void Point::move(int dx, int dy)
{ _x += dx; _y += dy; }
```

Pointeur sur fonction membre

```
#include "Point.h"                                    // main.cpp

int main(void)
{
 Point a(5,2);
 Point* b = new Point(1,5);

 void (Point::*fct)(void) const;  // Rappel: Point::display est const !
 fct = &Point::display;

 (a.*fct)();
 (b->*fct)();

 delete b;
 return 0;
}
```

```
$ g++ Point.cpp main.cpp  # un makefile, c'est mieux !
$ ./a.out
(5,2)(1,5)
```


Fonctions membres statiques

Les fonctions membres `static` d'une classe sont des fonctions membres pouvant être invoquées indépendamment de tout objet de cette classe. Elles ne manipulent que les attributs `static`..!

```
// Point.h
#ifndef POINT_H
#define POINT_H

class Point {
  public:
      ...
      static const char* getClassName(void);
  private:
      ...
      static const char* _className;
};
#endif
```

```
// Point.cpp
const char* Point::_className = "Point";
const char* Point::getClassName(void)
{
  return _className;
}
...
```

```
#include "Point.h"                    // main.cpp
...
int main(void)
{
  cout << Point::getClassName() << endl;
  return 0;
}
```


Bonnes pratiques (1)
Conseils pour l'écriture d'une classe

Dans une classe `C`, toujours avoir :

- Un constructeur par défaut (ex: `C(void);`)
  ou ayant des arguments par défaut (ex: `C(int v = 0);`)
- Un constructeur par recopie (ex: `C(const C& c);`)
- Un destructeur (ex: `virtual ~C(void);`)
  Le mot clef `virtual`, ...
  c'est pour l'héritage, voir plus loin !

Bonnes pratiques (2)
Conseils pour l'écriture d'une classe

Mais aussi :

- Un opérateur d'affectation (ex: `C& operator=(const C& c);`)
  `operator=` est une fonction membre...
- Un opérateur d'affichage
  (ex: `friend ostream& operator<<(ostream& os, const C& c);`)
  Le mot clef `friend`, ...
  indique que `operator<<` est une fonction amie...voir plus loin !
  Remarque : une fonction amie d'une classe peut accéder aux parties `private/protected` de cette classe !

Bonnes pratiques (3)
Conseils pour l'écriture d'une classe

On peut également ajouter :

- `friend bool operator==(const C& c1, const C& c2);` et
- `friend bool operator!=(const C& c1, const C& c2);`

Deux fonctions amies pour tester l'égalité de 2 objets de la classe :

```
bool operator==(const C& c1, const C& c2)
{
 // A faire ...  test de chaque attribut
}
bool operator!=(const C& c1, const C& c2)
{
 return !(c1==c2);
}
```


Bonnes pratiques : un exemple (1)

```
#ifndef C_H             // Fichier C.h
#define C_H

#include <iostream>
using namespace std;

class C
{
   friend ostream& operator<<(ostream& os, const C& c);

 public :

   // Allocateurs/Desallocateurs

            C(int v = 0);
            C(const C& c);
            C& operator=(const C& c);
   virtual ~C(void);
```

à suivre ...

Bonnes pratiques : un exemple (2)

```
                      // Fichier C.h (suite)
   // Comparaisons

   friend  bool operator==(const C& c1, const C& c2);
   friend  bool operator!=(const C& c1, const C& c2);

 private :

   // Attributs

           int  _v;   // Un int
           int* _pv;  // Un pointeur sur int (il faudra faire un 'new int')

   // Methodes privees d'allocation/desallocation

           void _copy(const C& c);
           void _destroy(void);
};

#endif // C_H
```

fin `C.h`

Bonnes pratiques : un exemple (3)

```
#include "C.h"          // Fichier C.cpp

C::C(int v) : _v(v), _pv(new int)                              // Constructeur
{
 *_pv = _v*2;
}
C::C(const C& c)                                    // Constructeur par recopie
{
 _copy(c);
}
C& C::operator=(const C& c)                                     // Affectation
{
 if (this != &c) { _destroy(); _copy(c); }
 return *this;
}
C::~C(void)                                                     // Destructeur
{
 _destroy();
}
```

à suivre ...

Bonnes pratiques : un exemple (4)

```
                    // Fichier C.cpp (suite)
bool operator==(const C& c1, const C& c2)              // Comparaisons ==
{
 if (c1._v != c2._v)     return false;
 if (*c1._pv != *c2._pv) return false;

 return true;
}
bool operator!=(const C& c1, const C& c2)              // Comparaisons !=
{
 return !(c1==c2);
}
ostream& operator<<(ostream& os, const C& c)                // Affichage
{
 os << "(" << c._v << " " << *c._pv << ")";
 return os;
}
```

à suivre ...

Bonnes pratiques : un exemple (5)

```
                    // Fichier C.cpp (suite)
void C::_copy(const C& c)                  // Methode privee d'allocation
{
  _pv = new int;
   _v = c._v;
 *_pv = *c._pv;
}
void C::_destroy(void)                 // Methode privee de desallocation
{
 if (_pv) delete _pv;
}
```

fin `C.cpp`

Remarque :

- `_copy` pour : `C(const C& c)` et `C& operator=(const C& c)`
- `_destroy` pour : `~C(void)` et `C& operator=(const C& c)`

Bonnes pratiques : un exemple (5)

```
#include <iostream>                 // Fichier main.cpp
#include "C.h"

using namespace std;

int main(void)
{
 C a(10), b;
 C *c = new C(a);

 cout << a << " " << b << " " << *c << endl;

 if (a==b) cout << "a==b" << endl;
 else cout << "a!=b" << endl;
 b = a;

 cout << a << " " << b << " " << *c << endl;

 delete c;
 return 0;
}
```

```
// # un makefile, c'est mieux !
$ g++ C.cpp main.cpp
$ ./a.out
(10 20) (0 0) (10 20)
a!=b
(10 20) (10 20) (10 20)
$
```


Les fonctions amies

■ Les fonctions amies

1. **Les fonctions `friend`**
2. **Fonction amie indépendante**
3. **Fonction amie de plusieurs classes**
4. **Fonction amie membre d'une autre classe**
5. **Classe amie**
6. **Exemple de fonction amie**

Les fonctions `friend`

- Normalement, seuls les membres d'une classe ont accès aux parties `protected`/`private` de cette classe.
- Le mécanisme des "amies" permet à une fonction non membre, voire même à une autre classe de rompre le mécanisme d'encapsulation.
- Ces fonctions (ou ces classes) devront être déclarées `friend` dans la classe en question.

Fonction amie indépendante

```
class A {
    friend int f(int n, const A& a);   /* Fonction amie */
  private :                            /* de la classe A */
    int _a;
};

int f(int n, const A& a) { return n + a._a; }
```


Fonction amie de plusieurs classes

```
class A;          // Le compilateur doit savoir que A est une classe :  F
class B;          // Le compilateur doit savoir que B est une classe :  V

class A {
    friend int f(const A& a, const B& b);             /* Fonction amie */
  private :                                          /* de la classe A */
    int _a;
};

class B {
    friend int f(const A& a, const B& b);             /* Fonction amie */
  private :                                          /* de la classe B */
    int _b;
};

int f(const A& a, const B& b) { return a._a + b._b; }
```


Fonction amie membre d'une autre classe

```
class A; // Le compilateur doit savoir que A est une classe :  V
class B; // Le compilateur doit savoir que B est une classe :  F

class B {
 public :
   int f(int n, const A& a);
};

class A {
   friend int B::f(int n, const A& a); /* La methode f de la */
 private :                             /* classe B est amie  */
   int _a;                             /* de la classe A     */
};

int B::f(int n, const A& a) { return n + a._a; }
```

Classe amie

```
class A;  // Le compilateur doit savoir que A est une classe:  F
class B; // Le compilateur doit savoir que B est une classe:  V?

