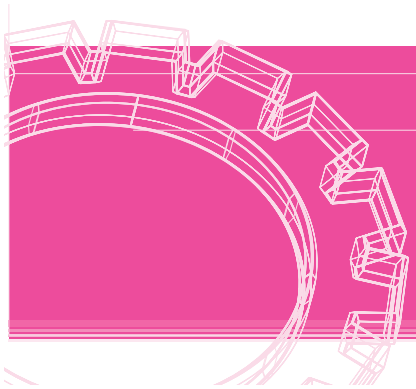
Chapitre 20

Exercices de synthèse



Exercice 142

Énoncé

Réaliser une classe nommée `set_int` permettant de manipuler des ensembles de nombres entiers. Le nombre maximal d'entiers que pourra contenir l'ensemble sera précisé au constructeur qui allouera dynamiquement l'espace nécessaire. On prévoira les opérateurs suivants (e désigne un élément de type `set_int` et n un entier :

- <<, tel que e<<n ajoute l'élément n à l'ensemble eþ;
- %, tel que `n % e` vale `1` si `n` appartient à e et `0` sinonþ;
- <<, tel que `flot << e` envoie le contenu de l'ensemble e sur le flot indiqué, sous la forme :

```
[entier1, entier2, ... entiern]
```

La fonction membre `cardinal` fournira le nombre d'éléments de l'ensemble. Enfin, on s'arrangera pour que l'affectation ou la transmission par valeur d'objets de type `set_int` ne pose aucun problème (on acceptera la duplication complète d'objets).

N.B. Le chapitre 21 vous montrera comment résoudre cet exercice à l'aide des composants standard introduits par la norme, qu'il ne faut pas chercher à utiliser ici.

Solution

Naturellement, notre classe comportera, en membres donnée, le nombre maximal (`nmax`) d'éléments de l'ensemble, le nombre courant d'éléments (`nelem`) et un pointeur sur l'emplacement contenant les valeurs de l'ensemble.

Comme notre classe comporte une partie dynamique, il est nécessaire, pour que l'affectation et la transmission par valeur se déroulent convenablement, de surdéfinir l'opérateur d'affectation et de munir notre classe d'un constructeur par recopie. Les deux fonctions membre (`operator =` et `set_int`) devront prévoir une « copie profonde » des objets. Nous utiliserons pour cela une méthode que nous avons déjà rencontrée et qui consiste à considérer que deux objets différents disposent systématiquement de deux parties dynamiques différentes, même si elles possèdent le même contenu.

L'opérateur `%` doit être surdéfini obligatoirement sous la forme d'une fonction amie, puisque son premier opérande n'est pas de type classe. L'opérateur de sortie dans un flot doit, lui aussi, être surdéfini sous la forme d'une fonction amie, mais pour une raison différente : son premier argument est de type `ostream`.

Voici la déclaration de notre classe `set_int` :

```
     /*  fichier SETINT.H  : déclaration de la classe set_int */
#include <iostream>
using namespace std ;
class set_int
{  int * adval ;                     // adresse du tableau des valeurs
   int nmax ;                        // nombre maxi d'éléments
   int nelem ;                       // nombre courant d'éléments
 public :
   set_int (int = 20) ;              // constructeur
   set_int (set_int &) ;             // constructeur par recopie
                                     //   voir remarque 1 ci-après
   set_int & operator = (set_int &) ; // opérateur d'affectation
                                     //   voir remarque 2 ci-après
   ~set_int () ;                      // destructeur
   int cardinal () ;                  // cardinal de l'ensemble
   set_int & operator << (int) ;      // ajout d'un élément
   friend int operator % (int, set_int &) ; // appartenance d'un élément
                                     //    voir remarque 3 ci-après
               // envoi ensemble dans un flot, voir remarque 4
   friend ostream & operator << (ostream &, set_int &) ;
} ;
```

