Vanden Berghen Frank
E-Mail : fvandenb@iridia.ulb.ac.be

Langage C++

Le 7-7-2003.

Langage C++

1.Introduction à C++

1.1. Extentions non-orientées objets C++

1.1.1. Qualificateur INLINE

Lors de la compilation, l'appel d'une fonction munie du qualificateur INLINE, est remplacé par le code même du corps de celle-ci. Cela permet de gagner du temps en exécution lorsque la procedure d'appel d'une fonction est non-négligeable par rapport au temps d'exécution de la fonction elle-même. Considérons maintenant le programme suivant :

```
#include <stdio.h>
int y=1;
#define Y_Plus_Un_1 y+1
#define Y_Plus_Un_2 (y+1)
inline int Y_Plus_Un_3( void ) { return y+1; };
void main()
{
 printf("test1: %u\ntest2: %u\ntest3: %u\n",
        2*Y_Plus_Un_1,  2*Y_Plus_Un_2,    2*Y_Plus_Un_3());
}
```

Le résultat est :

```
test1: 3
test2: 4
test3: 4
```

Les différentes façon de calculer `Y_Plus_Un` sont toutes "inline" mais `Y_Plus_Un_1` ne donne pas le résultat espéré. Cela est du au fait que le compilateur remplace tout simplement le contenu du define sans faire attention à ce qui est "autour". Il ignore donc les règles de priorité. Il est donc plus dangereux d'utiliser des `#define` que des `inline`.

1.1.2. Opérateur de résolution de portée.

```
#include <iostream.h>
int var = 12;
void main ()
{
  float var=1.1234;
  cout<<"local: "<<var<<"\nglogal: "<<::var<<"\n";
}
```

Le résultat est :

```
local: 1.1234
global: 12
```

Cet opérateur est surtout employé en programmation objet pour pouvoir distinguer les membres d'une classe héritière des membres de même nom des classes parentes.

Type référence / qualificateur CONST

Les type références sont souvent utilisés comme argument de fonctions. Cela évite la copie d'une structure qui peut être assez grande. Le qualificateur CONST est alors utilisé pour empêcher de modifier intempestivement la variable passée et garantir l'absence d'Effet de Bord.

Remarque:
- Pointeur vers une constante :
  type_valeur **const ***nom_pointeur***;** **ou** **const** *type_valeur* *****nom_pointeur***;**
- Pointeur constant :
  type_valeur ***const** *nom_pointeur*=**&***nom_variable***;**
- Variable référence (l'adresse et la variable pointée sont des constantes) :
  const *type_valeur***&***nom_référence=nom_variable;* **ou**
  type_valeur **const &***nom_référence=nom_variable;*
  (attention : dans cet exemple :
  type_valeur **&const** *nom_référence=nom_variable;*
  le qualificateur CONST est purement et simplement ignoré).

Les variables références constantes peuvent encore être utilisées pour accéder uniquement en lecture à un membre privé d'une classe. Dans le programme ci-dessous, je crée une classe qui a pour but de stocker un **integer A**. Après création, seule la lecture de **A** est permise. Pour lire **A**, nous pouvons employer `fnct_lecture()`(procédé lent) ou `Lecture` (procédé rapide). Attention, l'emploi de variables références ou de variables pointeurs dans une classe nécessite de définir un construteur de copie et un opérateur d'affectation pour cette classe.

```
class Compte
{ public:
      const int & lecture;

      int fnct_lecture() { return Solde; };

// contructeur :
      Compte(int S=0) :
            lecture (Solde), Solde(S) {};

// constructeur de copie:
      Compte( const Compte &inter ) :
            lecture (Solde), Solde(inter.lecture) {};

//opérateur d'affectation:
      Compte & operator = ( const Compte &inter )
      {
            Solde=inter.lecture;
            return *this;
      };
private:
      int Solde;
}
```

