Pointeur

- Modèle de mémoire
  - adresses continues et croissantes
- Opérateur adresse
  - identique en C et C++
  - Syntaxe : &

Adresses	Mémoire
...	...
000012	...
000008	….
000004	int b = 4
000000	int a = 3

Pointeur : un premier exemple

```
#include <iostream>
using namespace std;
int toto, tutu, titi, tata ;
void f( int x ) {  cout <<  "x vaut " << x << endl; }
int main( int argc, char** argv )
{
 int i, j, k ;
 cout << "f() @ "  << (long) &f << endl ;
 cout << "toto @ " << (long) &toto << endl ;
 cout << "tutu @ " << (long) &tutu << endl ;
 cout << "titi @ " << (long) &titi << endl ;
 cout << "tata @ " << (long) &tata << endl ;
 cout << "i @ "    << (long) &i << endl ;
 cout << "j @ "    << (long) &j << endl ;
 cout << "k @ "    << (long) &k << endl ;
 return 0 ;
}
```

Pointeur : un premier exemple

```
#include <iostream>
using namespace std;
int toto, tutu, titi, tata ;
void f( int x ) {  cout <<  "x vaut " << x << endl; }
int main( int argc, char** argv )
{
 int i, j, k ;
 cout << "f() @ "  << (long) &f << endl ;
 cout << "toto @ " << (long) &toto << endl ;
 cout << "tutu @ " << (long) &tutu << endl ;
 cout << "titi @ " << (long) &titi << endl ;
 cout << "tata @ " << (long) &tata << endl ;
 cout << "i @ "    << (long) &i << endl ;
 cout << "j @ "    << (long) &j << endl ;
 cout << "k @ "    << (long) &k << endl ;
 return 0 ;
}
```

```
$ ./hello-ptr.exe

f() @ 4294970484

toto @ 4294971604
tutu @ 4294971608
titi @ 4294971612
tata @ 4294971616

i @ 140734799804652
j @ 140734799804648
k @ 140734799804644
```

INSTITUT d'OPTIQUE GRADUATE SCHOOL

Pointeur : opérateur de déférencement

```
1: #include <iostream>
2: using namespace std;
3: int toto, tutu, titi, tata ;
4: void f( int x ) {  cout <<  "x vaut " << x << endl; }
5: int main(int argc, char** argv)
6: {
7:  int i    = 5 ;
8:  int* p_i = &i ;
9:  cout << " @i   = " << &i << endl
10:      << " p_i  = " << p_i << endl
11:      << " @p_i = " << &p_i << endl;
12: *p_i     = 100 ;
13: cout << " i = " << i << endl ;
14:
    return 0 ;
  }
```

Adresses	Mémoire
...	...
000012	p_i
000008	i
000004	...
000000	...

Qu'afficherait ce programme ?

INSTITUT d'OPTIQUE GRADUATE SCHOOL

Introduction aux références

- Références n'existent qu'en C++
- Forme plus sûre
  - pour passer une adresse à une fonction
  - nécessaire pour les classes et les objets
- Attention !
  - Syntaxe identique à l'opérateur d'adressage (&)

- Référence = pointeur constant automatiquement déférencé
- Référence correspond à *inout* en algorithmique

Références et pointeurs

Exemple de code

Exemple d'utilisation de référence

```
1:  #include <iostream>
2:  using namespace std;
3:
4:  void f( int & r ) {
5:   cout << "r vaut "  << r << endl;
6:   cout << "@r vaut " << &r << endl;
7:   r = 5;
8:   cout << "r vaut " << r << endl;
9:  }
10: int main(int argc, char** argv) {
11:  int x  = 47 ;
12:  cout << " x  = " << x << endl
13:       << " @x = " << &x << endl;
14:  f(x) ;
15:  cout << " x  = " << x << endl;
16:  return 0 ;
    }
```

INSTITUT d'OPTIQUE GRADUATE SCHOOL

Exemple d'utilisation de référence

```
1:  #include <iostream>
2:  using namespace std;
3:
4:  void f( int & r ) {
5:   cout << "r vaut "  << r << endl;
6:   cout << "@r vaut " << &r << endl;
7:   r = 5;
8:   cout << "r vaut " << r << endl;
9:  }
10: int main(int argc, char** argv) {
11:  int x  = 47 ;
12:  cout << " x  = " << x << endl
13:       << " @x = " << &x << endl;
14:  f(x) ;
15:  cout << " x  = " << x << endl;
16:  return 0 ;
    }
```

Passage par référence de r !!!

INSTITUT d'OPTIQUE GRADUATE SCHOOL

Exemple d'utilisation de référence

```
1:  #include <iostream>
2:  using namespace std;
3:
4:  void f( int & r ) {
5:   cout << "r vaut "  << r << endl;
6:   cout << "@r vaut " << &r << endl;
7:   r = 5;
8:   cout << "r vaut " << r << endl;
9:  }
10: int main(int argc, char** argv) {
11:  int x  = 47 ;
12:  cout << " x  = " << x << endl
13:       << " @x = " << &x << endl;
14:  f(x) ;
15:  cout << " x  = " << x << endl;
16:  return 0 ;
    }
```

Passage par référence de r !!!

```
>$ ./hello-reference.exe
 x  = 47
 @x = 0x7fff5fbff8cc
r vaut 47
@r vaut 0x7fff5fbff8cc
r vaut 5
 x  = 5
```

INSTITUT d'OPTIQUE GRADUATE SCHOOL

Exercice en TP sur les références vs pointeurs

L'échange du contenu de deux variables

- Avec des pointeurs

vs

- Avec des références

Propriétés des références

- Pointeur constant automatiquement déférencé

⇔ Pointeur sûr et fiable

- Une référence (&) est
  - Obligatoirement initialisée à sa création
  - Liée à une allocation (type primitif ou objet)
- Une référence n'est jamais
  - égale à NULL
  - effacée par l'opérateur delete

Portée des déclarations

- Les accolade { } définissent un bloc de portée.
- Validité : de la déclaration à la prochaine }
- Exemples :
  - structure de contrôle: for, while, if-else
  - fonction
  - ad-hoc
- En C++ déclaration à volée des variables
  - ≠ du langage C. Déclaration début de bloc

Déclaration et définition à la volée

```
1: #include <iostream>
2: using namespace std;
3: int main(int argc, char** argv) {
4:  for( int i=0 ; i < 10*argc; i++ )
5:  {
6:     int p = i ;                    // p est local au bloc de la boucle for
7:     cout << p << endl ;
8:  }
9:  int p = 4 ;                       // Ceci est un autre p dans le bloc main
10: cout << p << endl ;
11: {
12:  int p = 40 ;                     // Ceci est encore un autre p dans un
13:  cout << p << endl ;               // bloc ad-hoc
14: }
15: return 0 ;
16:}
```

Mot-clé *static* dans une fonction

```
1: #include <iostream>
2: using namespace std;
3:
4: void func() {
5:   static x = 0 ;
6:   cout << "x =" << ++x << endl;
7: }
8:
9: int main(int argc, char** argv) {
10: for(int i=0; i<10 ; i++) {
11:  func() ;
12: }
13: return 0 ;
14:}
15:
16:
```

Mot-clé *static* dans une fonction

```
1:  #include <iostream>
2:  using namespace std;
3:
4:  void func() {
5:    static x = 0 ;
6:    cout << "x =" << ++x << endl;
7:  }
8:
9:  int main(int argc, char** argv) {
10:  for(int i=0; i<10 ; i++) {
11:    func() ;
12:  }
13:  return 0 ;
14: }
15:
16:
```

Quelle exécution

- Sans le mot-clé *static*

- Avec ?