Voici ce que pourrait être la définition de notre classe (les points délicats sont commentés au sein même des instructions) :

```
#include "setint.h"
#include <iostream>
using namespace std ;
```

```
   /*************** constructeur ********************/
set_int::set_int (int dim)
{  adval = new int [nmax = dim] ;    // allocation tableau de valeurs
   nelem = 0 ;
}
   /****************** destructeur ******************/
set_int::~set_int ()
{  delete adval ;                    // libération tableau de valeurs
}
 /********** constructeur par recopie *************/
set_int::set_int (set_int & e)
{  adval = new int [nmax = e.nmax] ; // allocation nouveau tableau
   nelem = e.nelem ;
   int i ;
   for (i=0 ; i<nelem ; i++)         // copie ancien tableau dans nouveau
      adval[i] = e.adval[i] ;
}
   /************ opérateur d'affectation ************/
set_int & set_int::operator = (set_int & e)//commentaires fait pour b = a
{  if (this != &e)                          // on ne fait rien pour a = a
     { delete adval ;                // libération partie dynamique de b
       adval = new int [nmax = e.nmax] ; // allocation nouvel ensemble pour a
       nelem = e.nelem ;                 //      dans lequel on recopie
       int i ;                           //      entièrement l'ensemble b
       for (i=0 ; i<nelem ; i++)         //      avec sa partie dynamique
          adval[i] = e.adval[i] ;
     }
   return * this ;
}
   /************ fonction membre cardinal ***********/
int set_int::cardinal ()
{  return nelem ;
}

   /************ opérateur d'ajout << ***************/
set_int & set_int::operator << (int nb)
{        // on examine si nb appartient déjà à l'ensemble
         //   en utilisant l'opérateur %
         // s'il n'y appartient pas, et s'il y a encore de la place
         //   on l'ajoute à l'ensemble
   if ( ! (nb % *this) && nelem < nmax )  adval [nelem++] = nb ;
   return (*this) ;
}
```

```
   /*********** opérateur d'appartenance % **********/
int operator % (int nb, set_int & e)
{  int i=0 ;
         // on examine si nb appartient déjà à l'ensemble
         //  (dans ce cas i vaudra nele en fin de boucle)
   while ( (i<e.nelem) && (e.adval[i] != nb) ) i++ ;
   return (i<e.nelem) ;
}

   /****** opérateur << pour sortie sur un flot *****/
ostream & operator << (ostream & sortie, set_int & e)
{  sortie << "[ " ;
   int i ;
   for (i=0 ; i<e.nelem ; i++)
      sortie << e.adval[i] << " " ;
   sortie << "]" ;
   return sortie ;
}
```

Remarques

1. On pourrait ajouter le qualificatif `const` au constructeur par recopie, ce qui autoriserait l'initialisation d'un objet par un objet constant. Mais compte tenu des possibilités d'utilisation de l'autre constructeur dans des conversions implicites, on autoriserait du même coup l'initialisation par un entier ou un flottant, ce qui n'est guère satisfaisant ; on pourrait cependant interdire de telles possibilités en utilisant le mot-clé `explicit` dans la déclaration du constructeur.
2. La transmission de la valeur de retour de l'opérateur d'affectation n'est utile que si l'on souhaite permettre les affectations multiples. Il n'est pas indispensable de transmettre l'argument et la valeur de retour par référence, mais cela évite les recopies. On pourrait ici déclarer constant l'unique argument, ce qui autoriserait l'utilisation d'un objet constant en second opérande de l'affectation (moyennant une recopie). Mais, du même coup, compte tenu des possibilités de conversions implicites, on autoriserait également l'utilisation d'un entier ou d'un flottant, ce qui n'est pas nécessairement souhaité ; là encore, ces possibilités pourraient être interdites, moyennant l'utilisation appropriée du mot-clé `explicit`.
3. On pourrait ajouter le qualificatif `const` à l'opérateur `%`, ce qui permettrait de travailler sur un ensemble constant ; néanmoins, dans ce cas, il y aurait transmission d'une copie de cet ensemble à la fonction, ce qui n'est plus très efficace !
4. La remarque précédente s'applique à l'opérateur `<<`.
5. Notez qu'ici il n'est pas possible d'agrandir l'ensemble au-delà de la limite qui lui a été impartie lors de sa construction. Il serait assez facile de remédier à cette lacune en modifiant sensiblement la fonction d'ajout d'un élément (`operator <<`). Il suffirait, en effet, qu'elle prévoie, lorsque la limite est atteinte, d'allouer un nouvel emplacement dynamique, par exemple d'une taille double de l'emplacement existant, d'y recopier l'actuel contenu et de libérer l'ancien emplacement (en actualisant convenablement les membres donnée de l'objet).