1.1.4. Arguments de fonctions par défaut.

Les paramètres par défaut d'une fonction doivent être les dernier de la liste. De plus, il ne peut y avoir de "trou" dans l'appel. Il n'est pas possible d'initialiser avec ce mécanisme des tableaux ou des structures. Les 3 déclarations suivantes sont conflictuelles et rejetées par le compilateur :

```
void fonction ( int a=0 );
void fonction ( );
void fonction ( int a ) ;
```

Autre exemple:

```
#include <iostream.h>

struct complexe { float reel; float imag; };
const complexe UN {1.,0.};
```

```
void increment ( complexe &C, complexe INCR= UN )
       // impossible d'écrire : complexe INCR= {1.,0}
{ C.reel+=INCR.reel; C.imag+=INCR.imag; }

void main()
{
      complexe x={2.,3.}, y{4.,5.};
      increment(x,y);   cout<<x.reel<<","<<x.imag<<"\n";
      increment(x);     cout<<x.reel<<","<<x.imag<<"\n";
};
```

Ce programme donne :
```
6,8
7,8
```

1.1.5. Surcharge de fonctions

Un fonction est dite suchargée lorsque il existe plusieurs déclarations de cette fonction uniquement différenciée par les paramètres qu'elle prend. Attention le type de retour n'est pas pris en compte dans la signature de la fonction : l'exemple de déclaration suivant sera refusé par le compilateur :

```
int  fonction( int );
char fonction( int );
```

Exemple équivalent à l'exemple du paragraphe précédent :

```
#include <iostream.h>

struct complexe { float reel; float imag; };

void increment ( complexe &C, complexe INCR )
{ C.reel+=INCR.reel; C.imag+=INCR.imag; }

void increment ( complexe &C )
{ C.reel+=1.; }

void main()
{
      complexe x={2.,3.}, y{4.,5.};
      increment(x,y);   cout<<x.reel<<","<<x.imag<<"\n";
      increment(x);     cout<<x.reel<<","<<x.imag<<"\n";
};
```

1.1.6. les type dynamiques

L'opérateur `new` permet d'allouer dynamiquement des objets. (Au momment de la contruction d'un tableau d'objet, le programme fait appel au constructeur par défaut. Si celui-ci n'existe pas alors le compilateur s'arrête.) L'opérateur `delete` permet de libérer la place mémoire d'un objet. L'opérateur `delete[]` permet de libérer la place mémoire d'un ou plusieurs objets.

Soit l'exemple suivant (l'exemple est incomplet mais suffisant pour la clarté de l'exposé):

```
#include <iostream.h>

class Compte
{
  public:
// déclaration implicite d'un constructeur par défaut:
      Compte(int a=0): solde(a) { cout<<" creation\n";    };
      ~ Compte()                { cout<<" destruction\n"; };
  private:
      int solde;
};

void main()
{
   cout<<"test1\n";
   Compte &ref= * new Compte;
```

```
   delete &ref;

   cout<<"test2\n";
   Compte *ptr= new Compte;
   delete[] ptr;

   cout<<"test3\n";
   Compte *mat_cpt= new Compte [2];
   Delete[] mat_cpt;

   cout<<"test4\n";
   Compte *mat_cpt2= new Compte [2];
   Delete mat_cpt2;
}
```

La sortie du programme est:

```
test1
 creation
 destruction
test2
 creation
 destruction
test3
 creation
 creation
 destruction
 destruction
test4
 creation
 creation
 destruction
```

Nous voyons lors du test 4 que l'emploi de `delete[]` est obligatoire sur les tableaux. Il est possible d'utiliser la commande `delete[]` même sur un objet unique (voir test 2). D'un point de vue méthodologique, il est donc conseillé de toujours utiliser la forme de l'opérateur `delete` suivie des crochets : `delete[]`.