Mot-clé *static* HORS d'une fonction

Toto.hpp

```
1: #include <iostream>
2: using namespace std;
3:
4: static int xfs ;
5:
6: int main(int argc, char** argv) {
7:  xfs = 1 ;
8:  return 0 ;
9: }
```

- Dans ce cas, static force xfs à n'exister que dans ce fichier !

- On ne peut PLUS référencer xfs depuis l'extérieur
→ (erreur d'édition de lien)

Rappel sur le Mot-clé *extern*

main.cpp

```
1:
2: extern int job;
3:
4: int main(int argc, char** argv) {
5:  job = 1 ;
6:  return 0 ;
7: }
8: int job ;
```

- Déclaration
- Promesse d'une définition d'une variable globale nommée job

Retour sur l'édition de lien

- Seul l'éditeur de lien peut vérifier la validité des identifiants déclarés en *static* ou *extern*
- Deux types de lien :
  - interne : stockage unique pour fichier entrain d'être compilé.

  → Utilisation du mot-clé *static*
  - externe : stockage partagé pour tous les fichiers compilés

  → mot-clé *extern*

Constante en C++

```
#include <iostream>

using namespace std ;

int   const  N_CTE = 10 ;   // CECI EST UNE CONSTANTE

float const PI = 3.14159f;   // AUSSI UNE CONSTANTE


int main(int argc, char** argv) {
 cout << PI << endl ;
 cout << N_CTE << endl ;
 return 0 ;
}
```

INSTITUT d'OPTIQUE GRADUATE SCHOOL

Constante en C++

```
#include <iostream>

using namespace std ;

int   const  N_CTE = 10 ;    // CECI EST UNE CONSTANTE

float const PI = 3.14159f;    // AUSSI UNE CONSTANTE


int main(int argc, char** argv) {
 cout << PI << endl ;
 cout << N_CTE << endl ;
 return 0 ;
}
```

- Mot-clé *const*
- Une constante est variable nommée
  - initialisée
  - impossible à modifier par la suite

Constante en C++

```
#include <iostream>

using namespace std ;

int   const  N_CTE = 10 ;    // CECI EST UNE CONSTANTE

float const PI = 3.14159f;    // AUSSI UNE CONSTANTE


int main(int argc, char** argv) {
 cout << PI << endl ;
 cout << N_CTE << endl ;
 return 0 ;
}
```

- En C, on utiliserait une macro
  - Absence type
  - Pas de contrôle de la portée
  - Dur à débugger

→ A PROSCRIRE

Constante en C++ comme modificateur

```
1:  int main(int argc, char** argv)
2:  {
3:   int i               = 10 ;  // i est en entier
4:
5:   const int * p_i  = &i ;
6:
7:   int const *   p2_i = p_i ;
8:
9:   int const p3_i = &i ;
10:
11:  int const const p4_i = &i ;
12:
13:  return 0 ;
14: }
15:
16:
```

Constante en C++ comme modificateur

```
1:  int main(int argc, char** argv)
2:  {
3:   int i               = 10 ;  // i est en entier
4:
5:   const int  * p_i   = &i ;  // p_i est un pointeur sur un entier constant
6:
7:   int const *   p2_i = p_i ; //
8:
9:   int * const p3_i = &i ;   //
10:
11:  int const * const p4_i = &i ; //
12:
13:  return 0 ;
14: }
15:
16:
```

Constante en C++ comme modificateur

```
1:  int main(int argc, char** argv)
2:  {
3:   int i               = 10 ;  // i est en entier
4:
5:   const int * p_i  = &i ;  // p_i est un pointeur sur un entier constant
6:
7:   int const *   p2_i = p_i ; // pareil que p_i MAIS STYLE A PREFERRER
8:
9:   int * const p3_i = &i ;   //
10:
11:  int const * const p4_i = &i ; //
12:
13:  return 0 ;
14: }
15:
16:
```

Constante en C++ comme modificateur

```
1:  int main(int argc, char** argv)
2:  {
3:   int i               = 10 ;  // i est en entier
4:
5:   const int * p_i  = &i ;  // p_i est un pointeur sur un entier constant
6:
7:   int const *   p2_i = p_i ; // pareil que p_i MAIS STYLE A PREFERRER
8:
9:   int * const p3_i = &i ;   // p3_i est un pointeur CONSTANT sur un entier
10:
11:  int const * const p4_i = &i ; //
12:
13:  return 0 ;
14: }
15:
16:
```

Constante en C++ comme modificateur

```
1:  int main(int argc, char** argv)
2:  {
3:   int i               = 10 ;  // i est en entier
4:
5:   const int  * p_i   = &i ;  // p_i est un pointeur sur un entier constant
6:
7:   int const *   p2_i = p_i ; // pareil que p_i MAIS STYLE A PREFERRER
8:
9:   int * const p3_i = &i ;   // p3_i est un pointeur CONSTANT sur un entier
10:
11:  int const * const p4_i = &i ; // p4_i est un pointeur CONSTANT sur un
12:                                 // sur un entier AUSSI CONSTANT
13:
14:  return 0 ;
15: }
16:
```

Opérateurs explicites de casts en C++

- Ancien opérateurs hérités du C

```
int b = 200 ;
unsigned int long a = (unsigned long int ) b ;
```

- Problèmes :
  - Désactive la vérification de type par le compilateur
  - Sources de bugs difficiles à trouver
- Solution C++. 4 opérateurs explicites
  - `static_cast`, `const_cast`, `reinterpret_cast`, `dynamic_cast`

Opérateurs explicites de casts en C++

- `static_cast` :

→ le plus sûr qui explicite des conversions souvent implicites

- `const_cast` :

→ pour convertir des pointeurs constants vers des non-constants

- `reinterpret_cast` :

→ le plus dangereux. permet de changer complètement le type

- `dynamic_cast` :

  → équivalent d'un downcasting (cf. Héritage)

static_cast<>

```
1: #include <iostream>
2:
3: int main( int argc, char** argv ) {
4:  int i = 0x7fff ;
5:  long l = static_cast<long>(i)  ;      // Conversion tjrs OK
6:  float f = static+cast<float>(i);      // Conversion tjrs OK
7:
8:  i = static_cast<int>(l); // Perte de precision en connaissance de cause
9:  i = static_cast<int>(f); // Perte de precision en connaissance de cause
10:
11: void*  vp = &i;                       // Toujours valide
12: float* fp = (float*) vp ;            // Ancienne façon ala C. DANGEREUX !!!
13: float* fp = static_cast<float*>(vp);   // Nouvelle façon DANGEREUX AUSSI
14: return 0 ;
15:}
16:
```