class A {
    friend class B;             /* Toute la classe B est amie */
  private :                     /* de la classe A             */
    int _a;
};

class B {
  public :
    int f(const A& a) { return 3 + a._a; }
    int g(const A& a) { return 3 * a._a; }
};
```


Exemple de fonction amie (version 1)

```
#include <iostream>
using namespace std;
class Point {
    friend void display(const Point& p);
  public :
    // Constructeur, destructeur, etc ...
  private :
    // Attributs
    int _x;
    int _y;
};

void display(const Point& p)      // Exemple d'appel :  display(unPoint);
{
  cout << '(' << p._x << ',' << p._y << ')';
}
```


Exemple de fonction amie (version 2)

```
#include <iostream>
using namespace std;
class Point {
    friend ostream& operator<<(ostream& os, const Point& p);
  public :
    // Constructeur, destructeur, etc ...
  private :
    // Attributs
    int _x;
    int _y;
};
ostream& operator<<(ostream& os, const Point& p)     // Exemple d'appel :
{                                                   // cout << unPoint << endl;
  os << '(' << p._x << ',' << p._y << ')';
  return os;
}
```


Les classes génériques

- **Les classes génériques**
  1. **Les types génériques**
  2. **Exemple de classe générique**
  3. **Utilisation d'une classe générique**

Les types génériques

```
template <class TYPE_FORMEL>
class ClasseGenerique {
 public :
           ClasseGenerique(void);
           ClasseGenerique(const ClasseGenerique<TYPE_FORMEL>& c);
   virtual ~ClasseGenerique(void);
           const TYPE_FORMEL& g(void) const;
           void f(const TYPE_FORMEL& t);
 private :
      TYPE_FORMEL _attribut;
};

ClasseGenerique<int> instanceDeClasseGenerique1;
ClasseGenerique<double> instanceDeClasseGenerique2;
```


Exemple de classe générique : `Stack<T>`

```
#ifndef STACK_H
#define STACK_H

const int MAX = 10;

template <class T>
class Stack {                                    // Fichier Stack.h
 public :    Stack(void);
   virtual ~Stack(void);
   bool empty(void) const;
   bool full(void) const;
   int size(void) const;
   const T& top(void) const;
   void push(const T& t);
   T pop(void);
 private :
   T _stack[MAX];
   int _size;
};
#endif // STACK_H
```

Remarque
On peut aussi faire :
- L'injection dans un flot de sortie (`operator<<`)
- Le constructeur par recopie
- L'opérateur d'affectation (`operator=`)
- ...

`Stack<T>` (1)

```
#include <assert.h>                    // Fichier Stack.cpp
#include "Stack.h"

template <class T>
Stack<T>::Stack(void) :  _size(0) {}

template <class T>
Stack<T>::~Stack(void) {}

template <class T>
bool Stack<T>::empty(void) const { return _size == 0; }

template <class T>
bool Stack<T>::full(void) const { return _size == MAX; }
```

à suivre ...

`Stack<T>` (2)

```
                                  // Fichier Stack.cpp (suite)

template <class T>
int Stack<T>::size(void) const { return _size; }

template <class T>
const T& Stack<T>::top(void) const {
  assert(!empty());
  return _stack[_size - 1];
}
```

à suivre ...

Stack<T> (3)

```
                              // Fichier Stack.cpp (suite)

template <class T>
void Stack<T>::push(const T& t) {
   assert(!full());
   _stack[_size++] = t;
}

template <class T>
T Stack<T>::pop(void) {
   assert(!empty());
   return _stack[--_size];
}
```

fin Stack.cpp

Utilisation d'une classe générique : Stack<T>

```
#include <iostream>
#include "Stack.h"
#include "Stack.cpp"   // Pour g++, il faut include le .cpp !

using namespace std;

int main(void)                              // Fichier main.cpp
{
  Stack<int> s1;
  s1.push(1); s1.push(2); s1.push(3);
  while(!s1.empty()) { cout << s1.pop() << endl;}

  Stack<double>* s2 = new Stack<double>;
  s2->push(1.0); s2->push(2.56); s2->push(3.14);
  while(!s2->empty()) { cout << s2->pop() << endl; }
  delete s2;

  return 0;
}
```


Utilisation d'une classe générique : makefile

```
main: Stack.o main.o
        g++ Stack.o main.o -o main

Stack.o: Stack.h Stack.cpp
        g++ -c Stack.cpp

main.o: Stack.h Stack.cpp main.cpp
        g++ -c main.cpp
```

```
$ make
g++ -c Stack.cpp
g++ -c main.cpp
g++ Stack.o main.o -o main
$ ./main
3
2
1
3.14
2.56
1
$
```


Utilisation d'une classe générique : #include "Stack.cpp" !

Pour éviter d'inclure le `.cpp`, on peut dans le `.h` (1) :

- Soit, utiliser des fonctions membres `inline` (voir page 132)

```
template <class T>
class Stack {
 public :
   ....
   bool empty(void) const;
   ....
 private :
   ....
};
template <class T>
inline bool Stack<T>::empty(void) const { return _size == 0; }
```

Utilisation d'une classe générique : `#include "Stack.cpp"` !

Pour éviter d'inclure le `.cpp`, on peut dans le `.h` (2) :

- Soit, utiliser l'écriture condensée (voir page 129).

```
template <class T>
class Stack {
 public :
   ....
   bool empty(void) const { return _size == 0; }
   ....
 private :
   ....
};
```


Utilisation d'une classe générique : `#include "Stack.cpp"` !

Normalement, pour éviter d'inclure le `.cpp`, **il faut** dans le `.h` :

- **Utiliser** le mot clef `export`