Le programme demandé permettant de créer une liste chaînée triée de nombres entiers :

```
==============================LISTE.H=======================================
#ifndef UserPile
#define UserPile

#include <stdio.h>

#define vrai 1
#define faux 0

class element_liste_ordonnee
{
  public:
        int keyindex;
        element_liste_ordonnee *next;
        virtual void affiche()=0;
};

class element_int : public element_liste_ordonnee
{
  public:
      int el;
      element_int(int _elem): el(_elem) { keyindex=el; };
      void affiche() { printf("%u",el); };
};

class classe_liste
{ public:

      classe_liste(): head(NULL), taille(0), lire_taille(taille) {};
      ~classe_liste();
```

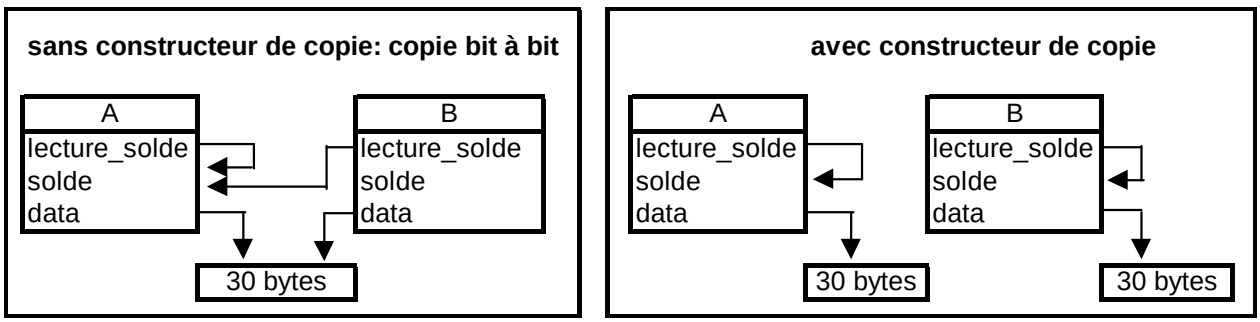
```
      const int &lire_taille;
      void vide();

      void add(element_liste_ordonnee *elem);
      element_liste_ordonnee *get(int number);

   private:
      element_liste_ordonnee *head;
      int taille;
};
#endif

================================LISTE.CPP=====================================
#include "liste.h"

classe_liste::~classe_liste() { vide(); };

void classe_liste::vide()
{
      element_liste_ordonnee*p;
      while (head!=NULL)
      {
            p=head->next;
            delete(head);
            head=p;
      };
      taille=0;
};

void classe_liste::add(element_liste_ordonnee *e)
{
      element_liste_ordonnee*temp,*p=head;
      if (head!=NULL)
      {
            while ((p->keyindex>e->keyindex) &&
                  (p->next!=NULL)              ) p=p->next;
            temp=p->next; p->next=e; e->next=temp;
      }
      else
      {
            head=e; e->next=NULL;
      }
      taille++;
};

element_liste_ordonnee*classe_liste::get(int n)
{
      element_liste_ordonnee*p=head;
      if (n>taille) return NULL;
      else
      {
            for ( ; n>1 ; n-- ) p=p->next;
            return p;
      };
};

============================LISTEMAIN.CPP=====================================
#include <stdio.h>
#include "liste.h"

classe_liste nombre;


void data_in()
{
      int number=0;
      char rep='o';
      while (rep=='o')
      {
            printf("nombre a rajouter dans la liste : ");
```

```
            scanf("%d",&number);
            nombre.add(new element_int(number));
            printf("encore ? (o/n) ");
            scanf("%c",&rep);
      };
};

void data_out(classe_liste *liste)
{
      for (int i=1; i<=liste->lire_taille; i++)
      {
            liste->get(i)->affiche(); printf("\n");
      }
};

void main()
{
      data_in();
      data_out(&nombre);
}
```

Lorsqu'un objet contient des pointeurs ou des références, il faut écrire un constructeur de copie et redéfinir l'opérateur =. En effet, la copie d'un objet se fait bit à bit. Soit l'exemple suivant:

```
class Compte
{ public:
      const int & lecture_solde;
      Compte() : lecture (Solde), Solde(0) { data=(char(*)[30])malloc(30) };
  Private:
      int solde;
      char (*data)[30];
}

   ...
 void fonction_bidon(Compte B) { // Que contient B ici ? }
   ...
 Compte A;
 fonction_bidon(A);
    ...
```