Voici un exemple de programme utilisant la classe `set_int`, accompagné du résultat fourni par son exécution :

```
   /************* test de la classe set_int *********/
#include "setint.h"
#include <iostream>
using namespace std ;

main()
{  void fct (set_int) ;
   void fctref (set_int &) ;
   set_int ens ;
   cout << "donnez 10 entiers \n" ;
   int i, n ;
   for (i=0 ; i<10 ; i++)
      { cin >> n ;
        ens << n ;
      }
   cout << "il y a : " << ens.cardinal () << " entiers différents\n" ;
   cout << "qui forment l\'ensemble : " << ens << "\n" ;
   fct (ens) ;
   cout << "au retour de fct, il y en a " << ens.cardinal () << "\n" ;
   cout << "qui forment l\'ensemble : " << ens << "\n" ;
   fctref (ens) ;
   cout << "au retour de fctref, il y en a " << ens.cardinal () << "\n" ;
   cout << "qui forment l\'ensemble : " << ens << "\n" ;
   cout << "appartenance de -1  : " <<  -1 % ens << "\n" ;
   cout << "appartenance de 500 : " << 500 % ens << "\n" ;
   set_int ensa, ensb ;
   ensa = ensb = ens ;
   cout << "ensemble a : " << ensa << "\n" ;
   cout << "ensemble b : " << ensb << "\n" ;
}

void fct (set_int e)
{  cout << "ensemble reçu par fct : " << e << "\n" ;
   e << -1 << -2 << -3 ;
}

void fctref (set_int & e)
{  cout << "ensemble reçu par fctref : " << e << "\n" ;
   e << -1 << -2 << -3 ;
}
```

```
donnez 10 entiers
3 5 3 1 8 5 1 7 7 3
il y a : 5 entiers différents
qui forment l'ensemble : [ 3 5 1 8 7 ]
ensemble reçu par fct : [ 3 5 1 8 7 ]
au retour de fct, il y en a 5
qui forment l'ensemble : [ 3 5 1 8 7 ]
ensemble reçu par fctref : [ 3 5 1 8 7 ]
au retour de fctref, il y en a 8
qui forment l'ensemble : [ 3 5 1 8 7 -1 -2 -3 ]
appartenance de -1  : 1
appartenance de 500 : 0
ensemble a : [ 3 5 1 8 7 -1 -2 -3 ]
ensemble b : [ 3 5 1 8 7 -1 -2 -3 ]
```

Exercice 143

Énoncé

Créer une classe `vect` permettant de manipuler des « vecteurs dynamiques » d'entiers, c'est-à-dire des tableaux d'entiers dont la dimension peut être définie au moment de leur création (une telle classe a déjà été partiellement réalisée dans l'exercice 39). Cette classe devra disposer des opérateurs suivants :

- `[]` pour l'accès à une des composantes du vecteur, et cela aussi bien au sein d'une expression qu'à gauche d'une affectation (mais cette dernière situation ne devra pas être autorisée sur des « vecteurs constants ») ;
- ==, tel que si `v1` et `v2` sont deux objets de type `vect`, `v1==v2` prenne la valeur 1 si `v1` et `v2` sont de même dimension et ont les mêmes composantes et la valeur `0` dans le cas contraire ;
- <<, tel que `flot<<v` envoie le vecteur v sur le flot indiqué, sous la forme :

```
<entier1, entier2, ... , entiern>
```

De plus, on s'arrangera pour que l'affectation et la transmission par valeur d'objets de type `vect` ne pose aucun problème ; pour ce faire, on acceptera de dupliquer complètement les objets concernés.