```
export
template <class T>
class Stack {
 public :
   ....
   bool empty(void) const;
   ....
 private :
   ....
};
```

MAIS ATTENTION : ça dépend du compilateur !

La surdéfinition des opérateurs

■ **La surdéfinition des opérateurs**

1. **Les opérateurs en C++**
2. **Modes de surdéfinition**
3. **Cohérence des opérateurs**
4. **Exemples de surdéfinitions**

Les opérateurs en C++

Les opérateurs prédéfinis du langage C++ sont des fonctions que l'on peut donc surdéfinir pour les types définis par l'utilisateur.

Opérateur ≡ Fonction

$$x = y + z$$

$$\equiv$$

$$\texttt{operator=}(x, \texttt{operator+}(y, z))$$

Opérateurs prédéfinis (1)

`::`	résolution de portée	`classe::membre`
`::`	global	`::nom`
`.`	sélection de membre	`objet.membre`
`->`	sélection de membre	`pointeur->membre`
`[]`	indexation	`pointeur[expr]`
`()`	appel de fonction	`expr(liste expr)`
`()`	constructeur	`type(liste expr)`
`sizeof`	taille d'un objet	`sizeof expr`
`sizeof`	taille d'un type	`sizeof(type)`
`--`	postdécrémentation	`lvalue--`
`--`	prédécrémentation	`--lvalue`

Opérateurs prédéfinis (2)

`++`	postincrémentation	`lvalue++`
`++`	préincrémentation	`++lvalue`
`~`	complément	`~expr`
`!`	négation	`!expr`
`-`	moins unaire	`-expr`
`+`	plus unaire	`+expr`
`&`	adresse de	`&lvalue`
`*`	déréférence	`*expr`
`new`	allocation mémoire	`new type`
`delete`	désallocation mémoire	`delete pointeur`
`delete[]`	désallocation tableau	`delete [] pointeur`
`()`	conversion de type	`(type)expr`

Opérateurs prédéfinis (3)

`.*`	sélection de membre	`objet.*pointeur`
`->*`	sélection de membre	`pointeur->*pointeur`
`*`	multiplication	`expr * expr`
`/`	division	`expr / expr`
`%`	modulo	`expr % expr`
`+`	addition	`expr + expr`
`-`	soustraction	`expr - expr`
`<<`	décalage à gauche	`expr << expr`
`>>`	décalage à droite	`expr >> expr`

Opérateurs prédéfinis (4)

`<`	plus petit	`expr < expr`
`<=`	plus petit ou égal	`expr <= expr`
`>`	plus grand	`expr > expr`
`>=`	plus grand ou égal	`expr >= expr`
`==`	égalité	`expr == expr`
`!=`	non égalité	`expr != expr`
`&`	ET bit–à–bit	`expr & expr`
`^`	OU exclusif bit–à–bit	`expr ^ expr`
`|`	OU inclusif bit–à–bit	`expr | expr`
`&&`	ET logique	`expr && expr`
`||`	OU logique	`expr || expr`
`? :`	expression conditionnelle	`expr? expr : expr`

Opérateurs prédéfinis (5)

=	affectation	lvalue = expr
*=	multiplication et affectation	lvalue *= expr
/=	division et affectation	lvalue /= expr
%=	modulo et affectation	lvalue %= expr
+=	addition et affectation	lvalue += expr
-=	soustraction et affectation	lvalue -= expr
<<=	décalage et affectation	lvalue <<= expr
>>=	décalage et affectation	lvalue >>= expr
&=	ET et affectation	lvalue &= expr
|=	OU inclusif et affectation	lvalue |= expr
^=	OU exclusif et affectation	lvalue ^= expr
,	séquence (virgule)	expr , expr

Associativité et priorités des opérateurs

Associativité : Les opérateurs unaires et les opérateurs d'affectation sont associatifs à droite; tous les autres sont associatifs à gauche.

- `*p++` ≡ `*(p++)`
- `a = b = c` ≡ `a = (b = c)`
- `a + b + c` ≡ `(a + b) + c`

Priorité : Dans les tableaux précédents, chaque encadré contient les opérateurs de même précédence (priorité). Un opérateur a une précédence plus élevée que ceux des encadrés inférieurs.

Propriétés des opérateurs

- La priorité, l'associativité et l'arité des opérateurs prédéfinis ne peuvent pas être modifiées.
- Il n'est pas possible de créer de nouveaux opérateurs.
- Les opérateurs d'affectation (`=`) et d'adressage (`&`) sont implicitement surdéfinis par le compilateur.
- Il n'est pas possible de modifier la définition des opérateurs des types prédéfinis.

Opérateurs surdéfinissables

+	-	*	/
%	^	&	|
~	!	=	<
>	+=	-=	*=
/=	%=	^=	&=
|=	<<	>>	>>=
<<=	==	!=	<=
>=	&&	||	++
--	,	->*	->
()	[]	new	delete

Modes de surdéfinitions

- On ne peut pas définir de nouveaux opérateurs. On ne peut que surdéfinir les opérateurs existants.
- On ne peut modifier ni l'arité, ni l'associativité, ni la priorité d'un opérateur.
- Un opérateur peut être une fonction membre ou une fonction non membre (amie ou non).

  Si `op` est **membre** : `a op b` ≡ `a.op(b)`

  Si `op` n'est pas **membre** : `a op b` ≡ `op(a,b)`

Membre ou non membre ? (1)

- Une fonction membre possède implicitement comme premier opérande le pointeur d'autoréférence `this`.

```
type1 Classe::membre( /*Classe* this,*/ type2,type3)
```

- Si un opérateur requiert un opérande gauche d'une autre classe, il ne peut pas être une fonction membre.

  Exemple: `Classe c;  cout << c;   // ≡ operator<<(cout,c);`

- Un opérateur non membre ayant besoin d'accéder aux membres non publics d'une classe, doit être une fonction `friend`.

  Exemple:

```
friend ostream& operator<<(ostream& os, const Classe& c);
```


Membre ou non membre ? (2)

- Les opérateurs `()`, `=`, `->` et `[]` doivent être des fonctions membres non statiques pour garantir que leur premier opérande soit une "lvalue".
- Les opérateurs commutatifs doivent être réalisés par des fonctions non membres.

Exemples d'opérateurs membres (1)

```
Point& Point::operator+=(const Point& p) {            // Membre.
  _x += p._x; _y += p._y;
  return *this;
}

Point Point::operator+(const Point& p) {              // Membre.
  Point point(*this);
  point += p;
  return point;
}
```

Exemples d'opérateurs membres (2)

```
int Point::operator()(void) {                          // Membre.
  return ((_x * _x) + (_y * _y));
}

ostream& operator<<(ostream& s, const Point& p) {    // Non membre
  return s << '(' << p._x << ',' << p._y << ')';    // mais, amie.
}
```

Utilisation :

```
Point p1, p2(2,3);
cout << p1 + p2 << endl; // (2,3)                 Rappel : p1≡(0,0)
cout << p1 + 2 << endl; // ≡ p1 + Point(2,0)
cout << 2 + p1 << endl; // erreur : + est fonction membre
cout << p2() << endl;   // 13
```


Exemples d'opérateurs non membres (1)

```
Point& Point::operator+=(const Point& p) {                  // Membre.
  _x += p._x; _y += p._y;
  return *this;
}

Point operator+(const Point& p1, const Point& p2) { // Non membre
  Point point(p1);                                   // amie ou non.
  point += p2;                                       // En général, amie.
  return point;
}
```


Exemples d'opérateurs non membres (2)

```
ostream& operator<<(ostream& s, const Point& p) {    // Non membre
  return s << '(' << p._x << ',' << p._y << ')';    // mais, amie.
}
```

Utilisation :

```
Point p(2,3);
cout << p << endl;
cout << p + 2 << endl; // ≡ p + Point(2,0)
cout << 2 + p << endl; // ≡ Point(2,0) + p
```


Cohérence des opérateurs

- Respecter la sémantique habituelle des opérateurs.

  `+` : addition de nombres, de matrices, de chaînes, ...
  `=` : affectation
  `<<` : injection
  ...

- Respecter les relations entre opérateurs

  `==` et `!=`
  `<`, `<=`, `>=` et `>`
  ...

Cohérence des opérateurs == et !=

```
bool Classe::operator==(const Classe& c) const;              // à définir

bool Classe::operator!=(const Classe& c) const {
   return !operator==(c);
}
```

Remarque :

Ici, `operator==` et `operator!=` ont été déclarées comme fonctions membres... on peut choisir de les mettre `friend` !

```
bool operator==(const Classe& c1, const Classe& c2);         // à définir

bool Classe::operator!=(const Classe& c1, const Classe& c2) {
   return !operator==(c1,c2);
}
```


Cohérence des opérateurs <, <=, >= et >

```
bool Classe::operator<(const Classe& c) const;   // à définir

bool Classe::operator<=(const Classe& c) const {
   return !(c < (*this));
}

bool Classe::operator>=(const Classe& c) const {
   return !operator<(c);
}

bool Classe::operator>(const Classe& c) const {
   return c < (*this);
}
```


Cohérence des opérateurs <op>= et <op>

```
// exemple avec *= et *
Classe& Classe::operator*=(const Classe& c);     // à définir

Classe operator*(const Classe& c1, const Classe& c2) {
   Classe c(c1);
   return c *= c2;
}
```


Cohérence des opérateurs ++ et --

```
// préincrémentation   :  ++c
Classe& Classe::operator++(void);                    // à définir

// postincrémentation  :  c++
Classe Classe::operator++(int) {
   Classe c(*this);
   ++(*this);
   return c;
}
```

Cohérence des opérateurs =

```
Classe::Classe(const Classe& c);  // constructeur par recopie
Classe::Classe(const Cousin& c);               // constructeur
                                              // par cousinage

Classe& Classe::operator=(const Classe& c) {    // affectation
   if(this != &c) { /* ... */ }
   return *this;
}
Classe& Classe::operator=(const Cousin& c) {    // affectation
   { /* ... */ }
   return *this;
}
```


Exemples de surdéfinitions : une classe `Array<T>`

Un tableau homogène est une collection d'objets de même type, adressables par un index entier.

- Interface de la classe `Array<T>` (fichier `Array.h`)
- Implémentation de la classe `Array<T>` (fichier `Array.cpp`)
- Test de la classe `Array<T>` (fichier `testArray.cpp`)

Interface de la classe `Array<T>`

