Informatique III: Programmation en C++

Lundi 16 Janvier 2006

Expressions booléennes

Introduction

De la même manière qu'une formule peut produire une valeur numérique quand elle est évaluée, elle peut produire une valeur de vérité vrai / faux, aussi appelée valeur booléenne, de type `bool` en C++.

Il existe par défaut deux valeurs constantes, `true` et `false`, qui correspondent aux deux valeurs que peut prendre une expression ou une variable de type `bool`.

Exemple

```
1  bool b;
2  int x = 14, y = 15;
3  b = x > y;
```

Opérateurs booléens

Les opérateurs de comparaisons de valeurs numériques (<, ==, etc.) renvoient des booléens. Des plus, les opérateurs `||` et `&&` calculent respectivement le “ou” et le “et”, l’opérateur `!` est l’opérateur de négation qui transforme `true` en `false` et réciproquement.

Expression	Valeur
`true || false`	`true`
`true && false`	`false`
`false || false`	`false`
`true && true`	`true`
`!true`	`false`
`!false`	`true`

Priorité des opérateurs

Les opérateurs booléens ont une priorité plus faible que les opérateurs de comparaison, qui ont eux-même une priorité plus faible que les opérateurs de calculs numériques.

Par exemple

```
1 x + 3 < 5 || y - 3 > 10
```

et

```
1 ((x + 3) < 5) || ((y - 3) > 10)
```

sont égales. **Il ne faut hésiter à mettre plus de parenthèses que moins.**

Exercice

Que valent

```
(12 < 5) || (13 > 3)
!(10 + 12 < 4) && (3 > 5)
```


Correction

Que valent

```
(12 < 5) || (13 > 3)      true
!(10 + 12 < 4) && (3 > 5)  false
```

Fonctions booléennes

De la même manière qu'une fonction peut renvoyer une valeur numérique (`int` ou `float`), elle peut renvoyer une valeur booléenne

```
bool est_parfait(int n) {
  int s = 0;
  for(int k = 1; k < n; k++)
    if(n % k == 0) s += k;
  return n == s;
}
```


Tests

Les tests dans `if()`, `while`, `do` et `for` sont en réalité des expressions booléennes. Par exemple pour trouver le premier nombre parfait plus grand que 10,000.

```
  int n = 10000;
  do {
    n++;
  } while(!est_parfait(n));
```


Ensembles d'entiers

Introduction

Nous allons voir pratiquement comment le C++ permet de manipuler abstraitement des structures de données de manière unifiée.

Pour cela nous allons construire des ensembles d'entiers de plusieurs manière différentes (avec des listes ou des tableaux par exemple), et montrer comment l'on peut travailler avec – et en particulier les parcourir – de manière unifiée.

Avec ce que nous avons vu jusqu'ici, nous pouvons définir un ensemble d'entiers:

```
1  class EnsembleEntiers {
2  public:
3    virtual bool contient_entier(int k) = 0;
4    virtual void ajouter_entier(int k) = 0;
5  };
```

Il faudra ensuite hériter de cette classe pour construire un ensemble réellement utilisable.

Avec cette défition nous pouvons néanmoins par exemple écrire

```
1  int combien_dans_1n(int n, EnsembleEntiers *ee) {
2    int nb = 0;
3    for(int i = 1; i <= n; i++)
4      if(ee->contient_entier(i)) nb++;
5    return nb;
6  }
```


Pour que tout cela soit utilisable, il faut contruire des classes qui héritent de `EnsembleEntiers` et dans lesquelles les méthodes sont effectivement programmées.

Nous allons d'abord en construire deux en partant de zéro, puis une troisième en héritant de `ListeEntiers`.