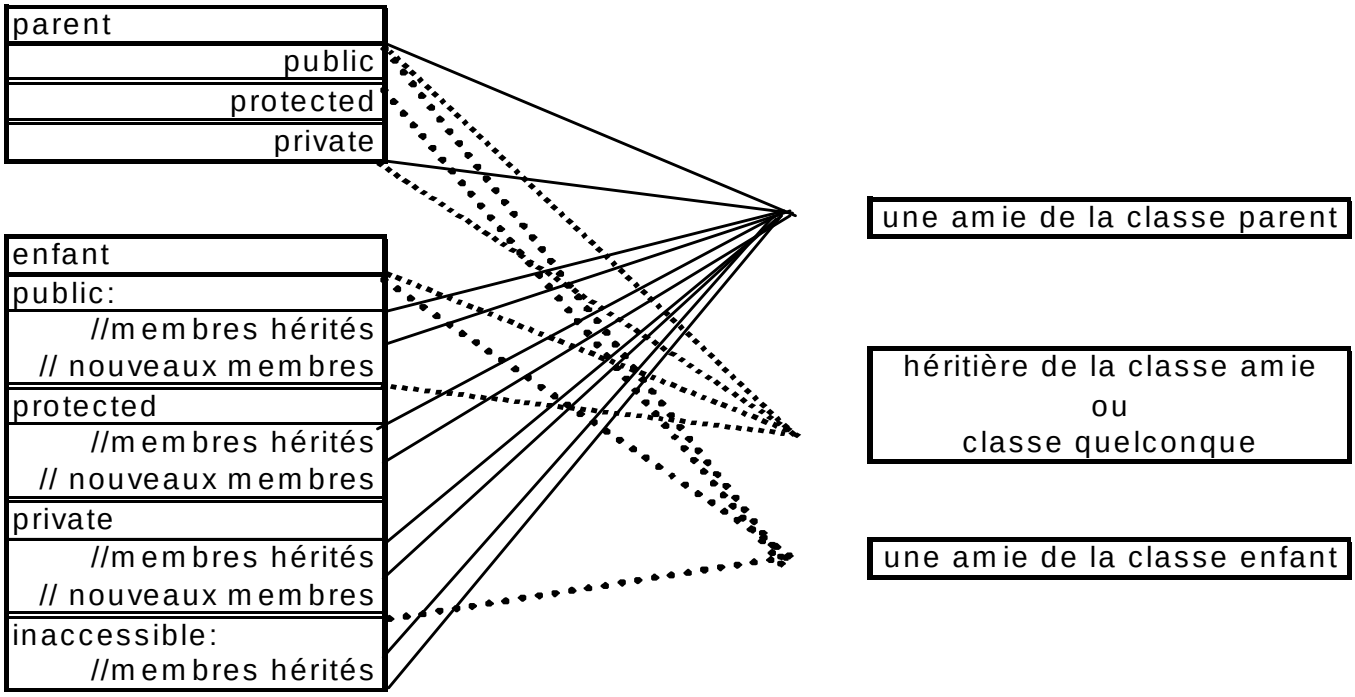
Nous avons dans le corps de la focntion_bidon :



1.2. Les Classes

1.2.1. Qualificateur PUBLIC et PRIVATE

Une classe est une structure évoluée munie de règles d'accessibilité, de régles d'héritage. Les membres d'une classe peuvent être aussi bien des membres classique de structure (variable, tableau, pointeur, référence, …) que des fonctions. Ces fonctions ayant un rapport étroit avec les membres de données de la classe. Les status *private* ou *protected* rendent un membre d'une classe innaccessible hors de sa classe excepté pour les classes déclarées *friend*. La différence entre private et protected intervient lors de l'héritage de la classe. Voici 2 tableaux résumant bien ces différentes règles:



status fonctions héritées		héritage en mode:		
		non-spécifié (private)	protected	public
status initials fonctions	public	private	protected	public
	protected	private	protected	protected
	private	non-accessible	non-accessible	non-accessible
	virtual	virtual	virtual	virtual
	static	static	static	static