N.B. Le chapitre 21 vous montrera comment résoudre cet exercice à l'aide des composants standard introduits par la norme, qu'il ne faut pas chercher à utiliser ici.

Solution

Rappelons que lorsqu'on définit des objets constants, il n'est pas possible de leur appliquer une fonction membre publique, sauf si cette dernière a été déclarée avec le qualificatif `const` (auquel cas une telle fonction peut indifféremment être utilisée avec des objets constants ou non constants). Pour obtenir l'effet demandé de l'opérateur `[]`, lorsqu'il est appliqué à un vecteur constant, il est nécessaire d'en prévoir deux définitions dont l'une s'applique aux vecteurs constants ; pour éviter qu'on ne puisse, dans ce cas, l'utiliser à gauche d'une affectation, il est nécessaire qu'elle renvoie son résultat par valeur (et non par adresse comme le fera la fonction applicable aux vecteurs non constants).

Voici la déclaration de notre classe :

```
#include <iostream>
using namespace std ;
class vect
{  int nelem ;                      // nombre de composantes du vecteur
   int * adr ;                      // pointeur sur partie dynamique
 public :
   vect (int n=1) ;                 // constructeur "usuel"
   vect (vect & v) ;                // constructeur par recopie,
                                    //   voir remarque 1 ci-après
   ~vect () ;                       // destructeur
   friend ostream & operator << (ostream &, vect &) ; // sortie sur un flot
   vect & operator = (vect & v) ;   // surdéfinition opérateur affectation
                                    //   voir remarque 2 ci-après
   int & operator [] (int i) ;      // surdef [] pour vect non constants
   int   operator [] (int i) const ;   // surdef [] pour vect constants
} ;
```

Remarques

1. On pourrait ajouter le qualificatif `const` au constructeur par recopie, ce qui autoriserait l'initialisation d'un vecteur par un vecteur constant. Mais compte tenu des possibilités de l'autre constructeur dans des conversions implicites, on autoriserait du même coup l'initialisation par un entier ou un flottant, ce qui n'est guère satisfaisant ; on pourrait cependant interdire de telles possibilités en utilisant le mot-clé `explicit` dans la déclaration du constructeur.

2. La transmission de la valeur de retour de l'opérateur d'affectation n'est utile que si l'on souhaite permettre les affectations multiples. Il n'est pas indispensable de transmettre l'argument et la valeur de retour par référence, mais cela évite les recopies. On pourrait ici déclarer constant l'unique argument, ce qui autoriserait l'utilisation d'un objet constant en second opérande de l'affectation (moyennant une recopie). Mais, du même coup, compte tenu des possibilités de conversions implicites, on autoriserait également l'utilisation d'un entier ou d'un flottant, ce qui n'est pas nécessairement souhaité ; là encore, ces possibilités pourraient être interdites, moyennant l'utilisation appropriée du mot-clé `explicit`.

Voici la définition des différentes fonctions :

```
#include "vect.h"
#include <iostream>
using namespace std ;

vect::vect (int n)                  // constructeur "usuel"
     { adr = new int [nelem = n] ;
     }
vect::vect (vect & v)               // constructeur par recopie
     { adr = new int [nelem = v.nelem] ;
       int i ;
       for (i=0 ; i<nelem ; i++)
           adr[i] = v.adr[i] ;
     }
vect::~vect ()                    // destructeur
     { delete adr ;
     }
vect & vect::operator = (vect & v) // surdéfinition opérateur affectation
     { if (this != &v)             // on ne fait rien pour a=a
         { delete adr ;
           adr = new int [nelem = v.nelem] ;
           int i ;
           for (i=0 ; i<nelem ; i++)
             adr[i] = v.adr[i] ;
         }
       return * this ;
     }
int & vect::operator [] (int i)   // surdéfinition opérateur []
     { return adr[i] ;
     }
int   vect::operator [] (int i) const // surdéfinition opérateur [] pour cst
     { return adr[i] ;
     }
ostream & operator << (ostream & sortie, vect & v)
     { sortie << "<" ;
       int i ;
       for (i=0 ; i<v.nelem ; i++) sortie << v.adr[i] << " " ;
       sortie << ">" ;
       return sortie ;
     }
```