INSTITUT d'OPTIQUE GRADUATE SCHOOL

const_cast<>

```
1: #include <iostream>
2:
3: int main( int argc, char** argv )
4: {
5:  int const i = 2000 ;
6:
7:  int*  j = &i;                          // DEPRECATED
8:  float* fp = const_cast<int*>(&i);  // Nouvelle façon  = A PREFERER
9:  return 0 ;
10:}
```

Tableaux en C++

```
1: #include <iostream>
2: using namespace std ;
3: int main( int argc, char* argv[] )
4: {
5:  int a[2000] ;
6:  for( int i=0 ; i < 2000 ; i++ ) {
7:    cout << a[i] << endl;
8:  }
9:
10: return 0 ;
   }
```

INSTITUT d'OPTIQUE GRADUATE SCHOOL

Tableaux en C++

```
1:  #include <iostream>
2:  using namespace std ;
3:  int main( int argc, char* argv[] )
4:  {
5:   int a[2000] ;                          // Tableau sur la pile de 2000 entiers
6:   for( int i=0 ; i < 2000 ; i++ ) {
7:     cout << a[i] << endl;
8:   }
9:
10:  return 0 ;
    }
```

Tableaux en C++

```
1: #include <iostream>
2: using namespace std ;
3: int main( int argc, char* argv[] )
4: {
5:  int a[2000] ;                    // Tableau sur la pile de 2000 entiers
6:  for( int i=0 ; i < 2000 ; i++ ) {
7:    cout << a[i] << endl;
8:  }
9:
10: cout << "a" << &a << endl ;
11: cout << "&a[0]" << &a[0] << endl ;
12: return 0 ;
   }
```

INSTITUT d'OPTIQUE GRADUATE SCHOOL

Tableaux et pointeur en C++

```
1: #include <iostream>
2: using namespace std ;
3: int main( int argc, char* argv[] )
4: {
5:  int a[2000] ;                          // Tableau sur la pile de 2000 entiers
6:  for( int i=0 ; i < 2000 ; i++ ) {
7:    cout << a[i] << endl;
8:  }
9:
10: cout << "a" << &a << endl ;
11: cout << "&a[0]" << &a[0] << endl ;
12: return 0 ;
   }
```

```
#>./hello.exe
&a    = 0x7fff5fbfd9b0
&a[0] = 0x7fff5fbfd9b0
```

- type de a : int*

→ a est un pointeur

→ [] : opérateur

INSTITUT d'OPTIQUE GRADUATE SCHOOL

La gestion dynamique de la mémoire

- Rappel en C , la gestion du tas :
  - malloc() : allocation de mémoire

    → Il faut vérifier le retour de malloc
  - calloc() : allocation continue de mémoire et initialisation à zéro de la zone réservée
  - realloc() : redimensionnement de la zone mémoire allouée
  - free() : libération de la mémoire

La gestion dynamique de la mémoire

- En C++, nouveaux opérateurs :
  - new : opérateur qui alloue de la mémoire en fonction du type demandé (pointeur, objet,...)

  → Pas besoin de vérifier si l'opération a reussi

  - delete : libération de la mémoire pour pointeur et objet

  - delete[] : libération de la mémoire pour une zone continue (e.g. tableau)

Tableaux dynamiques en C++

```
1:  #include <iostream>
2:  using namespace std ;
3:  int main( int argc, char* argv[] ) {
4:    int* p = new int[1];                    // Allocation dynamique Tableau
5:                                            // p : pointeur sur int
6:    cout << " p =  " << p << endl;
7:    *p = 10;
8:    cout << " *p = p[0] = " << p[0] << endl;
9:    delete p;                               // libération de p (equivalent à un free)
10:   int* p2 = NULL;
11:   cout << " p2 =  " << p2 << endl;
12:   p2 = new int;                           // Allocation de p2 : pointeur sur int
13:   *p2 = 100;
14:   cout << " p2[0] =  " << p2[0] << endl;
15:   cout << " *p2 =  "   << *p2 << endl;
16:   delete p2;                              // libération de p2 (equivalent à un free)
      return 0 ;
    }
```

INSTITUT d'OPTIQUE GRADUATE SCHOOL

Tableaux dynamiques en C++

```
1: #include <iostream>
2: using namespace std ;
3: int main( int argc, char* argv[] ) {
4:   int* p = new int[1];                    // Allocation dynamique Tableau
5:                                           // p : pointeur sur int
6:   cout << " p =  " << p << endl;
7:   *p = 10;
8:   cout << " *p = p[0] = " << p[0] << endl;
9:   delete p;                               //
10:  int* p2 = NULL;
11:  cout << " p2 =  " << p2 << endl;
12:  p2 = new int;                          //
13:  *p2 = 100;
14:  cout << " p2[0] =  " << p2[0] << endl;
15:  cout << " *p2 =  "   << *p2 << endl;
16:  delete p2;                             //
     return 0 ;
   }
```

```
#>./hello.exe
 p =  0x100100080
 *p = p[0] = 10
 p2 =  0
 p2[0] =  100
 *p2 =  100
```

INSTITUT d'OPTIQUE GRADUATE SCHOOL

Tableaux dynamiques en C++

```
1:  #include <iostream>
2:  using namespace std ;
3:  int main( int argc, char* argv[] )
4:  {
5:    int* const p3 = new int[100];    // tableau de 100 elements alloués sur le tas
6:                                     // p3 pointeur constant alloué dynamiquement
7:    cout << " p3[10] = " << p3[10] << endl;
8:    // p3++; // COMPILATION ERROR error: increment of read-only variable ‘p3’
9:    delete[] p3;
10:
11:   int* const p4 = new int[argc] ;  // p4 tableau dynamique alloué sur le tas
12:   delete[] p4;

      return 0 ;
    }
```

Le mot-clé *inline*

```
1: inline int plusOne( int x ) ; // Déclaration d'une fonction inline
2:
3: inline int plusOne( int x )  // Définition d'une fonction inline
4: {
5:   return ++x ;
6: }
7:
8: int main( int argc, char** argv )
9: {
10: int const  foo = 3 ;
11: int        bar = plusOne(foo) ;
12:
13: return 0 ;
14:}
```

- S'applique à une fonction
- Typage effectué
- Remplace le contenu de la fonction à l'appel de la fonction
- Utilisation dans les en-tête

Compilation : vue d'ensemble pour GCC

- Front-end
- Middle-end
- Back-end
- Parsing
- GIMPLIFICATION
- Pass Manager

Éléments non traites

- Structures de contrôle
  - do...while
  - switch
  - goto
- Mot-clé asm

•Mot-clés C++ absents du langage C

C++ Keywords That Are Not C Keywords					
and	*and_eq*	*asm*	*bitand*	*bitor*	*bool*
catch	*class*	*compl*	*const_cast*	*delete*	*dynamic_cast*
explicit	*export*	*false*	*friend*	*inline*	*mutable*
namespace	*new*	*not*	*not_eq*	*operator*	*or*
or_eq	*private*	*protected*	*public*	*reinterpret_cast*	*static_cast*
template	*this*	*throw*	*true*	*try*	*typeid*
typename	*using*	*virtual*	*wchar_t*	*xor*	*xor_eq*

Précédence des opérateurs

- Chaque boite contient les opérateurs de même priorité.
- Les boites sont listées des opérateurs avec la plus forte priorité jusqu'à la moins forte

- Cours 1
  - Généralités
  - Le processus de création d'un programme
  - Tour d'horizon du C++
  - La programmation impérative en C++
- Cours 2
  - Vers l’objet.
- Cours 3
- Cours 4

- Cours 1
  - Généralités
  - Le processus de création d'un programme
  - Tour d'horizon du C++
  - La programmation impérative en C++
- Cours 2
  - Vers l’objet
- Cours 3
- Cours 4

Pourquoi la programmation objet ?