1.2.2. Constructeur d'objets

Le constructeur d'un objet a pour but d'initialiser les membres de données de la classe. Lors de l'appel au contructeur le programme effectue aussi une mise a jour de la table des méthodes virtuelles de façon transparente pour l'utilisateur. Le constructeur par défaut est le constructeur ne prenant pas de paramètres. Le constructeur de copie est utilisé lors de la copie de la classe. Le destructeur est appellé implicitement avant la destruction de la classe. Il contiendra le code pour libérer la mémoire dynamiquement allouée pour la classe. Reprenons la classe *compte* du paragraphe 1.1.3 :

```
class Compte
{ public:
      const int & lecture;

      int fnct_lecture() { return Solde; };

// contructeur :
      Compte(int s=0) :
            lecture (Solde), Solde(s) {};
```

```
// constructeur de copie:
      Compte( const Compte &inter ) :
            lecture (Solde), Solde(inter.lecture) {};

//opérateur d'affectation:
      Compte() & operator = ( const Compte &inter )
      {
            Solde=inter.lecture;
            return *this;
      };

  private:
      int Solde;
}
```

nous avons :

```
void affiche_solde( Compte c5 )
{
    printf("%u",c5.lecture);
}; // appel au destructeur de c5

void main ()
{
      Compte c1,c4,          // constructeur par défaut
             c2(c1),c3=c1;   // constructeur de copie

      c4=c1;                 // surcharge de l'opérateur =
      affiche_solde(c1);     // constructeur de copie (c5=c1)

}; // appel au destructeur de c1,c2,c3,c4
```

Considérons l'exemple suivant :

```
class Dosssier
{  public:
     Compte cpt1, cpt2;
     Dossier ( int n1, int n2 )
     {
        cpt1= Compte(n1);
        cpt2= Compte(n2);
     };
};
```

Il y aura pour la création d'une instance de Dossier :

- 2 appels au constructeur par défaut de Compte
- 2 appels au constructeur à 1 argument de compte
- 2 affectation à l'aide de l'opérateur =

Il est plus simple et plus rapide d'écrire :

```
class Dosssier
{  public:
     Compte cpt1, cpt2;
     Dossier ( int n1, int n2 ) : cpt1(n1), cpt2(n2) {};
};
```

Cette fois , il y aura pour la création d'une instance de Dossier :

- 2 appels au constructeur à 1 argument de compte

Héritage:

Lors de la création d'une instance, nous avons:

- appel du contructeur de la classe mère
- appel des constructeur des membres de la classe héritière
- exécution des instructions contenues dans le corps du contructeur de l'héritière

Lors de la destruction d'une instance, nous avons:

- exécution des instructions du corps du destructeur de la classe héritière
- appel des desctructeurs des membres de la classe héritière.
- appel du destructeur de la classe mère (avec exécution du corps en premier)

Une variable membre *static* est considérée comme une variable globale. Considérons l'exemple suivant:

```
#include <stdio.h>
class Compte
{ public:
      Compte()
      {
        static int nb_compte=0;
           // les variables static sont déclarées et
           // initialisées dans le corps d'une fonction
           // membre

        numero=nb_compte++;
      }
      int numero;
}

void main()
{
   Compte c1,c2;
   printf("%u %u",c1.numero, c2.numero);
}
```

