Examen de langage C++ et calcul scientifique

2ème anné de maîtrise, Grenoble

Pierre Saramito

mars 2006

Examen commun aux options :
– ingénierie de la modélisation et simulation numérique
– mathématiques appliquées

Le sujet d'examen comporte deux parties indépendantes. De plus, la plupart des questions sont indépendantes les unes des autres, bien que chaque partie ait une unité de thème.

Les documents sont autorisés.

Partie 1 : vecteurs creux

De façon analogue aux matrices creuses, nous pouvons définir des vecteurs creux. Ils représentent en quelque sorte la ligne ou la colonne d'une matrice creuse : seuls les éléments non-nuls seront représentés. Ce type de vecteur est particulièrement utile lorsque nous travaillon dans un espace de dimension n de grande taille (un million par exemple) et que seul un petit nombre, noté $\mathrm{nnz}(x)$, ($\mathrm{nnz}(x) = 5$ ou 10 par exemple) des éléments du vecteur x sont non nuls.

$$\mathrm{nnz}(x) = \mathrm{card}\{i \in [1, n];\ x_i \neq 0\}$$

Ce type de situation arrive fréquement en calcul scientifique : météo, mécanique des fluides, etc. Dans ce cas, nous souhaitons que les calcul de $z = x + y$ ou $z = \lambda x$ prennent de l'ordre de 5 ou 10 opérations au lieu d'un millions d'additions d'éléménts presque tous nuls...

Pour cela nous introduisons la classe suivante :

```
template <class T>
class sparse_array {
  public:
    sparse_array(size_t n0 = 0);
    // ...
  protected:
    valarray<T>      val;
    valarray<size_t> idx;
    size_t           n;
};
```

Question 1 : À titre d'exemple, considérons les vecteurs suivants, pour $n = 6$:

$$\begin{array}{cccccccc} \text{x} & = & (1, & 2, & 0, & \text{-7}, & 0, & 0) \\ \text{y} & = & (0, & 3, & 5, & 7, & 11, & 0) \end{array}$$

Donner pour ces deux vecteurs la représentation creuse `(val,idx)` des valeurs non nulles et des indices associés.

Question 2 : Écrire le constructeur d'un vecteur de taille n, où n est passé en argument du constructeur, et dont tous les éléments initialisés à zéro ($\text{nnz} = 0$).

Question 3 : Écrire le constructeur de copie et l'opérateur d'affectation de cette classe.

Question 4 : Définir la fonction membre `size`, qui renvoie n et la fonction `nnz` qui renvoie le nombre d'éléments non nuls du vecteur.

Question 5 : Écrire une fonction amie `operator*` qui permet de calculer λx, où x est un vecteur creux et λ un scalaire, en exactement $\text{nnz}(x)$ multiplications.

Question 6 : Donner, pour les deux vecteurs x et y de la question 1, la représentation creuse `(val,idx)` du vecteur $z = x + y$. Combien vaut $\text{nnz}(z)$?

Question 7 : Écrire une fonction membre `add_nnz` de la classe qui calcule le nombre d'éléments non-nuls de la somme $x + y$ de deux vecteurs creux x et y en $\mathcal{O}(\text{nnz}(x) + \text{nnz}(y))$ opérations.

Note : *prendre soin de vérifier que l'algorithme proposé donne un résultat correct pour les vecteurs x et y de la question 1.*

Question 8 : Écrire une fonction `operator+` d'addition de deux vecteurs creux x et y en $\mathcal{O}(\text{nnz}(x) + \text{nnz}(y))$ opérations.

Note : *vérifier également que le résultat est correct pour les vecteurs x et y de la question 1.*

Question 9 : Proposer un format de fichier pour les vecteurs creux, en s'inspirant du format de fichier utilisé pour les matrices creuses. **Écrire** les fonctions d'entrée-sortie correpondantes.