- Nécessité
  - Organisation/Regroupement des données d'une même entité

  ➔ Structures ?
  - Séparation données de leurs accès/manipulations

  ➔ Fonctions/Procédures ?

Structures

- Structures en C
- Typedef à la rescousse
- Structures en C++
  - Visibilité
  - Fonctions sur les Structures

Structures en C

- Un exemple simple de tableau dynamique
- Typedef à la rescousse
- Structures en C++
  - Visibilité
  - Fonctions sur les Structures

Structures en C

cdyntab.h

```
1: struct  CDynTabType {
2:  int size;
3:  float* data;
4: };
5:
6:
7:
8:
9:
10:
11:
12:
13:
```

INSTITUT d'OPTIQUE GRADUATE SCHOOL

Structures en C

cdyntab.h

```
1: struct  CDynTabType {
2:  int size;
3:  float* data;
4: };
5:
6:
7:
8:
9:
10:
11:
12:
13:
```

cdyntab-main.c

```
int main(int argc, char* argv[] ) {
  struct CDynTabType a;
  a.size = 10;
  a.data = (float*)
       malloc( sizeof(float)* a.size) );
  a.data[8] = 100.0f;

 for(int i=0; i < a.size; i++ )
   printf("a[%d] = %f \n",i, a.data[i]);

  //Utilisation de la structure etc…
  free(a.data);
  return 0;
}
```

Structures en C

cdyntab.h

```
1: struct  CDynTabType {
2:  int size;
3:  float* data;
4: };
5:
6:
7:
8:
9:
10:
11:
12:
13:
```

cdyntab-main.c

```
int main(int argc, char* argv[] ) {
  struct CDynTabType a;
  a.size = 10;
  a.data = (float*)
       malloc( sizeof(float)* a.size) );
  a.data[8] = 100.0f;

 for(int i=0; i < a.size; i++ )
   printf("a[%d] = %f \n",i, a.data[i]);

  //Utilisation de la structure etc…
  free(a.data);
  return 0;
}
```

typedef à la rescousse

cdyntab.h

```
1: typedef struct
2: CDynTabType {
3:    int size;
4:    float* data;
5: } CDynTab;
6:
7:
8:
9:
10:
11:
12:
13:
```

cdyntab-main.c

```
int main(int argc, char* argv[] ) {
  CDynTab a;
  a.size = 10;
  a.data = (float*)
       malloc( sizeof(float)* a.size) );
  a.data[8] = 100.0f;

 for(int i=0; i < a.size; i++ )
   printf("a[%d] = %f \n",i, a.data[i]);

  //Utilisation de la structure etc…
  free(a.data);
  return 0;
}
```

Opérations sur les structures en C

cdyntab.h

```
1: typedef struct  CDynTabType {
2:  int size;
3:  float* data;
4: } CDynTab;
5:
6: int initialize(CDynTab* s, int size );
7: void cleanup(CDynTab* s );
8: float setElem (CDynTab* s, int index );
9: void reallocate(CDynTab* s );
10:void display(CDynTab* s );
11:
12:
13:
```

cdyntab-main.c

```
int main(int argc, char* argv[] ) {
  CDynTab a;
  a.size = 10;
  a.data = (float*)
       malloc( sizeof(float)* a.size) );
  a.data[8] = 100.0f;

 for(int i=0; i < a.size; i++ )
   printf("a[%d] = %f \n",i, a.data[i]);

  //Utilisation de la structure etc…
  free(a.data);
  return 0;
}
```

Opérations sur les structures en C

cdyntab.h

```
1: typedef struct  CDynTabType {
2:  int size;
3:  float* data;
4: } CDynTab;
5:
6: int initialize(CDynTab* s, int size );
7: void cleanup(CDynTab* s );
8: float setElem (CDynTab* s, int index );
9: void reallocate(CDynTab* s );
10:void display(CDynTab* s );
11:
12:
13:
```

cdyntab-main.c

```
int main(int argc, char* argv[] ) {
  CDynTab a;
  a.size = 10;
  a.data = (float*)
       malloc( sizeof(float)* a.size) );
  a.data[8] = 100.0f;

 for(int i=0; i < a.size; i++ )
   printf("a[%d] = %f \n",i, a.data[i]);

  //Utilisation de la structure etc…
  free(a.data);
  return 0;
}
```

INSTITUT d'OPTIQUE GRADUATE SCHOOL

Opérations sur les structures en C

cdyntab.h

```
1: typedef struct  CDynTabType {
2:  int size;
3:  float* data;
4: } CDynTab;
5:
6: int initialize(CDynTab* s, int size );
7: void cleanup(CDynTab* s );
8:
9: float setElem (CDynTab* s, int index );
10: void display(CDynTab* s );
11:
12: void reallocate(CDynTab* s );
13:
```

cdyntab.c

```
#include “cdyntab.h”
#include <stdlib.h>
#include <stdio.h>

int initialize(CDynTab* s, int size ) {
   s->size = size;
   s->data = (float*)
     malloc( sizeof (float) * size );
    if( ! s->data ) {   return -1; }
    return 0;
}

// Implementation continues
```

INSTITUT d'OPTIQUE GRADUATE SCHOOL

Opérations sur les structures en C

cdyntab.h

```
1: typedef struct  CDynTabType {
2:  int size;
3:  float* data;
4: } CDynTab;
5:
6: int initialize(CDynTab* s, int size );
7: void cleanup(CDynTab* s );
8:
9: float setElem (CDynTab* s, int index );
10: void display(CDynTab* s );
11:
12: void reallocate(CDynTab* s );
13:
```

cdyntab-main.c

```
int main(int argc, char* argv[] ) {
  CDynTab a;
  a.size = 10;
  a.data = (float*)
       malloc( sizeof(float)* a.size) );
  a.data[8] = 100.0f;

 for(int i=0; i < a.size; i++ )
   printf("a[%d] = %f \n",i, a.data[i]);

  //Utilisation de la structure etc…
  free(a.data);
  return 0;
}
```

INSTITUT d'OPTIQUE GRADUATE SCHOOL

Opérations sur les structures en C

cdyntab.h

```
1: typedef struct  CDynTabType {
2:  int size;
3:  float* data;
4: } CDynTab;
5:
6: int initialize(CDynTab* s, int size );
7: void cleanup(CDynTab* s );
8:
9: float setElem (CDynTab* s, int index );
10: void display(CDynTab* s );
11:
12: void reallocate(CDynTab* s );
13:
```

cdyntab-main.c

```
int main(int argc, char* argv[] ) {
  CDynTab a;
  a.size = 10;
  a.data = (float*)
       malloc( sizeof(float)* a.size) );
  a.data[8] = 100.0f;

 for(int i=0; i < a.size; i++ )
   printf("a[%d] = %f \n",i, a.data[i]);

  //Utilisation de la structure etc…
  free(a.data);
  return 0;
}
```