À titre indicatif, voici un petit programme utilisant la classe `vect`, accompagné du résultat fourni par son exécution :

```
#include "vect.h"
#include <iostream>
using namespace std ;

main()
{  int i ;
   vect v1(5), v2(10) ;
   for (i=0 ; i<5 ; i++) v1[i] = i ;
   cout << "v1 = " << v1 << "\n" ;
   for (i=0 ; i<10 ; i++) v2[i] = i*i ;
   cout << "v2 = " << v2 << "\n" ;
   v1 = v2 ;
   cout << "v1 = " << v1 << "\n" ;
   vect v3 = v1 ;
   cout << "v3 = " << v3 << "\n" ;
   vect v4 = v2 ;
   cout << "v4 = " << v4 << "\n" ;
   // const vect w(3) ; w[2] = 5 ;  // conduit bien à erreur compilation
}
```

```
v1 = <0 1 2 3 4 >
v2 = <0 1 4 9 16 25 36 49 64 81 >
v1 = <0 1 4 9 16 25 36 49 64 81 >
v3 = <0 1 4 9 16 25 36 49 64 81 >
v4 = <0 1 4 9 16 25 36 49 64 81 >
```

Exercice 144

Énoncé

Réaliser une classe nommée `bit_array` permettant de manipuler des tableaux de bits (autrement dit, des tableaux dans lesquels chaque élément ne peut prendre que l'une des deux valeurs `0` ou `1`). La taille d'un tableau (c'est-à-dire le nombre de bits) sera définie lors de sa création (par un argument passé à son constructeur). On prévoira les opérateurs suivants :

- `+=`, tel que `t+=n` mette à `1` le bit de rang `n` du tableau `t` ;
- `-=`, tel que `t-=n` mette à `0` le bit de rang `n` du tableau `t` ;

- `[]`, tel que l'expression `t[i]` fournisse la valeur du bit de rang `i` du tableau `t` (on ne prévoira pas, ici, de pouvoir employer cet opérateur à gauche d'une affectation, comme dans `t[i] = ...)`;
- `++`, tel que `t++` mette à `1` tous les bits de `t` ;
- `--`, tel que `t--`mette à `0` tous les bits de `t` ;
- `<<`, tel que `flot << t` envoie le contenu de *t* sur le flot indiqué, sous la forme :

```
<* bit1, bit2, ... bitn *>
```

On fera en sorte que l'affectation et la transmission par valeur d'objets du type `bit_array` ne pose aucun problème.

N.B. Le chapitre 21 vous montrera comment résoudre cet exercice à l'aide des composants standard introduits par la norme, qu'il ne faut pas chercher à utiliser ici.

Solution

Si l'on cherche à minimiser l'emplacement mémoire utilisé pour les objets de type `bit_array`, il est nécessaire de n'employer qu'un seul bit pour représenter un « élément » d'un tableau. Ces bits devront donc être regroupés, par exemple à raison de `CHAR_BIT` (défini dans `limits.h`) bits par caractère.

Manifestement, il faut prévoir que l'emplacement destiné à ces différents bits soit alloué dynamiquement en fonction de la valeur fournie au constructeur : pour `n` bits, il faudra `n/CHAR_BIT+1` caractères.

En membres donnée, il nous suffit de disposer d'un pointeur sur l'emplacement dynamique en question, ainsi que du nombre de bits du tableau. Pour simplifier certaines des fonctions membre, nous prévoirons également de conserver le nombre de caractères correspondant.