Partie 2 : promotion de type

Il est possible d'effectuer des aditions, soustractions, multiplication ou divisions entre un nombre complexe et un nombre du type virgule flottante associé. Ces opérations sont spécifiées dans la classe `complex` de la librairie standard du `C++` de la façon suivante :

```
template <class T>
class complex {
  public:
    complex(const T& re = 0, const T& im = 0);
    const T& re() const { return real; }
    const T& im() const { return imag; }
    // ...
  protected:
    T real, imag;
};
```

La classe `T`, passée en paramètre du modèle de classe, représente le type de virgule flottante.

Cette classe est complétée par les fonctions suivantes, également données par la librairie standard :

```
template <class T>
complex<T> friend operator+ (const complex<T>& z1, const complex<T>& z2) {
  return complex<T>(z1.re() + z2.re(), z1.im() + z2.im()); }
template <class T>
complex<T> friend operator+ (const T& x, const complex<T>& z) {
  return complex<T>(x + z.re(), z.im()); }
template <class T>
complex<T> friend operator+ (const complex<T>& z, const T& x);
  return complex<T>(z.re() + x, z.im()); }
```

Le language C++ possède trois types à virgule flottante, par ordre de précision croissante : `float`, `double` et `long double`, et qui représentent la simple, double et triple ou quadruple précision. Ces types peuvent être combinées entre eux dans des expressions lorsqu'il n'y a pas perte de précision :

```
float   x1 = 1.0;
double  x2 = 2.0;
double  x3 = x1 + x2;
```

Question 1 : Considérer l'extrait de code suivant :

```
complex<float>   z1 (0.0, 1.0);
complex<double>  z2 (1.0, 2.0);
complex<double>  z3 = z1 + z2;
```

Expliquez pourquoi ce code conduit à un échec à la compilation.

Question 2 : La promotion de type dans une expression à virgule flottante est définie par une relation de la forme :

$$\texttt{float} + \texttt{double} \longrightarrow \texttt{double}$$

Spécifiez complètement la promotion des types à virgule flottante dans le tableau suivant :

+	float	double	long double
float			
double			
long double			

Question 3 : La promotion de type entre deux types `T1` et `T2` sera définie par `typename promote<T1,T2> : :type`. La classe promote est défine par :

```
class undefined {};

template <typename T1, typename T2>
class promote {
  public:
    typedef undefined type;
};
```

Par défaut, la promotion entre deux types quelconques est `undefined`. Les promotions effectives sont définies par des *spécialisations* de la classe `promote`, de la forme :

```
template <> class promote<float,double> {
  public:
    typedef double type;
};
template <> class promote<double,double> {
  public:
    typedef double type;
};
// ...
```

Notez que la spécialsation n'effectue à l'aide de la déclaration `template<>` et en précisant explicitement les types `T1` et `T2`. **Définir** l'opérations d'addition générale $z_1 + z_2$ entre deux nombres complexes z_1 et z_2 utilisant des représentations flottantes a priori différentes. On supposera que la classe `promote` a été complétée (inutile de développer les neuf combinaisons en répondant à cette question).

Question 4 : Définir l'opérations d'addition $x + z$ générale entre un nombre un nombre flottant x et un complexe z associé à une représentation flottante à priori différente. De même, **définir** $z + x$.

Question 5 : Dans le cours, nous avons étudié la classe polynôme, de la forme :

```
template <class T>
class polynom {
  public:
  template <class U>
  friend polynom<U> operator+ (const polynom<U>& p, const polynom<U>& q);
  // ...
};
```

Expliquez pourquoi le code suivant conduit à un échec à la compilation :

```
symbol           X;
polynom<float>   p1 = 1;
polynom<double>  p2 = 3*X;
polynom<double>  p3 = p1 + p2;
```

Question 6 : Utilisant la classe `promote`, **re-définir** l'opération d'addition de façon à ce que ce code fonctionne.

Question 7 : Quelles sont les modifications à apporter aux classes précédentes pour que le code suivant fonctionne :

```
symbol           X;
complex<float>   i (0,1);
polynom<complex<float> >  p1 = 1 + i;
polynom<double>           p2 = 3*X;
polynom<complex<double> > p3 = p1 + p2;
```