```
#ifndef ARRAY_H                                              // Array.h
#define ARRAY_H

#include <iostream>

using namespace std;

template <class T> class Array;                              // Pour les fonctions
template <class T> ostream& operator<<(ostream& s, const Array<T>& a);    // friend template
                                                             // mais aussi <>

template <class T>                              // tableau homogène
class Array {

   friend ostream& operator<< <>(ostream& s, const Array<T>& a);

   public:       /*...*/                               // services
   protected:    /*...*/                             // utilitaires
   private:      /*...*/                   // représentation interne
                                         // et services internes
};

#endif
```


Constructeurs/affectation/destructeur

```
public:
   Array(int capacity = 10);        // Constructeur par défaut
   Array(const Array<T>& t);      // Constructeur par recopie
   Array<T>& operator=(const Array<T>& t);      // Affectation
   Array(const T* CArray, int capacity);
                                 // Constructeur par cousinage
   virtual ~Array(void);                          // Destructeur
```

suite `public` ...

Opérateur de Transtypage

```
public:
   operator T*(void);
```

suite `public` ...

Inspecteurs

```
public:
   int  size(void) const;
   int  capacity(void) const;
   bool empty(void) const;
   bool full(void) const;
   int  has(const T& element) const; // retourne l'index ou -1
```

suite `public` ...

Opérateurs (1)

```
public:
   bool operator==(const Array<T>& a) const;
   bool operator!=(const Array<T>& a) const;
   const T& operator[](int index) const;
   T& operator[](int index);
```

suite `public` ...

Opérateurs (2)

```
public:
   Array<T>& operator+=(const Array<T>& a);
   friend Array<T> operator+ <>(const Array<T>& a1,
                                       const Array<T>& a2);
   Array<T>& operator<<(const T& element);
   Array<T>& operator>>(T& element);
   T operator()(const T& t, T (*f)(const T&,const T&)) const;
```

Remarque : comme `operator+` est une fonction amie `template`, il faut dans `Array.h` (page 191), mettre les mêmes informations que pour `ostream& operator<<(ostream& s, const Array<T>& a);`

⇒ `template <class T>`

```
Array<T> operator+(const Array<T>& a1, const Array<T>& a2);
```

suite `public` ...

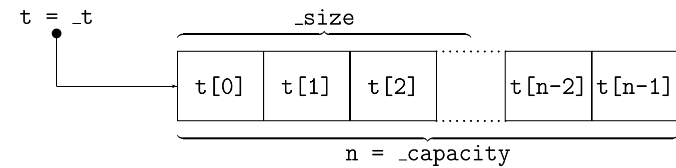
Modifieurs

```
public:
   void insert(const T& element);
   T    remove(void);
```

fin `public`!

Représentation interne d'un tableau

```
private:
   int _size;             // nombre d'éléments dans le tableau
   int _capacity;                        // capacité du tableau
   T* _t;                                // tableau C dynamique
```



suite `private` ...

Services internes d'un tableau

```
private:                              // Utilitaires privés de :
   void _copy(const Array<T>& t);          // - copie,
   void _destroy(void);                    // - désallocation
   void _realloc(int newCapacity);         // - réallocation
```

fin `private` ...!

Implémentation de la classe `Array<T>`

```
#include <assert.h>                                  // Array.cpp
#include "Array.h"
```

à suivre ...

Services internes _copy, _destroy et _realloc (1)

```
// ................................................... _copy(a);
template <class T>
void Array<T>::_copy(const Array<T>& a)
{
   _size = a._size;
   _capacity = a._capacity;
   _t = new T[_capacity];
   for(int i = 0; i < a._size; i++) {
      _t[i] = a._t[i];
   }
}
```

à suivre ...

Services internes _copy, _destroy et _realloc (2)

```
// .................................................. _destroy();
template <class T>
void Array<T>::_destroy(void)
{
   if(_t) delete [] _t;
   _size = 0;
   _capacity = 0;
   _t = NULL;
}
```

à suivre ...

Services internes _copy, _destroy et _realloc (3)

```
// .................................... _realloc(newCapacity);
template <class T>
void Array<T>::_realloc(int newCapacity)
{
   if (_capacity == newCapacity) return;
   T* newt = new T[newCapacity];
   int limit = _size < newCapacity ?  _size :  newCapacity;
   for(int i = 0; i < limit; i++) {
      newt[i] = _t[i];
   }
   _destroy();
   _size = limit;
   _capacity = newCapacity;
   _t = newt;
}
```

à suivre ...

Constructeur par défaut

```
// ......................................... Array<T> anArray;
// ............................... Array<T> anArray(capacity);
template <class T>
Array<T>::Array(int capacity)
:  _size(0), _capacity(capacity), _t(new T[capacity])
{
   // for(int i = 0; i < _capacity; i++) { _t[i] = T(); }
}
```

Remarque : lors du `new T[capacity]` et pour chaque cases du tableau créé, il y a **appel implicite** au **constructeur par défaut** de la classe `T`.
⇒ ce constructeur par défaut doit exister !

à suivre ...

Constructeur par recopie

```
// ........................................ Array<T> anArray(a);
template <class T>
Array<T>::Array(const Array<T>& a)
{
   _copy(a);
}
```

à suivre ...

Affectation

```
// ................................................ anArray = a;
template <class T>
Array<T>& Array<T>::operator=(const Array<T>& a)
{
   if(this != &a) {
      _destroy();
      _copy(a);
   }
   return *this;
}
```

à suivre ...

Constructeur par cousinage

```
// ......................... Array<T> anArray(CArray,capacity);
template <class T>
Array<T>::Array(const T* CArray, int capacity)
:  _size(0), _capacity(capacity), _t(new T[capacity])
{
   for(int i = 0; i < capacity; i++) {
      insert(CArray[i]);
   }
}
```

à suivre ...

Destructeur

```
template <class T>
Array<T>::~Array(void)
{
   _destroy();
}
```

à suivre ...

Transtypage

```
// .................................. T* CArray = (T*)anArray;
template <class T>
Array<T>::operator T*(void)
{
   return _t;
}
```

à suivre ...

Contenance et contenu (1)

```
// ............................................ anArray.size();
template <class T>
int Array<T>::size(void) const { return _size; }

// ...................................... anArray.capacity();
template <class T>
int Array<T>::capacity(void) const { return _capacity; }

// ......................................... anArray.empty();
template <class T>
bool Array<T>::empty(void) const { return _size == 0; }
```

à suivre ...

Contenance et contenu (2)

```
// ............................................. anArray.full();
template <class T>
bool Array<T>::full(void) const
{
   return _size == _capacity;
}

// ..................................... anArray.has(element);
template <class T>                  // retourne l'index ou -1
int Array<T>::has(const T& element) const
{
   for(int i = 0; i < _size; i++) {
      if(_t[i] == element) return i;
   }
   return -1;
}
```

à suivre ...

Indexation

```
// ............ anArray[index]; // Pour un Array non constant
template <class T>
T& Array<T>::operator[](int index)
{
   assert(index >= 0 && index < _size);
   return _t[index];
}

// ............... anArray[index]; // Pour un Array constant !
template <class T>
const T& Array<T>::operator[](int index) const
{
   assert(index >= 0 && index < _size);
   return _t[index];
}
```

à suivre ...

Egalité

```
// ............................................... anArray == a;
template <class T>
bool Array<T>::operator==(const Array<T>& a) const
{
   if(_size != a._size) return false;
   if(_capacity != a._capacity) return false;
   for(int i = 0; i < _size; i++) {
      if(_t[i] != a._t[i]) return false;
   }
   return true;
}
```

à suivre ...

Non égalité

```
// ............................................... anArray != a;
template <class T>
bool Array<T>::operator!=(const Array<T>& a) const
{
   return !operator==(a);
}
```

à suivre ...

Concaténation (1)