Ensemble de taille finie

```
1  class EnsembleEntiersFini : public EnsembleEntiers {
2    int *contenu;
3    int taille_max, taille;
4  public:
5    EnsembleEntiersFini(int t);
6    ~EnsembleEntiersFini();
7    virtual bool contient_entier(int k);
8    virtual void ajouter_entier(int k);
9  };
```

```
1  EnsembleEntiersFini::EnsembleEntiersFini(int t) {
2    taille_max = t;
3    taille = 0;
4    contenu = new int[taille_max];
5  }
6
7  EnsembleEntiersFini::~EnsembleEntiersFini() {
8    delete[] contenu;
9  }
```



```
1  bool EnsembleEntiersFini::contient_entier(int k) {
2    for(int l = 0; l < taille; l++)
3      if(k == contenu[l]) return true;
4    return false;
5  }
6
7  void EnsembleEntiersFini::ajouter_entier(int k) {
8    if(taille == taille_max) abort();
9    contenu[taille] = k;
10   taille++;
11 }
```


Exercice

Construire une classe qui hérite de `EnsembleEntiers` et qui permet de représenter un ensemble de zéro ou une valeur. Utiliser `abort` pour les opérations impossibles.

```
class EnsembleEntiers {
public:
  virtual bool contient_entier(int k) = 0;
  virtual void ajouter_entier(int k) = 0;
};
```


Correction

```
1  class Singleton : public EnsembleEntiers {
2    bool vide;
3    int valeur;
4  public:
5    Singleton() { vide = true; }
6    virtual bool contient_entier(int k) {
7      return !vide && valeur == k;
8    }
9    virtual void ajouter_entier(int k) {
10     if(vide) { vide = false; valeur = k; }
11     else abort();
12   }
13 };
```

Une liste vue comme un ensemble

Par un mécanisme d'héritage, en partant de:

```
1  class ListeEntiers {
2    Cellule *premiere;
3  public:
4    ListeEntiers();
5    ~ListeEntiers();
6    virtual void ajouter(int n);
7    virtual int taille();
8    virtual bool contient(int n);
9  };
```


Nous pouvons faire

```
1  class ListeCommeEnsemble : public ListeEntiers,
2                             public EnsembleEntiers {
3  public:
4    virtual bool contient_entier(int k) {
5      return contient(k);
6    }
7    virtual void ajouter_entier(int k) {
8      ajouter(k);
9    }
10 };
```


Itérateurs

Boucle sur des ensembles

Il faut souvent faire des boucles sur des ensembles sans savoir quelle est leur structures sous-jacentes (listes, tableaux, autres ?). On introduit le concept suivant:

```
1  class IterateurEntier {
2  public:
3    virtual bool pas_fini() = 0;
4    virtual int suivant() = 0;
5  };
```

La méthode `pas_fini()` renvoie `true` s'il reste des valeurs à parcourir et `suivant` renvoie une valeur et passe à la suivante donc **on récupérera une valeur différente la prochaine fois qu'on l'appelera**.

Exemple d'utilisation d'un itérateur (1)

On peut par exemple programmer une fonction pour sommer les valeurs que parcourt un itérateur:

```
1  int somme(IterateurEntier *it) {
2    int s = 0;
3    while(it->pas_fini()) s += it->suivant();
4    return s;
5  }
```


Exemple d'utilisation d'un itérateur (2)

ou pour trouver la plus grande valeur parmi des valeurs que parcourt un itérateur:

```
1  int maximum(IterateurEntier *it) {
2    if(it->pas_fini()) {
3      int m = it->suivant();
4      while(it->pas_fini()) {
5        int n = it->suivant();
6        if(n > m) m = n;
7      }
8      return m;
9    } else abort();
10 }
```


Exercice

Programmez une routine qui compte le nombre de valeurs négatives parmi des valeurs parcourues par un itérateur.

```
int nombre_valeurs_negatives(IterateurEntier *it);
```


Correction

```
int nombre_valeurs_negatives(IterateurEntier *it) {
  int n = 0;
  while(it->pas_fini()) if(it->suivant() < 0) n++;
  return n;
}
```