Les opérateurs `+=`, `-=`, `++` et `--` peuvent être définis indifféremment sous la forme d'une fonction membre ou d'une fonction amie. Ici, nous avons choisi des fonctions membre. L'énoncé ne précise rien quant au résultat fourni par ces 4 opérateurs. En fait, on pourrait prévoir qu'ils restituent le tableau après qu'ils y ont effectué l'opération voulue, mais en pratique, cela semble de peu d'intérêt. Ici, nous avons donc simplement prévu que ces opérateurs ne fourniraient aucun résultat.

Pour que `[]` ne soit pas utilisable dans une affectation de la forme `t[i] = ...,` il suffit de prévoir qu'il fournisse son résultat par valeur (et non par référence comme on a généralement l'habitude de le faire).

Naturellement, ici encore, l'énoncé nous impose de surdéfinir l'opérateur d'affectation et de prévoir un constructeur par recopie.

Voici ce que pourrait être la déclaration de notre classe `bit_array` :

```
  /* fichier bitarray.h : déclaration de la classe bit_array */
#include <iostream>
using namespace std ;
class bit_array
{   int nbits ;       // nombre courant de bits du tableau
    int ncar ;        // nombre de caractères nécessaires (redondant)
    char * adb ;      // adresse de l'emplacement contenant les bits
  public :
    bit_array (int = 16) ;      // constructeur usuel
    bit_array (bit_array &) ;   // constructeur par recopie,
                                //   voir remarque 1 ci-après
    ~bit_array () ;             // destructeur
                                // les opérateurs binaires
    bit_array & operator = (bit_array &) ;   // affectation,
                                             //   voir remarque 2 ci-après
    int operator [] (int) ;                  // valeur d'un bit
    void operator += (int) ;                 // activation d'un bit
    void operator -= (int) ;                 // désactivation d'un bit
                                             // envoi sur flot
    friend ostream & operator << (ostream &, bit_array &) ;

                                // les opérateurs unaires
    void operator ++ () ;                          // mise à 1
    void operator -- () ;                          // mise à 0
    void operator ~ () ;                           // complément à 1
} ;
```

Remarques

1. On pourrait ajouter le qualificatif `const` au constructeur par recopie, ce qui autoriserait l'initialisation d'un objet `bit_array` par un objet constant. Mais compte tenu des possibilités de conversions implicites, on autoriserait du même coup l'initialisation par un entier ou par un flottant, ce qui n'est guère satisfaisant ; on pourrait cependant interdire de telles possibilités en utilisant le mot-clé `explicit` dans le constructeur à un argument de type `int`.

2. La présence d'une valeur de retour dans l'opérateur d'affectation n'est utile que pour permettre les affectations multiples. La transmission par référence de l'unique argument n'est pas obligatoire. On pourrait ajouter le qualificatif `const` pour autoriser l'affectation d'un objet constant (moyennant alors une recopie). Mais, du même coup, compte tenu des possibilités de conversions implicites, on autoriserait également l'utilisation d'un entier ou d'un flottant, ce qui n'est pas nécessairement satisfaisant ; là encore, ces possibilités pourraient être interdites, moyennant l'utilisation appropriée du mot-clé `explicit`.