INSTITUT d'OPTIQUE GRADUATE SCHOOL

Opérations sur les structures en C

cdyntab.h

```
1: typedef struct  CDynTabType {
2:  int size;
3:  float* data;
4: } CDynTab;
5:
6: int initialize(CDynTab* s, int size );
7: void cleanup(CDynTab* s );
8:
9: float setElem (CDynTab* s, int index );
10: void display(CDynTab* s );
11:
12:  void reallocate(CDynTab* s );
13:
```

cdyntab-main.c

```
int main(int argc, char* argv[] ) {
  CDynTab a;
  a.size = 10;
  a.data = (float*)
       malloc( sizeof(float)* a.size) );
  a.data[8] = 100.0f;

 for(int i=0; i < a.size; i++ )
   printf("a[%d] = %f \n",i, a.data[i]);

  //Utilisation de la structure etc…
  free(a.data);
  return 0;
}
```

INSTITUT d'OPTIQUE GRADUATE SCHOOL

Opérations sur les structures en C

cdyntab.h

```
1: typedef struct  CDynTabType {
2:  int size;
3:  float* data;
4: } CDynTab;
5:
6: int initialize(CDynTab* s, int size );
7: void cleanup(CDynTab* s );
8:
9: float setElem (CDynTab* s, int index );
10:  void display(CDynTab* s );
11:
12: void reallocate(CDynTab* s );
13:
```

cdyntab-main.c

```
int main(int argc, char* argv[] ) {
  CDynTab a;
  a.size = 10;
  a.data = (float*)
       malloc( sizeof(float)* a.size) );
  a.data[8] = 100.0f;

 for(int i=0; i < a.size; i++ )
   printf("a[%d] = %f \n",i, a.data[i]);

  //Utilisation de la structure etc…
  free(a.data);
  return 0;
}
```

INSTITUT d'OPTIQUE GRADUATE SCHOOL

Opérations sur les structures en C

cdyntab.h

```
1: typedef struct  CDynTabType {
2:  int size;
3:  float* data;
4: } CDynTab;
5:
6: int initialize(CDynTab* s, int size );
7: void cleanup(CDynTab* s );
8:
9: float setElem (CDynTab* s, int index );
10: void display(CDynTab* s );
11:
12: void reallocate(CDynTab* s );
13:
```

cdyntab-main.c

```
#include “cdyntab.h”
int
main(int argc, char* argv[] ) {
  CDynTab a;
  if(! initialize( &a, 10)) {
    exit(-1);
   }
  setElem(&a, 100.0f, 8);

  display( &a );

  cleanup( &a );
  return 0;
}
```

INSTITUT d'OPTIQUE GRADUATE SCHOOL

Pourquoi n'est ce pas de l'objet ?

cdyntab.h

```
1: typedef struct  CDynTabType {
2:  int size;
3:  float* data;
4: } CDynTab;
5:
6: int initialize(CDynTab* s, int size );
7: void cleanup(CDynTab* s );
8:
9: float setElem (CDynTab* s, int index );
10: void display(CDynTab* s );
11:
12: void reallocate(CDynTab* s );
13:
```

- Clash des noms de fonctions
- La visibilité *publique*
  - des données
  - des fonctions

➔ pas de "*private*"

- Gestion dynamique de la mémoire
- Absence de type
  - ➔ Nécessité du typedef

➔ Structures en C++

Structures en C++

- La gestion dynamique de la mémoire
  - ➔ Utilisation de new, delete et delete[]
- Conflit sur les noms de fonctions
  - namespace (réponse partielle)
- Encapsulation
  - données
  - fonctions
    - reallocate devrait être *private*
    - les autres *public*

Structures en C++ : déclaration

cdyntab.hpp

```
1: #ifndef DYNTAB_STRUCT_HPP
2: #define DYNTAB_STRUCT_HPP
3:
4: struct  DynTab {
5:   int size;
6:   float* data;
7: };
8: #endif
```

cdyntab-main.cpp

```
#include "cdyntab.hpp"
int main(int argc, char* argv[] ) {
  DynTab tab;

  return 0;
}
```

- PAS DE typedef NECESSAIRE
- CDynTab est un VRAI TYPE EN C++

INSTITUT d'OPTIQUE GRADUATE SCHOOL

Structures en C++ : encapsulation

cdyntab.hpp

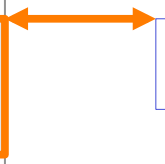
```
1: #ifndef DYNTAB_STRUCT_HPP
2: #define DYNTAB_STRUCT_HPP
3:
4: struct  DynTab {
5:  int size;
6:  float* data;
7:
8:  int initialize(int size );
9:  void cleanup();
10: float setElem (float f,int index );
11: void display();
12: void reallocate();
13:};
14:#endif
```

INSTITUT d'OPTIQUE GRADUATE SCHOOL

Structures en C++ : encapsulation

cdyntab.hpp

```
1: #ifndef DYNTAB_STRUCT_HPP
2: #define DYNTAB_STRUCT_HPP
3:
4: struct  DynTab {
5:   int size;
6:   float* data;
7:
8:   int initialize(int size );
9:   void cleanup();
10: float setElem (float f,int index );
11: void display();
12: void reallocate();
13:};
14:#endif
```



Données

INSTITUT d'OPTIQUE GRADUATE SCHOOL

Structures en C++ : encapsulation

cdyntab.hpp

```
1: #ifndef DYNTAB_STRUCT_HPP
2: #define DYNTAB_STRUCT_HPP
3:
4: struct  DynTab {
5:  int size;
6:  float* data;
7:
8:  int initialize(int size );
9:  void cleanup();
10: float setElem (float f,int index );
11: void display();
12: void reallocate();
13:};
14:#endif
```

Fonctions/Opérations

INSTITUT d'OPTIQUE GRADUATE SCHOOL

Structures en C++ : encapsulation

cdyntab.hpp

```
1: #ifndef CDYNTAB_STRUCT_HPP
2: #define CDYNTAB_STRUCT_HPP
3:
4: struct  CDynTab {
5:  int size;
6:  float* data;
7:
8:  int initialize(int size );
9:  void cleanup();
10: float setElem (float f,int index );
11: void display();
12: void reallocate();
13:};
14:#endif
```

Plus de passage de pointeur de structure dans les fonctions

INSTITUT d'OPTIQUE GRADUATE SCHOOL

Structures en C++ : implémentation

cdyntab.hpp

```
1: #ifndef DYNTAB_STRUCT_HPP
2: #define DYNTAB_STRUCT_HPP
3:
4: struct  DynTab {
5:  int size;
6:  float* data;
7:
8:  int initialize(int size );
9:  void cleanup();
10: float setElem (float f,int index );
11: void display();
12: void reallocate();
13:};
14:#endif
```

cdyntab.cpp

```
#include "cdyntab.hpp"


int
DynTab::initialize(int a_size) {
    size = a_size;
    data = new float[size];
    if( ! data ) {   return -1; }
    return 0;
}
```

