

Boucles définies

Juliusz Chroboczek

20 Septembre 2010

1 Boucles définies

Une *boucle* est une structure de contrôle qui sert à exécuter le même bloc de code de multiples fois. Une boucle définie, ou boucle *for*, est une boucle dont l'exécution est contrôlée par un *compteur de boucle* dont la valeur varie entre deux valeurs bien définies.

Syntaxe et sémantique des boucles définies Le fragment de code suivant exécute les deux instructions `printf` pour toutes les valeurs de `i` allant de 1 à 10 :

```
int i;
for(i = 1; i <= 10; i = i + 1) {
    printf("J'ai collé %d timbres.\n", i);
    printf("Il m'en reste %d.\n", 10 - i);
}
```

Une boucle `for` est composée du mot-clé `for` suivi de trois expressions entre parenthèses, séparées par des points-virgule, puis d'un bloc de code entre accolades :

$$\text{for}(e_1; e_2; e_3) \text{ bloc}$$

L'exécution de la boucle procède de la façon suivante :

1. l'expression e_1 , dite *initialisation*, est exécutée ;
2. le *test* e_2 est évalué ; s'il est faux, on sort de la boucle, et l'exécution de la boucle est terminée ;
3. le corps de la boucle est exécuté ;
4. l'expression e_3 est exécutée ;
5. on recommence à l'étape 2.

Pas arbitraires L'instruction e_3 d'une boucle définie sert typiquement à incrémenter le compteur de boucle, c'est-à-dire à ajouter une constante à sa valeur. Cette constante n'est pas forcément égale à 1.

Par exemple, le fragment de code suivant :

```
int i;
for(i = 0; i < 10; i = i + 1)
    printf("%d est pair.\n", 2 * i);
```

peut aussi s'écrire :

```
int i;
for(i = 0; i < 20; i = i + 2)
    printf("%d est pair.\n", i);
```

Lorsque le pas est négatif, il faudra penser à inverser le sens du test :

```
int i;
for(i = 9; i >= 0; i = i - 1)
    printf("Il me reste %d timbres.\n", i);
```

2 Quelques abréviations

Utilisée toute seule,

- l'expression `i++` est équivalente à `i = i + 1`;
- l'expression `i--` est équivalente à `i = i - 1`;
- l'expression `i += c` est équivalente à `i = i + c`;
- l'expression `i -= c` est équivalente à `i = i - c`.

Ces abréviations sont souvent utilisées dans l'expression e_3 d'une boucle définie.

3 Quelques types

Nous avons déjà vu les types `int` (entiers signés codés sur 4 octets) et `double` (nombres à virgule flottante codés sur 8 octets). Le C permet l'utilisation de plusieurs autres types¹.

Nombres entiers Une variable de type `int` contient un nombre entier, signé, codé sur 4 octets. Le spécificateur de format `printf` et `scanf` correspondant à `int` est « `%d` ».

Nombres à virgule flottante Une variable de type `double` contient un nombre à virgule flottante, codé sur 8 octets. Le spécificateur de format correspondant est « `%lf` » (affichage usuel), `%le` (affichage en notation scientifique) ou `%lg` (choisit automatiquement l'affichage).

Du fait des phénomènes d'arrondi, l'utilisation de l'opérateur d'égalité « `==` » avec des opérandes en virgule flottante n'est presque jamais une bonne idée.

Caractères Le type `char` est celui des caractères, codés sur 1 octet. Le spécificateur de format correspondant est `%c`.

1. Il s'agit là de types prédéfinis. Lors du cours de programmation C, vous apprendrez aussi à définir vos propres types.

Type des constantes Tout comme les variables, les constantes sont typées en C. Une constante entière, par exemple « 2 », a le type `int`. Pour écrire une constante de type `double`, il faut inclure un point décimal (la version anglophone de la virgule décimale), par exemple « 2.0 ».

Surcharge des opérateurs Les opérateurs arithmétiques, « + », « - », etc., sont *surchargés* : leur opération dépend du type de leurs opérandes. Lorsqu'ils sont appliqués à des opérandes de types différents, l'opérande du type le plus petit est automatiquement converti dans le type du plus grand.

La surcharge est particulièrement notable dans le cas de l'opérateur « / », qui effectue une division entière (euclidienne) sur les entiers, et une division en virgule flottante sur les valeurs à virgule flottante.

Quel est le type et la valeur des expressions suivantes ?

1. $5 / 2$;
2. $5.0 / 2.0$;
3. $5 / 2.0$;
4. $(1 / 3) * 3$;
5. $(1.0 / 3.0) * 3.0$.