Voici la définition des différentes fonctions.

```
#include "bitarray.h"
#include <climits>
#include <iostream>
using namespace std ;

bit_array::bit_array (int nb)
{  nbits = nb ;
   ncar = nbits / CHAR_BIT + 1 ;
   adb = new  char [ncar] ;
   int i ;
   for (i=0 ; i<ncar ; i++) adb[i] = 0 ;    // raz
}
bit_array::bit_array (bit_array & t)
{  nbits = t.nbits ; ncar = t.ncar ;
   adb = new  char [ncar] ;
   int i ;
   for (i=0 ; i<ncar ; i++) adb[i] = t.adb[i] ;
}
bit_array::~bit_array()
{  delete adb ;
}

bit_array & bit_array::operator = (bit_array & t)
{  if (this != & t)      //  on ne fait rien pour t=t
     { delete adb ;
       nbits = t.nbits ; ncar = t.ncar ;
       adb = new char [ncar] ;
       int i ;
       for (i=0 ; i<ncar ; i++)
          adb[i] = t.adb[i] ;
     }
   return *this ;
}
int bit_array::operator [] (int i)
{       // le bit de rang i s'obtient en considérant le bit
        //   de rang i % CHAR_BIT du caractère de rang i / CHAR_BIT
    int carpos = i / CHAR_BIT ;
    int bitpos = i % CHAR_BIT ;
    return  ( adb [carpos] >> CHAR_BIT - bitpos -1 ) & 0x01 ;
}
void bit_array::operator += (int i)
{    int carpos = i / CHAR_BIT ;
     if (carpos < 0 || carpos >= ncar) return ;    // protection
     int bitpos = i % CHAR_BIT ;
     adb [carpos] |= (1 << (CHAR_BIT - bitpos - 1) ) ;
}
```

```
void bit_array::operator -= (int i)
{    int carpos = i / CHAR_BIT ;
     if (carpos < 0 || carpos >= ncar) return ;    // protection
     int bitpos = i % CHAR_BIT ;
     adb [carpos] &= ~(1 << CHAR_BIT - bitpos - 1) ;
}
ostream & operator << (ostream & sortie, bit_array & t)
{   sortie << "<* " ;
    int i ;
    for (i=0 ; i<t.nbits ; i++)
       sortie << t[i] << " " ;
    sortie << "*>" ;
    return sortie ;
}
void bit_array::operator ++ ()
{   int i ;
    for (i=0 ; i<ncar ; i++) adb[i] = 0xFFFF ;
}
void bit_array::operator -- ()
{   int i ;
    for (i=0 ; i<ncar ; i++) adb[i] = 0 ;
}

void bit_array::operator ~ ()
{   int i ;
    for (i=0 ; i<ncar ; i++) adb[i] = ~ adb[i] ;
}
```

Voici un programme d'essai de la classe `bit_array`, accompagné du résultat fourni par son exécution :

```
   /* programme d'essai de la classe bit_array */
main ()
{ bit_array t1 (34) ;
  cout << "t1 = " << t1 << "\n" ;
  t1 +=3 ; t1 += 0 ; t1 +=8 ; t1 += 15 ; t1 += 33 ;
  cout << "t1 = " << t1 << "\n" ;
  t1-- ;
  cout << "t1 = " << t1 << "\n" ;
  t1++ ;
  cout << "t1 = " << t1 << "\n" ;
  t1 -= 0 ; t1 -= 3 ; t1 -= 8 ; t1 -= 15 ; t1 -= 33 ;
  cout << "t1 = " << t1 << "\n" ;
  cout << "t1 = " << t1 << "\n" ;
  bit_array t2 (11), t3 (17) ;
  cout << "t2 = " << t2 << "\n" ;
  t2 = t3 = t1 ;
  cout << "t3 = " << t3 << "\n" ;
}
```

```
t1 = <* 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 *>
t1 = <* 1 0 0 1 0 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 1 *>
t1 = <* 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 *>
t1 = <* 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 *>
t1 = <* 0 1 1 0 1 1 1 1 0 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 0 *>
t1 = <* 0 1 1 0 1 1 1 1 0 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 0 *>
t2 = <* 0 0 0 0 0 0 0 0 0 0 0 *>
t3 = <* 0 1 1 0 1 1 1 1 0 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 0 *>
```

Exercice 145

Énoncé

La capacité des nombres entiers est limitée par la taille du type `longint`. Créer une classe `big_int` permettant de manipuler des nombres entiers de **valeur absolument quelconque**.

Pour ne pas alourdir l'exercice, on se limitera à des nombres sans signe et à l'opération d'addition ; on s'arrangera toutefois pour que des expressions mixtes (c'est-à-dire mélangeant des objets de type `long_int` avec des entiers usuels) aient un sens.

On définira l'opérateur << pour qu'il permette d'envoyer un objet de type `big_int` sur un flot. Parmi les différents constructeurs, on en prévoira un avec un argument de type chaîne de caractères, correspondant aux chiffres d'un « grand entier ».

On fera en sorte que l'affectation et la transmission par valeur d'objets de type `big_int` ne posent aucun problème.