INSTITUT d'OPTIQUE GRADUATE SCHOOL

Structures en C++ : implémentation

cdyntab.hpp

```
1: #ifndef DYNTAB_STRUCT_HPP
2: #define DYNTAB_STRUCT_HPP
3:
4: struct  DynTab {
5:  int size;
6:  float* data;
7:
8:  int initialize(int size );
9:  void cleanup();
10: float setElem (float f,int index );
11: void display();
12: void reallocate();
13:};
14:#endif
```

cdyntab.cpp

```
#include "cdyntab.hpp"


int
DynTab::initialize(int a_size) {
    size = a_size;
    data = new float[size];
    if( ! data ) {   return -1; }
    return 0; }
```

Spécification forte de l'appartenance de la fonction à une structure

INSTITUT d'OPTIQUE GRADUATE SCHOOL

Structures en C++ : implémentation

cdyntab.hpp

```
1: #ifndef DYNTAB_STRUCT_HPP
2: #define DYNTAB_STRUCT_HPP
3:
4: struct  DynTab {
5:  int size;
6:  float* data;
7:
8:  int initialize(int size );
9:  void cleanup();
10: float setElem (float f,int index );
11: void display();
12: void reallocate();
13:};
14:#endif
```

cdyntab.cpp

```
#include "cdyntab.hpp"


int
DynTab::initialize(int a_size) {
    size = a_size;
    data = new float[size];
    if( ! data ) {   return -1; }
    return 0;
}
```

Absence de sélection des membres de la structure

(pas de ->)

INSTITUT d'OPTIQUE GRADUATE SCHOOL

Structures en C++ : implémentation

cdyntab.hpp

```
1: #ifndef DYNTAB_STRUCT_HPP
2: #define DYNTAB_STRUCT_HPP
3:
4: struct  DynTab {
5:  int size;
6:  float* data;
7:
8:  int initialize(int size );
9:  void cleanup();
10: float setElem (float f,int index );
11: void display();
12: void reallocate();
13:};
14:#endif
```

cdyntab.cpp

```
#include "cdyntab.hpp"

int DynTab::initialize(int size) {
    size = a_size;
    data = new float[size];
    if( ! data ) {   return -1; }
    return 0; }

void DynTab::cleanup() {
  delete[] data;
}
```

INSTITUT d'OPTIQUE GRADUATE SCHOOL

Structures en C++ : utilisation

cdyntab.hpp

```
1: #ifndef DYNTAB_STRUCT_HPP
2: #define DYNTAB_STRUCT_HPP
3:
4: struct  DynTab {
5:  int size;
6:  float* data;
7:
8:  int initialize(int size );
9:  void cleanup();
10: float setElem (float f, int index );
11: void display();
12: void reallocate();
13:};
14:#endif
```

cdyntab-main.cpp

```
#include "cdyntab.hpp"

int main(int argc, char* argv[] ) {
  DynTab tab;
  tab.initialize( 100 );
  tab.setElement( 1.0f, 10 );
  tab.display();
  tab.cleanup();

  return 0;
}
```

Structures en C++ : résumé

- Vrai type
- "Encapsule" données ET fonctions
- Mais alors structures = classes ?

  ➔ Pas encore!
- Manque
  - Contrôle accès aux données et fonctions
  - Concepts objet:
    - heritage, agregation, polymorphisme,…

De la structure à la classe : contrôle accès

cdyntab.hpp

```
1:struct  DynTab {
2:  private:
3:  int size;
4:  float* data;
5:
6:  public:
7:  int initialize(int size );
8:  void cleanup();
9:  float setElem (float f, int index );
10: void display();
11: private:
12: void reallocate();
13:};
14:#endif
```

De la structure à la classe : contrôle accès

cdyntab.hpp

```
1:struct  DynTab {
2:  private:
3:   int size;
4:   float* data;
5:
6:  public:
7:   int initialize(int size );
8:   void cleanup();
9:   float setElem (float f, int index );
10:  void display();
11:  private:
12:  void reallocate();
13:};
14:#endif
```

Données privées désormais

Fonctions publiques

Fonctions privées

INSTITUT d'OPTIQUE GRADUATE SCHOOL

De la structure à la classe : contrôle accès

cdyntab.hpp

```
1:struct  DynTab {
2:  private:
3:   int size;
4:   float* data;
5:
6:  public:
7:   int initialize(int size );
8:   void cleanup();
9:   float setElem (float f, int index );
10: void display();
11: private:
12: void reallocate();
13:};
14:#endif
```

Remarques

- Aucun changement dans l'implémentation
- i.e., cdyntab.cpp est inchangé
- Fichier *header* contrôle l'accès/visibilité aux données et fonctions de la structure

De la structure à la classe : initialisation

cdyntab.hpp

```
1:struct  DynTab {
2:  private:
3:  int size;
4:  float* data;
5:
6:  public:
7:  int initialize(int size );
8:  void cleanup();
9:  float setElem (float f, int index );
10: void display();
11: private:
12: void reallocate();
13:};
14:#endif
```

cdyntab-main.cpp

```
#include "cdyntab.hpp"

int main(int argc, char* argv[] ) {
  DynTab tab;
  tab.initialize( 100 );
  tab.setElement( 1.0f, 10 );
  tab.display();
  tab.cleanup();

  return 0;
}
```

INSTITUT d'OPTIQUE GRADUATE SCHOOL

De la structure à la classe : initialisation

cdyntab.hpp

```
1:struct  DynTab {
2:  private:
3:  int size;
4:  float* data;
5:
6:  public:
7:  int initialize(int size );
8:  void cleanup();
9:  float setElem (float f, int index );
10: void display();
11: private:
12: void reallocate();
13:};
14:#endif
```

cdyntab-main.cpp

```
#include "cdyntab.hpp"

int main(int argc, char* argv[] ) {
  DynTab tab;
  tab.initialize( 100 );
  tab.setElement( 1.0f, 10 );
  tab.display();
  tab.cleanup();

  return 0;
}
```

INSTITUT d'OPTIQUE GRADUATE SCHOOL

De la structure à la classe : initialisation

cdyntab.hpp

```
1:struct  DynTab {
2:   private:
3:   int size;
4:   float* data;
5:
6:   public:
7:   int initialize(int size );
8:   void cleanup();
9:   float setElem (float f, int index );
10: void display();
11: private:
12: void reallocate();
13:};
14:#endif
```

cdyntab-main.cpp

```
#include "cdyntab.hpp"


int main(int argc, char* argv[] ) {
  DynTab tab;
  tab.initialize( 100 );
  tab.setElement( 1.0f, 10 );
  tab.display();
  tab.cleanup();

  return 0;
}
```

!!! SEGFAULT !!!

ACCES à une zone mémoire non allouée au programme!

INSTITUT d'OPTIQUE GRADUATE SCHOOL

De la structure à la classe: initialisation

- Comment garantir qu'une structure est initialisée ?

⇔ Forcer l'appel à initialize

➔ Une fonction spéciale : le constructeur !