```
// ............................................... anArray += a;
template <class T>
Array<T>& Array<T>::operator+=(const Array<T>& a)
{
   for(int i = 0; i < a._size; i++) {
      insert(a._t[i]);
   }
   return *this;
}
```

à suivre ...

Concaténation (2)

```
// .................................................... a1 + a2;
template <class T>
Array<T> operator+(const Array<T>& a1, const Array<T>& a2)
{
   Array<T> anArray(a1);
   return anArray += a2;
}
```

à suivre ...

Ajout d'un élément

```
// ...................................... anArray << element;
template <class T>
Array<T>& Array<T>::operator<<(const T& element)
{
   insert(element);
   return *this;
}

// .................................. anArray.insert(element);
template <class T>
void Array<T>::insert(const T& element)
{
   if (full()) _realloc(2*_capacity);
   _t[_size++] = element;
}
```

à suivre ...

Retrait d'un élément

```
// ...................................... anArray >> element;
template <class T>
Array<T>& Array<T>::operator>>(T& element)
{
   element = remove();
   return *this;
}

// .................................. anArray.remove(void));
template <class T>
T Array<T>::remove(void)
{
   assert(!empty());
   return _t[--_size];
}
```

à suivre ...

Accumulateur

```
// ............................................ anArray(t,f);
template <class T>
T Array<T>::operator()(const T& t, T (*f)(const T&,const T&))
const
{
   T result = t;
   for(int i = 0; i < _size; i++) {
      result = f(result,_t[i]);
   }
   return result;
}
```

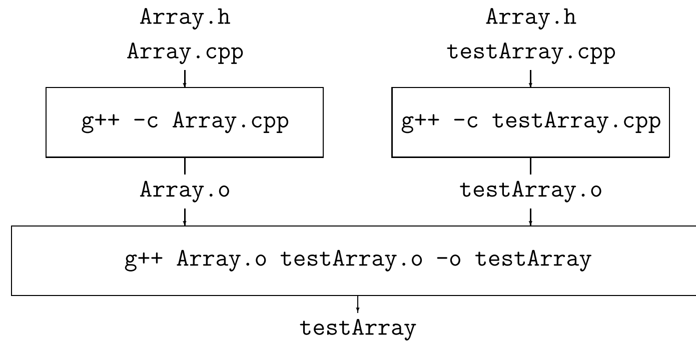
à suivre ...

Injection dans un flot

```
// .................................................... s << a;
template <class T>
ostream& operator<<(ostream& s, const Array<T>& a)
{
   s << '(';
   if(!a.empty()) {
      for(int i = 0; i < a._size - 1; i++) {
         s << a._t[i] << ',';
      }
      s << a._t[a._size - 1];
   }
   return s << ')';
}
```

fin `Array.cpp`

Test de la classe `Array<T>`



Programme de test (1)

```
#include <iostream>                              // testArray.cpp
#include "Array.h"
#include "Array.cpp" // Pour les classes template et g++ !

using namespace std;
int times(const int& x, const int& y) { return x * y; }
int main(void)
{
  int i = 0;
  Array<int> a1;
  Array<int>* a2 = new Array<int>(2);
  Array< Array<int> > a(1);
  cout << "a1 = " << a1 << ',';
  cout << "a2 = " << (*a2) << ',';
  cout << "a = " << a << endl;
```

à suivre ...

Programme de test (2)

```
a1 << 1 << 2 << 3 << 4 << 5;
(*a2) << 6 << 7;
a << a1 << (*a2) << (a1 + (*a2));

cout << "a1 = " << a1 << " :  " << a1(1,times) << endl;
cout << "a2 = " << (*a2) << endl;
cout << "a = " << a << endl;
```

à suivre ...

Programme de test (3)

```
  a1 += (*a2);
  cout << "a1 = " << a1 << endl;
  while(!a1.empty()) { a1 >> i; cout << i << ' '; }
  cout << endl << "a1 = " << a1 << endl;

  while(!a.empty()) { a >> (*a2); cout << (*a2) << ' '; }
  cout << endl << "a = " << a << endl;

  delete a2;

  return 0;
}
```

fin `testArray.cpp`

Exécution du test

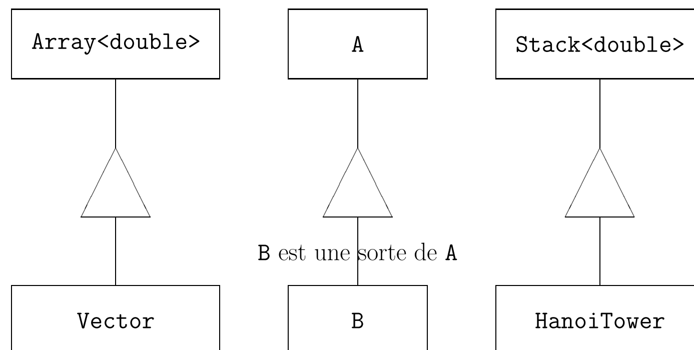
```
$ ./testArray
a1 = () , a2 = () , a = ()
a1 = (1,2,3,4,5) :  120
a2 = (6,7)
a = ((1,2,3,4,5),(6,7),(1,2,3,4,5,6,7))
a1 = (1,2,3,4,5,6,7)
7 6 5 4 3 2 1
a1 = ()
(1,2,3,4,5,6,7) (6,7) (1,2,3,4,5)
a = ()
$
```


L'héritage en C++

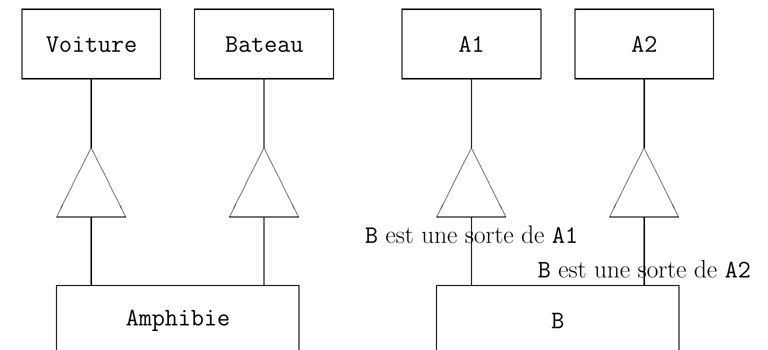
■ **L'héritage en C++**

1. **Mécanismes de base**
2. **Masquage de l'information**
3. **Modes de dérivation**
4. **Envois de messages**
5. **Classes abstraites**

Héritage simple (notation UML)



Héritage multiple (notation UML)



Mécanismes de base

- Une classe `B` (classe dérivée) peut hériter d'une classe `A` (classe de base).

```
class A;

class B : public A { /*...*/ };
```

- Certains attributs et méthodes de la classe de base `A` pourront être utilisées dans la classe dérivée `B` sans réécriture de code.
- Tout se passe comme si les interfaces `public` et `protected` de la classe de base `A` étaient ajoutées à l'interface de la classe dérivée `B`.

Spécialisation(1)

Une surdéfinition des méthodes de la classe de base est possible dans la classe dérivée.

```
// Fichier A.h
// ........... classe de base
class A {
 public :
           A(int a);
   virtual ~A(void);
   int f(void);
 protected :
   int _a;
};
```

```
// Fichier B.h
#include "A.h"
// ........... classe dérivée
class B : public A {
 public :
           B(void);
   virtual ~B(void);
   int f(void);
};
```

Remarque : pas d'obligation de surdéfinir toutes les méthodes.

Spécialisation(2)

```
// Fichier A.cpp
#include "A.h"
A::A(int a) : _a(a)
{
}
A::~A(void)
{
}
int A::f(void)
{
   return _a;
}
```

```
// Fichier B.cpp
#include "B.h"
B::B(void) : A(1)
{
}
B::~B(void)
{
}
int B::f(void)
{
   return _a*10;
}
```

Remarque : Pas de constructeur par défaut pour la classe `A`

⇒ Dans le constructeur de `B`, **appel explicite** au **constructeur** de `A`

Spécialisation(3)

```
// Fichier main.cpp
#include <iostream>
#include "A.h"
#include "B.h"
using namespace std;

int main(void)
{
   A a(17);
   B b;
   cout << a.f() << endl; // 17 (c.a.d. valeur interne _a de a)
   cout << b.f() << endl; // 10 (c.a.d. valeur interne _a de b
   return 0;              //                              * 10)
}
```

Enrichissement (1)

De nouveaux membres peuvent être définis dans la classe dérivée.