Nous aurons le résultat suivant :

```
0 1
```

1.3. Classes avec membres pointeurs /surcharge d'opérateur.

La classe demandée permettant de manipuler des matrices :

```
==========================MATRIX.H============================
class matrix
{
  friend ostream & operator << ( ostream &, matrix & );

  public:
      int &LectureX,&LectureY;
      matrix();
      matrix(int _lx,int _ly);
      matrix(int _lx,int _ly,void *t);
      matrix(const matrix &t);
      matrix & operator=  ( matrix &t);
      ~matrix();

      float  & operator[] ( int indice );
      matrix   operator+  ( matrix &a );
      matrix   operator-  ( matrix &a );
      matrix   operator*  ( matrix &b );

  private:
      int lx,ly;
      float (*head)[10]; // déclaration d'un pointeur vers un tableau de float
};

class Error_Matrix
{
  public:
      int lx1,ly1,lx2,ly2;
      Error_Matrix(int _lx1,int _ly1,int _lx2,int _ly2):
         lx1(_lx1), ly1(_ly1), lx2(_lx2), ly2(_ly2) {};
};

==========================MATRIX.CPP==========================
```

```
#include <iostream.h>
#include <malloc.h>
#include <memory.h>
#include "matrix.h"

matrix::matrix(int _lx,int _ly,void *t) :
      lx(_lx), ly(_ly), LectureX(lx), LectureY(ly)
{
      head=(float (*)[10])malloc(lx*ly*sizeof(float));
      memcpy(head,t,lx*ly*sizeof(float));
};

matrix::matrix() :
   lx(0), ly(0), LectureX(lx), LectureY(ly), head(NULL) {};

matrix::matrix(int _lx,int _ly):
      lx(_lx), ly(_ly), LectureX(lx), LectureY(ly), head(NULL)
{
      head=(float (*)[10])malloc(lx*ly*sizeof(float));
};

matrix::~matrix() { free(head); };

matrix::matrix(const matrix &t):
      lx(t.LectureX), ly(t.LectureY), LectureX(lx), LectureY(ly)
{
      head=(float (*)[10])malloc(lx*ly*sizeof(float));
      memcpy(head,t.head,lx*ly*sizeof(float));
}

matrix & matrix::operator= ( matrix &t)
{
      lx=t.LectureX; ly=t.LectureY;
      if (head!=NULL) free(head);
      head=(float (*)[10])malloc(lx*ly*sizeof(float));
      memcpy(head,t.head,lx*ly*sizeof(float));
      return *this;
}

float & matrix::operator[] ( int indice )
{
   return (*head)[indice];
};

ostream & operator << ( ostream & sortie , matrix &a )
{
   sortie<<"\n";
   for (int y=0; y<a.LectureY; y++)
   {
         for (int x=0; x<a.LectureX; x++)
              sortie<<a[x+y*a.LectureX]<<" ";
         sortie<<"\n";
   }
   return sortie;
}

matrix matrix::operator + ( matrix &b )
{
   matrix r(LectureX,LectureY);
   if ((b.LectureX!=LectureX) || (b.LectureY!=LectureY))
         throw ( Error_Matrix ( LectureX, LectureY,
                                  b.LectureX, b.LectureY ));
   else
         for (int i=0; i<(LectureX*LectureY); i++)
            r[i]=(*this)[i]+b[i];

   return r;
}

matrix matrix::operator - ( matrix &b )
{
```

```
   matrix r(LectureX,LectureY);

   if ((b.LectureX!=LectureX) || (b.LectureY!=LectureY))
         throw ( Error_Matrix ( LectureX, LectureY,
                                b.LectureX, b.LectureY ));
   else
       for (int i=0; i<(LectureX*LectureY); i++)
             r[i]=(*this)[i]-b[i];
   return r;
}

matrix  matrix::operator * ( matrix &b )
{
   matrix r(b.LectureX, LectureY);

   if (LectureX!=b.LectureY)
         throw ( Error_Matrix ( LectureX, LectureY,
                               b.LectureX, b.LectureY ));
   else
         for (int x=0,y,k; x<b.LectureX; x++)
            for (y=0; y<LectureY; y++)
              {
               r[x+y*b.LectureX]=0;
               for (k=0; k<LectureX; k++)
                  r[x+y*b.LectureX]+=((*this)[y*LectureX+k]*
                                     b[x+k*b.LectureX]     );
              };
   return r;
}

========================TESTMATRIX.CPP=========================
#include <iostream.h>
#include "matrix.h"

void main()
{
      char factice;
      float A[2*3]= { 1.,2.,3.,4.,5.,6. };
      matrix c(3,2,&A); matrix d(3,2,&A); matrix e(2,3,&A);
      cout<<c+d;
      cout<<c*e;
      try
      {
            cout<<c*d;
      }
      catch (Error_Matrix E)
      {
            cout<<"\nerreur calcul matriciel entre matrices ("
               <<E.lx1<<"x"<<E.ly1<<") et )"<<E.lx2<<"x"<<
               E.ly2<<").\n";
      }
      cin>>factice;
}
```