Le constructeur

cdyntab.hpp

```
1:struct  DynTab {
2:  private:
3:  int size;
4:  float* data;
5:
6:  public:
7:  DynTab(int a_size );
8:  void cleanup();
9:  float setElem (float f, int index );
10: void display();
11: private:
12: void reallocate();
13:};
14:#endif
```

cdyntab.cpp

```
#include "cdyntab.hpp"

DynTab::DynTab(int a_size) {
    size = a_size;
    data = new float[size];
    //if( ! data ) {   return -1; }
    //return 0;
}

// IMPLEMENTATION CONTINUE
```

INSTITUT d'OPTIQUE GRADUATE SCHOOL

Le constructeur

cdyntab.hpp

```
1:struct DynTab {
2:   private:
3:   int size;
4:   float* data;
5:
6:   public:
7:   DynTab(int a_size );
8:   void cleanup();
9:   float setElem (float f, int index );
10: void display();
11: private:
12: void reallocate();
13:};
14:#endif
```

cdyntab.cpp

```
#include "cdyntab.hpp"

DynTab::DynTab(int a_size) {
    size = a_size;
    data = new float[size];
    //if( ! data ) {   return -1; }
    //return 0;
}

// IMPLEMENTATION CONTINUE
```

Le constructeur

cdyntab.hpp

```
1:struct  DynTab {
2:   private:
3:   int size;
4:   float* data;
5:
6:   public:
7:   DynTab(int a_size );
8:   void cleanup();
9:   float setElem (float f, int index );
10: void display();
11: private:
12: void reallocate();
13:};
14:#endif
```

cdyntab.cpp

```
#include "cdyntab.hpp"


DynTab::DynTab(int a_size) {
    size = a_size;
    data = new float[size];
    //if( ! data ) {   return -1; }
    //return 0;
}


// IMPLEMENTATION CONTINUE
```

Aucune valeur de retour

Le constructeur est toujours appelé

→ Constructeur par défaut si aucun constructeur de défini

INSTITUT d'OPTIQUE GRADUATE SCHOOL

Le constructeur : initialisation/utilisation

cdyntab.hpp

```
1:struct  DynTab {
2:  private:
3:  int size;
4:  float* data;
5:
6:  public:
7:  DynTab(int size );
8:  void cleanup();
9:  float setElem (float f, int index );
10: void display();
11: private:
12: void reallocate();
13:};
14:#endif
```

cdyntab-main.cpp

```
#include "cdyntab.hpp"

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

  DynTab tab ( 100 );

  tab.setElement( 1.0f, 10 );
  tab.display();
  tab.cleanup();

  return 0;
}
```

INSTITUT d'OPTIQUE GRADUATE SCHOOL

Le constructeur : initialisation par défaut

cdyntab.hpp

```
1:struct  DynTab {
2:   private:
3:   int size;
4:   float* data;
5:   public:
6:
7:
8:   void cleanup();
9:   float setElem (float f, int index );
10: void display();
11: private:
12: void reallocate();
13:};
14:#endif
```

cdyntab-main.cpp

```
#include "cdyntab.hpp"


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

  DynTab tab;
 // ETC
  return 0; }
```

Constructeur par défaut ajouté par le compilateur :

- Qd il n'y a aucun constructeur
- Dangereux:

  ➔ Ne fait pas grand chose

- Precaution: tjrs tout initialiser

Le constructeur : liste d'initialisation

cdyntab.cpp

```
#include "cdyntab.hpp"

DynTab::DynTab(int a_size) {
    size = a_size;
    data = new float[size];
}
// IMPLEMENTATION CONTINUE
```

```
#include "cdyntab.hpp"

DynTab::DynTab(int a_size)
 : size( a_size),
   data( new float[size] )
{
 // AUTRE INITIALISATION
}
```

Le constructeur : liste d'initialisation

cdyntab.cpp

```
#include "cdyntab.hpp"

DynTab::DynTab(int a_size) {
    size = a_size;
    data = new float[size];
}
// IMPLEMENTATION CONTINUE
```

```
#include "cdyntab.hpp"

DynTab::DynTab(int a_size)
 : size( a_size),
   data( new float[size] )
{
 // AUTRE INITIALISATION
}
```

Liste d'initialisation du constructeur

- executée avant le bloc { } du constructeur
- Syntaxe commune à l'héritage
- Les types primitifs ont un constructeur fourni

INSTITUT d'OPTIQUE GRADUATE SCHOOL

Le constructeur : le pointeur *this*

cdyntab.cpp

```
#include "cdyntab.hpp"


DynTab::DynTab(int size)
:  size( size )  // OK but UGLY
{
    size = size; // HUH ???    == ERROR
    (*this).size = size  // BETTER :D
    this->size = size   //  LOVELY
    *this.size = size   // ERROR


    data = new float[size];
}
// IMPLEMENTATION CONTINUES
```

- *this* est un pointeur créé automatiquement par le compilateur
- Membre/Attribut de classe
- == Adresse de l'objet

De la structure à la classe: libération mémoire

- Comment garantir qu'une structure est détruite ?

⇔ Forcer l'appel à cleanup

But: éviter les fuites mémoires (*memory leaks*)

➔ Une fonction spéciale : le destructeur !

Le destructeur: libération de la mémoire

cdyntab.hpp

```
1:struct  DynTab {
2:   private:
3:   int size;
4:   float* data;
5:
6:   public:
7:   DynTab(int size );
8:   ~DynTab();
9:   float setElem (float f, int index );
10: void display();
11: private:
12: void reallocate();
13:};
14:#endif
```

cdyntab.cpp

```
#include "cdyntab.hpp"


DynTab::DynTab(int size) {
    size = a_size;
    data = new float[size];
    if( ! data ) {   return -1; }
    return 0;
}
```

INSTITUT d'OPTIQUE GRADUATE SCHOOL

Le destructeur: libération de la mémoire

cdyntab.hpp

```
1:struct  DynTab {
2:   private:
3:   int size;
4:   float* data;
5:
6:   public:
7:   DynTab(int a_size );
8:   ~DynTab();
9:   float setElem (float f, int index );
10: void display();
11: private:
12: void reallocate();
13:};
14:#endif
```

cdyntab.cpp

```
#include "cdyntab.hpp"

DynTab::DynTab(int a_size) {
    size = a_size;
    data = new float[size];
    if( ! data ) {   return -1; }
    return 0;
}
DynTab::~DynTab::cleanup() {
  delete[] data;
}
```

INSTITUT d'OPTIQUE GRADUATE SCHOOL

Le destructeur: libération de la mémoire

cdyntab.hpp

```
1:struct  DynTab {
2:  private:
3:  int size;
4:  float* data;
5:
6:  public:
7:  DynTab(int a_size );
8:  ~DynTab();
9:  float setElem (float f, int index );
10: void display();
11: private:
12: void reallocate();
13:};
14:#endif
```

cdyntab-main.cpp

```
#include "cdyntab.hpp"

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

  DynTab tab ( 100 );

  tab.setElement( 1.0f, 10 );
  tab.display();
  //Appel implicite à la sortie du bloc
  // DynTab::~DynTab();
  return 0;
}
```

INSTITUT d'OPTIQUE GRADUATE SCHOOL

Structures et classes

```
1:struct  DynTab {
2:   private:
3:   int size;
4:   float* data;
5:
6:   public:
7:   DynTab(int size );
8:   void cleanup();
9:   float setElem (float f, int index );
10: void display();
11: private:
12: void reallocate();
13:};
14:#endif
```