```
// Fichier A.h
// ........... classe de base
class A {
 public :
            A(int a);
   virtual ~A(void);
   int f(void);
 private :
   int _a;
};
```

```
// Fichier B.h

#include "A.h"

// ........... classe dérivée
class B : public A {
 public :
            B(int b);
   virtual ~B(void);
   int g(void);
 private :
   int _b;
};
```


Enrichissement (2)

```
// Fichier A.cpp

#include "A.h"

A::A(int a) :  _a(a)
{
}
A::~A(void)
{
}
int A::f(void)
{
  return _a;
}
```

```
// Fichier B.cpp

#include "B.h"

B::B(int b) :  A(1), _b(b)
{
}
B::~B(void)
{
}
int B::g(void)
{
  return _b;
}
```

Remarque : Pas de constructeur par défaut pour la classe `A`
⇒ Dans le constructeur de `B`, **appel explicite** au **constructeur** de `A`

Enrichissement (3)

```
// Fichier main.cpp

#include <iostream>
#include "A.h"
#include "B.h"

using namespace std;

int main(void)
{
  A a(17);
  B b(34);

  cout << a.f() << endl; // 17 (c.a.d.  valeur interne _a de a)
  cout << b.f() << endl; // 1  (c.a.d.  valeur interne _a de b)
  cout << b.g() << endl; // 34 (c.a.d.  valeur interne _b de b)
  return 0;
}
```


Héritage multiple (1)

L'héritage multiple permet à une classe
d'hériter de plusieurs classes de base.

```
// Fichier A.h
// ........ une classe de base
class A {
 public :
            A(void);
   virtual ~A(void);
   void fa(void);
   void g(void);
};
```

```
// Fichier B.h
// ........ une classe de base
class B {
 public :
            B(void);
   virtual ~B(void);
   void fb(void);
   void g(void);
};
```

```
class C : public A, public B {
  .....................
};
```

Héritage multiple (2)

```
// Fichier A.cpp

#include <iostream>
#include "A.h"

using namespace std;

A::A(void) {}
A::~A(void) {}
void A::fa(void)
{
   cout << "A::fa" << endl;
}
void A::g(void)
{
   cout << "A::g" << endl;
}
```

```
// Fichier B.cpp

#include <iostream>
#include "B.h"

using namespace std;

B::B(void) {}
B::~B(void) {}
void B::fb(void)
{
   cout << "B::fb" << endl;
}
void B::g(void)
{
   cout << "B::g" << endl;
}
```


Héritage multiple (3)

```
// Fichier C.h

#include "A.h"
#include "B.h"

// ....... la classe dérivée de A et B
class C : public A, public B {
 public :
             C(void);
    virtual ~C(void);
};
```

```
// Fichier C.cpp

#include <iostream>
#include "C.h"

using namespace std;

C::C(void) {}
C::~C(void) {}
```

Sur un objet de la classe `C` on peut appeler :

- la méthode `fa` de la classe `A` et la méthode `fb` de la classe `B`
- la méthode `g`. Oui mais laquelle? Celle de `A` ou celle de `B` ? ...ambiguïté

Héritage multiple (4)

```
// main.cpp

#include "C.h"

int main(void)
{
   C c;

   c.fa();    // A::fa
   c.fb();    // B::fb
   c.A::g();  // A::g            levée de l'ambiguïté
   c.B::g();  // B::g            levée de l'ambiguïté

   return 0;
}
```


Ordre d'appel des constructeurs et des destructeurs

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

class A {
 public :
             A(void) { cout << "A" << endl; }
    virtual ~A(void) { cout << "~A" << endl; }
};

class B : public A {
 public :
             B(void) { cout << "B" << endl; }
    virtual ~B(void) { cout << "~B" << endl; }
};

int main(void)
{
   B b;
   return 0;
}
```

```
$ ./a.out
A
B
~B
~A
$
```

`// Revient à : B(void) : A() {...}`

En effet, il y a **appel implicite** au **constructeur par défaut** de `A`

Restrictions

Une classe `B` **dérivée** d'une classe de base `A` hérite des différentes méthodes de la classe `A` mais, **n'hérite pas**

- des constructeurs
- du destructeur
- de l'affectation

Masquage de l'information

à l'extérieur de la classe

à l'intérieur de la classe

à l'intérieur d'une classe dérivée

A l'extérieur de la classe

A l'extérieur de la classe, seuls les membres `public` sont accessibles.

```
class A {
 public :
    int f(void);
    int p1;
 protected :
    int _p2;
 private :
    int _p3;
};
```

```
A a;

a.f();                    // ok

cout << a.p1;             // ok

cout << a._p2;        // erreur

cout << a._p3;        // erreur
```


A l'intérieur de la classe

A l'intérieur d'une classe, tous les membres de la classe (`public`, `protected` et `private`) sont accessibles aux autres membres.

```
class A {
 public :
    int f(void);
    int p1;
 protected :
    int _p2;
 private :
    int _p3;
};
```

```
int A::f(void)
{
    return p1 + _p2 + _p3;// ok
}
```

A l'intérieur d'une classe dérivée

Un membre d'une classe dérivée a accès aux membres `public` et `protected` de la classe de base... pas les parties `private` !

```
class B : public A {
 public :
    int f(void);
    int g(void);
};
```

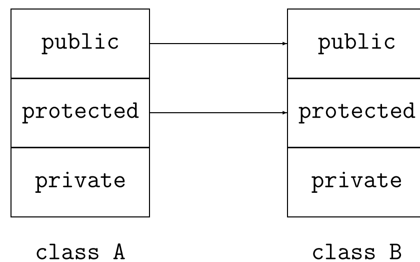
```
int B::f(void)
{
   return p1 + _p2;      // ok
}
int B::g(void)
{
   return _p3;       // erreur
}
```


Modes de dérivation

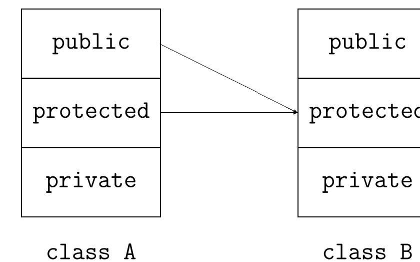
3 modes de dérivation

```
class B : public A { /*...*/ };

class B : protected A { /*...*/ };

{ class B : A { /*...*/ };
{ class B : private A { /*...*/ };
```

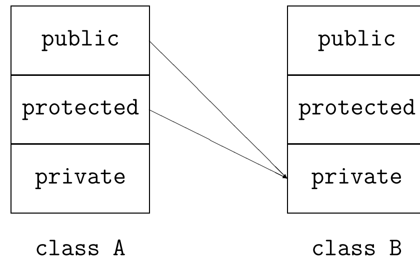

`class B : public A;`



`class B : protected A;`



```
class B : private A;
```



Envois de messages

Envoi de message : mécanisme qui consiste à appliquer une méthode locale à la classe d'un objet, par l'intermédiaire d'une liaison statique ou dynamique.

Liaison statique : la recherche de la méthode associée à un message est effectuée lors de la compilation des envois de message.

Liaison dynamique : la recherche de la méthode associée à un message est effectuée lors de l'exécution de l'envoi de message.

Liaison statique en C++

- L'héritage permet la conversion implicite d'un objet d'une classe dérivée à un objet d'une classe de base.
- Un pointeur sur un objet d'une classe de base peut être redirigé sur une instance d'une classe dérivée, toutefois le type de ce pointeur ayant été défini à la compilation, les méthodes choisies à l'exécution seront celles définies à la compilation.