Le résulat du programme est :

```
2 4 6
8 10 12

22 28
49 64

erreur calcul matriciel entre matrices (3x2) et (3x2).
```

Le pointeur *this* pointe vers l'objet en cours d'exécution. Lorsque nous écrivons `(*this)[i]`, nous exécutons la surcharge de l'opérateur `[]`, pour la classe `matrix`. Nous autions pu écrire : `(*head)[i]`, mais nous perdons alors l'abstraction objet(ADT). Surcharger un opérateur est interressant quand la surcharge permet l'écriture de façon intuitive de code relatif à cet objet. Utiliser l'opérateur `[]` pour accéder à un membre d'une liste chaînée est une chose à éviter. L'opérateur `[]` est souvent utilisé quand l'accès aux données est de complexité 1. Dans le cas d'une liste chaînée, l'accès à `[n]` est de complexité n. C'est pourquoi utiliser l'opérateur `[]` pour accéder à un membre d'une liste chaînée est une chose à éviter.

1.4. Héritage

Les classes ***fichier***, ***fichier_ascii***, ***fichier_record*** demandées ainsi que la focntion ***report2el*** et une trace d'exécution:

```
========================TEST.DAT==============================
123456789abcdefghijklmnoprstuvwxyz

========================FICHIER.CPP===========================
#include <stdio.h>

//----------déclaration de FICHIER-----------------
class fichier
{ public:
      fichier(char *name);
      ~fichier();
      virtual long int taille() =0;
      virtual void affichage(int i) =0;
  protected:
      FILE *stream;
};

//----------déclaration de FICHIER_ASCII-----------

class fichier_ASCII : public fichier
{ public:
      fichier_ASCII( char *name): fichier(name) {};
      virtual long int taille() ;
      virtual void affichage(int i) ;
};

//----------déclaration de FICHIER_RECORD----------

class fichier_Record : public fichier
{ public:
      fichier_Record( char *name, int l):
        fichier(name), long_elem(l) {};
      virtual long int taille() ;
      virtual void affichage(int i) ;
  private:
      int long_elem;
};

//----------définition de FICHIER-----------------

fichier::fichier(char *name)
  { stream=fopen(name,"r"); };

fichier::~fichier()
  { fclose(stream); };


//----------définition de FICHIER_ASCII-----------
long int fichier_ASCII::taille()
{
      fpos_t tmp;
      fseek(stream,0,SEEK_END);
      fgetpos(stream,&tmp);
      return (long int)tmp;
};

void fichier_ASCII::affichage(int i)
{
      fseek(stream,i,SEEK_SET);
      printf("%c ",fgetc(stream));
};

//----------définition de FICHIER_RECORD-----------
long int fichier_Record::taille()
{
```

```
      fpos_t tmp;
      fseek(stream,0,SEEK_END);
      fgetpos(stream,&tmp);
      return ((long int)tmp / long_elem);
};

void fichier_Record::affichage(int i)
{
      char tmp[30];
      fseek(stream,i*long_elem,SEEK_SET);
      fread(tmp,sizeof(char), long_elem, stream);
      tmp[long_elem]='\0';
      printf("%s ",tmp);
};

//----------programme principal--------------------
void report2el(fichier *f)
{
      for (int i=0; i<f->taille(); i+=2)
      {
            f->affichage(i);
      };
};

void main()
{
      fichier_ASCII f1("test.dat");
      fichier_Record f2("test.dat",4);
      report2el(&f1);
      printf("\n");
      report2el(&f2);
      printf("\n");
      getchar();
};
```

Trace d'exécution :

```
1 3 5 7 9 b d f h j l n p s u w y
1234 9abc hijk prst
```