- Structures ressemblent aux classes Java
- 1 fichier déclaration (.hpp)
- 1 fichier implémentation (.cpp)
- Mot-clé class

- Structure = Classe

→ ou presque

Structures et classes

```
1:class DynTab {
2:   private:
3:   int size;
4:   float* data;
5:
6:   public:
7:   DynTab(int size );
8:   void cleanup();
9:   float setElem (float f, int index );
10: void display();
11: private:
12: void reallocate();
13:};
14:#endif
```

- Structures
- 1 fichier déclaration (.hpp)
- 1 fichier implémentation (.cpp)
- Mot-clé class

- Structure = Classe

→ ou presque

INSTITUT d'OPTIQUE GRADUATE SCHOOL

Différences entre structures et classes

Par défaut :

- Structure
  - Membres publiques ➔ historique du C
- Classe
  - Membres privés

  ➔ principe fort de l'encapsulation en POO

- Cours 1
  - Généralités
  - Le processus de création d'un programme
  - Tour d'horizon du C++
  - La programmation impérative en C++
- Cours 2
  - Vers l’objet
  - Classe et Objet en C++
  - Héritage…

- Cours 1
  - Généralités
  - Le processus de création d'un programme
  - Tour d'horizon du C++
  - La programmation impérative en C++
- Cours 2
  - Vers l'objet
  - Classe et Objet en C++
  - Héritage…

Classe: déclaration et définition

cdyntab.hpp

```
1:class DynTab {
2:   private:
3:   int size;
4:   float* data;
5:
6:   public:
7:   DynTab(int size );
8:   ~DynTab();
9:   float setElem (float f, int index );
10: void display();
11: private:
12: void reallocate();
13:};
14:#endif
```

cdyntab.cpp

```
#include "cdyntab.hpp"
DynTab::DynTab(int a_size) {
    size = a_size;
    data = new float[size];
}
DynTab::~DynTab () {
  delete[] data;
}
DynTab::setElem( float f, int index ){
 data[index] = f;
}
// ETC ETC
```

INSTITUT d'OPTIQUE GRADUATE SCHOOL

Vue d'ensemble: Classe en C++

```
1: #ifdef IDENTIFIANT_UNIQUE_HPP
2: #define IDENTIFIANT_UNIQUE_HPP
3: class NomClasse {
4:  [public|protected|private]:
5:  // Membres/Attributs de classes
6:
7:  [public|protected|private]:
8:   // Constructeurs
9:   NomClasse(…);
10:   // Destructeur
11:   ~NomClasse();
12:  // Méthodes
13:  // Surcharge d'Opérateurs
14: };
15:#endif
```

- Un fichier .hpp
  - Déclaration
- Un fichier .cpp
  - Définition
  - Inclusion du .hpp

Constness dans les classes

- Membre constant
  - par objet
  - par classe
- Méthode constante

Membre Constant par objet

```
1: #ifdef IDENTIFIANT_UNIQUE_HPP
2: #define IDENTIFIANT_UNIQUE_HPP
3: class NomClasse {
4:   [public|protected|private]:
5:    int const SIZE;
6:
7:   [public|protected|private]:
8:    NomClasse( int size ) : SIZE( size )
9:    {}
10:    // Destructeur
11:    ~NomClasse();
12:   // Méthodes
13:   // Surcharge d'Opérateurs
14: };
15:#endif
```

- Membre constant
- Ne change plus une fois l'objet initialisé
- La constante n'est pas forcément identique entre 2 objets

INSTITUT d'OPTIQUE GRADUATE SCHOOL

Membre constant par classe

```
1: #ifdef IDENTIFIANT_UNIQUE_HPP
2: #define IDENTIFIANT_UNIQUE_HPP
3: class NomClasse {
4:   [public|protected|private]:
5:    static int const SIZE = 200;
6:
7:   [public|protected|private]:
8:    NomClasse();
9:
10:    // Destructeur
11:    ~NomClasse();
12:   // Méthodes
13:   // Surcharge d'Opérateurs
14: };
15:#endif
```

- Membre constant et statique initialisé à la compilation
- Identique pour tous les objets

Méthode constante et objet constant

```
1: int main() {
2:    int const i = 23;
3:
4:    Blob const b( 10 );
5:    return 0;
6: }
```

- Constante d'un type primitif

Méthode constante et objet constant

```
1: int main() {
2:    int const i = 23;
3:
4:    Blob const b( 10 );
5:    return 0;
6: }
```

- Constante d'un type primitif
- Objet constant de type classe Bloc

➔ Comment s'assurer que les appels de méthodes ne vont pas changer les attributs de b?

Méthode constante et objet constant

```
1: int main() {
2:    int const i = 23;
3:
4:    Blob const b( 10 );
5:    return 0;
6: }
```

- Constante d'un type primitif
- Objet constant de type classe Bloc

Comment s'assurer que les appels de méthodes ne vont pas changer les attributs de b?

➔ Méthode déclarée *const*

Méthode constante et objet constant

```
1: #ifdef  BLOB_HPP
2: #define BLOB_HPP
3: class Blob {
4:    int _m;
5:    public:
5:    Blob( int j ) :  _m( j ) { }
6:    int getMe() const
7:    {return _m; }
8:
9:    void setMe( int i )
10:   {  _m = i; }
11:
12: };
13:#endif
```

- Méthode déclarée *const*

== Méthode ne modifiant pas les attributs de la classe

Arguments par défaut

```
1:class DynTab {
2:   private:
3:   int size;
4:   float* data;
5:
6:   public:
7:   DynTab(int size  = 100);
8:   void cleanup();
9:   float setElem (float f ,
                    int index = 0 );
10: void display();
11: private:
12: void reallocate();
13:};
14:#endif
```

Arguments par défaut

```
1:class DynTab {
2:   private:
3:   int size;
4:   float* data;
5:
6:   public:
7:   DynTab(int size  = 100);
8:   void cleanup();
9:   float setElem (float f ,
                    int index = 0 );
10: void display();
11: private:
12: void reallocate();
13:};
14:#endif
```

args par défaut existent :

- Pour les constructeurs
  - Remplace le constructeur par défaut

INSTITUT d'OPTIQUE GRADUATE SCHOOL

Arguments par défaut

```
1:class DynTab {
2:   private:
3:   int size;
4:   float* data;
5:
6:   public:
7:   DynTab(int size  = 100);
8:   void cleanup();
9:   float setElem (float f ,
                    int index = 0 );
10: void display();
11: private:
12: void reallocate();
13:};
14:#endif
```

- Pour les constructeurs
  - Remplace le constructeur par défaut
- Pour les méthodes

➔ Alternative à la surcharge

INSTITUT d'OPTIQUE GRADUATE SCHOOL

Arguments par défaut

```
1:class DynTab {
2:   private:
3:   int size;
4:   float* data;
5:
6:   public:
7:   DynTab(int size  = 100);
8:   void cleanup();
9:   float setElem (float f ,
                    int index = 0 );
10: void display();
11: private:
12: void reallocate();
13:};
14:#endif
```

- Pour les constructeurs
  - Remplace le constructeur par défaut
- Pour les méthodes

➔ Alternative à la surcharge

- Les arguments par défaut doivent se trouver à la fin
- Les fichiers .cpp sont inchangés