Exemple de liaison statique (1)

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

class A {
 public :
    A(void) {}
   ~A(void) { cout << "~A" << endl; }       // Pas virtual !
   int f(void) { return 'A'; }              // Code 'A' : 65
};

class B : public A {
 public :
    B(void) {}
   ~B(void) { cout << "~B" << endl; }       // Pas virtual !
   int f(void) { return 'B'; }              // Code 'B' : 66
};
```

Exemple de liaison statique (2)

```
void main(void) {
  A* pa;
  pa = new A;
  cout << pa->f() << endl;                    // 65
  delete pa;                                  // ~A
  pa = new B;
  cout << pa->f() << endl;                    // 65
  delete pa;                                  // ~A
  return 0;
}
```

```
$ a.out
65
~A
65
~A
```


Liaison dynamique en C++

- Les fonctions **virtual** permettent de retarder le choix de la méthode la plus adaptée.
- Un pointeur sur un objet défini dans une classe de base peut, pendant le déroulement du programme, être redirigé sur n'importe quel autre instance des classes dérivées, le choix de la méthode à appliquer sera fait à l'exécution si la méthode dans la classe de base est déclarée **virtual**.

Exemple no 1 de liaison dynamique (1)

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

class A {
 public :
   A(void) {}
  ~A(void) { cout << "~A" << endl; }         // Pas virtual !
  virtual int f(void) { return 'A'; }        // Code 'A' : 65
};

class B : public A {
 public :
   B(void) {}
  ~B(void) { cout << "~B" << endl; }         // Pas virtual !
  virtual int f(void) { return 'B'; }        // Code 'B' : 66
};
```


Exemple no 1 de liaison dynamique (2)

```
void main(void) {
  A* pa;
  pa = new A;
  cout << pa->f() << endl;                    // 65
  delete pa;                                  // ~A
  pa = new B;
  cout << pa->f() << endl;                    // 66
  delete pa;                                  // ~A
  return 0;
}
```

```
$ a.out
65
~A
66
~A
```

Exemple no 2 de liaison dynamique (1)

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

class A {
 public :
          A(void) {}
  virtual ~A(void) { cout << "~A" << endl; }     // virtual !
  virtual int f(void) { return 'A'; }       // Code 'A' : 65
};

class B : public A {
 public :
          B(void) {}
  virtual ~B(void) { cout << "~B" << endl; }     // virtual !
  virtual int f(void) { return 'B'; }       // Code 'B' : 66
};
```


Exemple no 2 de liaison dynamique (2)

```
void main(void) {
  A* pa;
  pa = new A;
  cout << pa->f() << endl;                           // 65
  delete pa;                                         // ~A
  pa = new B;
  cout << pa->f() << endl;                           // 66
  delete pa;                                      // ~B ~A
  return 0;
}
$ a.out
65
~A
66
~B
~A
```


Liaison dynamique et les constructeurs/destructeur (1)

- La liaison dynamique ne fonctionne pas dans les constructeurs.

  C'est trop tôt !

  Un constructeur ne peut pas être déclaré `virtual`.
- La liaison dynamique ne fonctionne pas dans le destructeur.

  C'est trop tard !

  Un destructeur peut être déclaré `virtual` **(recommandé !)**

Liaison dynamique et les constructeurs/destructeur (2)

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

class A {
 public :
  virtual int f(void) { return 'A'; }       // Code 'A' : 65
          A(void) { cout << " A: " << f() << endl; }
  virtual ~A(void) { cout << "~A: " << f() << endl; }
};

class B : public A {
 public :
  virtual int f(void) { return 'B'; }       // Code 'B' : 66
          B(void) { cout << " B: " << f() << endl; }
  virtual ~B(void) { cout << "~B: " << f() << endl; }
};
```

Liaison dynamique et les constructeurs/destructeur (3)

```
void main(void) {
   A* pa;
   pa = new A;                                   // A: 65
   delete pa;                                    // ~A: 65
   pa = new B;                             // A: 65 B: 66
   delete pa;                            // ~B: 66 ~A: 65
}

$ a.out
 A: 65
~A: 65
 A: 65
 B: 66
~B: 66
~A: 65
```


Classe abstraite

Une fonction membre `virtual` à valeur nulle (`= 0`) oblige les classes dérivées à définir cette fonction.

```
class Abstraite {
 public :
   virtual int f(int n) = 0;
};
class Concrete :  public Abstraite {
 public :
   virtual int f(int n) { return n * n; }
};
```

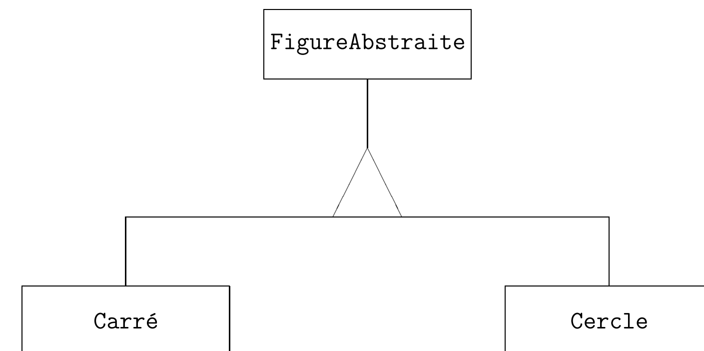
Une classe possédant au moins une fonction virtuelle pure est une classe abstraite, il ne peut exister d'instance de cette classe.

Rôle d'une classe abstraite

- Une classe abstraite permet de définir une interface commune à plusieurs implémentations différentes.
- Elle factorise le Type Abstrait de Données (TAD).

Remarque :
Dans une méthode d'une classe abstraite, il est possible d'appeler une méthode virtuelle pure de cette même classe abstraite..!

Une famille de figures géométriques (notation UML)



FigureAbtraite.h

Interface de FigureAbtraite

```
#ifndef FIGUREABSTRAITE_H                              // FigureAbtraite.h
#define FIGUREABSTRAITE_H

class FigureAbstraite
{
 public:
            FigureAbstraite(double val = 0.0);
   virtual ~FigureAbstraite(void);
   virtual  double aire(void) = 0; // Méthode virtuelle pure
   virtual  void printAire(void); // printAire appelle aire();

 protected:
   double _val;      // Pour un cercle, _val représente le rayon
                     // Pour un carre, _val représente le côté
};

#endif
```


FigureAbtraite.cpp

Implémentation de FigureAbtraite

```
#include <iostream>
#include "FigureAbstraite.h"                 // FigureAbtraite.cpp
using namespace std;

FigureAbstraite::FigureAbstraite(double val) :  _val(val)
{
}

void FigureAbstraite::printAire(void)
{
   cout << aire();
}

FigureAbstraite::~FigureAbstraite(void)
{
}
```


Carre.h

Interface de Carre

```
#ifndef CARRE_H                                                  // Carre.h
#define CARRE_H

#include "FigureAbstraite.h"

class Carre :  public FigureAbstraite
{
 public:
            Carre(double val = 0.0);
   virtual ~Carre(void);
   virtual  double aire(void);
};

#endif
```


Carre.cpp

Implémentation de Carre

```
#include "Carre.h"                                              // Carre.cpp

Carre::Carre(double val) :  FigureAbstraite(val)
{
}

Carre::~Carre(void)
{
}

double Carre::aire(void)
{
   return _val * _val;
}
```

Interface de Cercle

```
#ifndef CERCLE_H                                              // Cercle.h
#define CERCLE_H

#include "FigureAbstraite.h"

class Cercle : public FigureAbstraite
{
 public:
             Cercle(double val = 0.0);
   virtual ~Cercle(void);
   virtual  double aire(void);
};

#endif
```


Implémentation de Cercle

```
#include <math.h>                                            // Cercle.cpp
#include "Cercle.h"

Cercle::Cercle(double val) : FigureAbstraite(val)
{
}

Cercle::~Cercle(void)
{
}

double Cercle::aire(void)
{
  return M_PI * _val * _val;
}
```


Programme de test figures abstraites/figures concrètes

```
#include <iostream>                                           // main.cpp

#include "Carre.h"
#include "Cercle.h"

using namespace std;

int main(void)
{ // FigureAbstraite fa(10.0); // ERREUR car classe abstraite !
  Carre carre(10.0);
  FigureAbstraite *cercle= new Cercle(10.0);

  cout << "Aire de Carre :"; carre.printAire();   cout << endl;
  cout << "Aire de Cercle:"; cercle->printAire(); cout << endl;

  delete cercle;
  return 0;
}
```


Type Concret de Données

- **Type concret de données**
  1. **Du TAD au TCD**
  2. **Exemple de TCD**
  3. `class Base<T1,T2>;`
  4. `class Derived<T1,T2,T3> : public Base<T1,T2>;`

Du TAD au TCD

Type Abstrait de Données

(file d'attente, dictionnaire, magnétophone, ...)

+

Choix d'une représentation interne

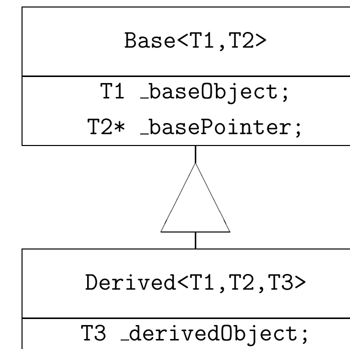
(tableau, liste, arbre, ...)

+

Implémentation dans un langage donné

(Ada, C, C++, Lisp, Prolog, ...)

Exemple de TCD



STL: Standard Template Library

- **Classes génériques de conteneurs:**
  - **vector**
  - **set**
  - **stack**
  - **list**
  - **map**
  - **...**

https://cplusplus.com/reference/stl

Exemple 1: le conteneur vector

```
#include <iostream>                                        // ex1.cpp
#include <vector>
using namespace std;
int main(void)          // Sortie ecran:  0 1 2 3 4 5 6 7 8 9
{ vector<int> v;
  int i;
  for(i=0;i<10;i++){
   v.push_back(i);
  }

  for(i=0;v.size();i++){
   cout << v[i] << endl;    // vector:  operator[] disponible !
  }
  return 0;
}
```

Exemple 2: le conteneur vector

```
#include <iostream>                                    // ex2.cpp
#include <vector>
using namespace std;
int main(void)          // Sortie ecran:  0 1 2 3 4 5 6 7 8 9
{
  vector<int> v;
  int i;
  for(i=0;i<10;i++){
   v.push_back(i);
  }
  vector<int>::iterator it;
  for(it=v.begin();it!=v.end();it++){
   cout << *it << endl;      // Un iterator peut etre vu
  }                          // comme un pointeur que l'on
  return 0;                  // deplace le long du conteneur
}
```


Exemple 3: le conteneur vector

```
#include <iostream>                                    // ex3.cpp
#include <vector>
using namespace std;
int main(void)          // Sortie ecran:  9 8 7 6 5 4 3 2 1 0
{
  vector<int> v;
  int i;
  for(i=0;i<10;i++){
   v.push_back(i);
  }
  vector<int>::reverse_iterator rit;
  for(rit=v.rbegin();rit!=v.rend();rit++){
   cout << *rit << endl;      // Un reverse_iterator peut etre vu
  }                           // comme un pointeur que l'on
  return 0;                   // deplace en sens inverse
}                             // le long du conteneur
```


STL: Standard Template Library

- **Extrêmement pratique pour manipuler des conteneurs !**
  - vector
  - set
  - stack
  - list
  - map
  - ...

https://cplusplus.com/reference/stl

C++ Moderne

- **C++ Moderne: version 2011 (C++11) et plus (C++14, C++17, C++20, etc.): correction de bugs et ajout de fonctionnalités!**
  - auto: déduction automatique de type
  - unique_ptr, shared_ptr, weak_ptr
    $\Longrightarrow$ smart pointer (vérification bornes, libération automatique, etc.)
  - des lambda fonctions, introspection
  - des boucles for un peu différentes
  - ...

http://sdz.tdct.org/sdz/introduction-a-c-2011-c-0x.html

http://tvaira.free.fr/dev/cours/c++-moderne.html

for_each et vector

Exemple 1: le conteneur vector & C++11

```
#include <iostream>                                    // ex1.cpp
#include <vector>              // g++ -std=c++17 ex1.cpp -o ex1
using namespace std;
void afficheElement(int element) { cout << element << endl; }

int main(void)          // Sortie ecran:  0 1 2 3 4 5 6 7 8 9
{                       //                9 8 7 6 5 4 3 2 1 0
   vector<int> v = {0,1,2,3,4,5,6,7,8,9}; // Init ok en C++11

   for_each(v.begin(),v.end(),afficheElement);

   for_each(v.rbegin(),v.rend(),[](int element)
                            { cout << element << endl; }
           ) ;
   return 0;
}
```


for et for_each

Exemple 2: tableau & C++11

```
#include <iostream>                                    // ex2.cpp
                               // g++ -std=c++17 ex2.cpp -o ex2
using namespace std;
void afficheElement(int element) { cout << element << endl; }

int main(void)          // Sortie ecran:  0 1 2 3 4 5 6 7 8 9
{
   int t[10] = {0,1,2,3,4,5,6,7,8,9};
   for(int x :  t) // A chaque tour, x:  copie d'un élément
   {
    x++; // Aucun effet sur le tableau t !
   }
   for_each(t,t+10,afficheElement); // Affichage
   return 0;
}
```


for et tableaux

Exemple 3: tableau & C++11

```
#include <iostream>                                    // ex3.cpp
using namespace std;           // g++ -std=c++17 ex3.cpp -o ex3

int main(void)          // Sortie ecran:  1 2 3 4 5 6 7 8 9 10
{
   int t[10] = {0,1,2,3,4,5,6,7,8,9};
   for(int& x :  t) { // A chaque tour, x:  référence un élément
    x++; // +1 sur chaque élément !
   }

   for(const int& x :  t) { // A chaque tour, x référence constante
    cout << x << endl;
   }
   return 0;
}
```


for et tableaux

Exemple 3: tableau & C++11

```
#include <iostream>                                    // ex3.cpp
using namespace std;           // g++ -std=c++17 ex3.cpp -o ex3

int main(void)          // Sortie ecran:  1 2 3 4 5 6 7 8 9 10
{
   int t[10] = {0,1,2,3,4,5,6,7,8,9};
   for(int& x :  t) { // A chaque tour, x:  référence un élément
    x++; // +1 sur chaque élément !
   }

   for(const int& x :  t) { // A chaque tour, x référence constante
    cout << x << endl;
   }
   return 0;
}
```

Exemple 4: map et auto & C++11

```
#include <iostream>                                   // ex4.cpp
#include <map>                      // g++ -std=c++17 ex4.cpp -o ex4
#include <string>
using namespace std;

int main(void)       // 2 x Sortie ecran :  un=1 deux=2 trois=3
{
  map<int,string> nombres={ {1,"un"}, {2,"deux"}, {3,"trois"} };

  for(pair<int,string> n:  nombre) {
   cout << n.first << "=" << n.second;
  }

  for(auto n:  nombre) { // auto:  déduction automatique de type
   cout << n.first << "=" << n.second;
  }

  return 0;
}
```


Bibliographie / Sites web

Ellis M.J., Stroustrup B. *The annoted C++ Reference Manual*, Addison–Wesley, 1990
International Thomson Publishing, 1997

Stroustrup B. *Le langage C++*, Addison–Wesley, 1992

https://www.cplusplus.com

http://casteyde.christian.free.fr/cpp/cours

Et le C++ Moderne... non abordé dans ce cours :-(

http://tvaira.free.fr/dev/cours/c++-moderne.html

http://sdz.tdct.org/sdz/introduction-a-c-2011-c-0x.html