Chapitre 1-5
Opérations

1. Hérité du C

1.1 Notion d'expression

Dans le code informatique, tout élément (ou ensemble d'éléments) qui fait l'objet d'une évaluation numérique est appelé une expression :

`a + b`, `a / b`, `a = b`, `&b`, `!b` sont des expressions.

Dans ces expressions +, /, =, `&` et `!` sont des opérateurs et les variables `a` et `b` sont les opérandes.

Ces combinaisons de variables avec des opérateurs sont appelées des expressions plutôt que des opérations, car il n'y a pas que des opérateurs arithmétiques. De plus, une valeur constante, une variable ou un appel de fonction seuls sont également considérés comme des expressions. Ce sont des expressions élémentaires. Les expressions qui font appel à des opérateurs et plusieurs arguments sont dites expressions composées.

La valeur numérique d'une expression, composée ou élémentaire, est le résultat de l'expression elle-même. S'il s'agit d'une opération arithmétique ou d'une affectation, la valeur numérique de l'expression est le résultat de cette opération. S'il s'agit d'un appel de fonction, c'est la valeur de retour de la fonction. L'expression vaut cette valeur, elle est cette valeur, et à ce titre une expression peut être intégrée dans une expression plus large, dans une affectation ou encore comme opérande dans une opération.

1.2 Opérations arithmétiques

Les opérateurs arithmétiques sont les suivants :

+ (plus) : le résultat de l'expression `a+b` est la somme.

– (moins) : le résultat de l'expression `a-b` est la soustraction.

* (multiplier) : le résultat de l'expression `a*b` est la multiplication.

/ (diviser) : le résultat de l'expression `a/b` est la division.

% (modulo) : le résultat de l'expression `a%b` est le reste de la division de a par b.

Affectations combinées

Toutes les opérations (y compris les opérations bit à bit présentées plus loin) peuvent se combiner avec une affectation. Par exemple :

```
int i = 10;
i = i +20;
i = i * 2;
i = i / 3;
i = i % 5;
```

peut se remplacer par :

```
i += 20;
i *= 2;
i /= 3;
i %= 5;
```

1.3 Valeurs aléatoires

La fonction C `rand()` permet d'obtenir très facilement une valeur pseudo-aléatoire. Elle retourne un nombre dans la fourchette maximum de 32768 définie dans cstdlib (stdlib.h) par la macroconstante `RAND_MAX`.

Exemple d'appel :

```
#include <iostream>


int main()
{
    int val;

    val = rand();
    std ::cout << val << '\n';


    std ::cin.get();
    return 0;
}
```

Si vous relancez le programme, vous constatez que vous obtenez toujours le même résultat qui est 41.

En effet, il n'y a pas d'aléatoire véritable dans un ordinateur, mais uniquement du pseudo-aléatoire. Il est obtenu par un calcul donnant un résultat quasi impossible à prédire. Lors du premier appel de la fonction `rand()`, ce calcul est opéré sur un premier nombre (la valeur 1).

Lors du second appel, le même calcul est opéré sur le nombre obtenu précédemment et ainsi de suite. Le pseudo-aléatoire est ainsi une suite de nombres pour ainsi dire impossible à prédire, obtenue à partir d'un premier nombre. Si le premier nombre est toujours le même, la suite aussi.

1.3.1 Avoir des suites différentes

Pour avoir des suites différentes, il faut spécifier un premier nombre différent. C'est le rôle de la fonction `srand()`. Par exemple :

```
srand(5);
```

spécifie la valeur 5 comme premier nombre. La suite sera différente, mais le problème reste car, à chaque lancement, nous aurons la même suite.

Pour obtenir des suites toujours différentes, la solution consiste à prendre l'heure pour premier nombre. Cette initialisation sur l'heure s'opère une seule fois, de préférence au tout début du programme. La fonction `time()` retourne généralement le temps en secondes écoulées depuis le 1er janvier 1970 à minuit (mais, selon les environnements, elle peut fonctionner différemment). Cette fonction se trouve dans la bibliothèque time.h :

```
#include <iostream>
#include <ctime>

int main()
{
    srand(time(NULL));



    int val = rand();
    std::cout << val << '\n';

    std::cin.get();
    return 0;
}
```

1.3.2 Définir une fourchette

Une fourchette maximum se définit en ajoutant un modulo sur la valeur de retour, le minimum s'obtient en ajoutant la valeur minimum. Par exemple, pour un nombre aléatoire entre 50 et 100, nous écrivons :

```
val = 50 + rand()%50
```

L'obtention de nombres aléatoires entre 0 et 1 se fait en divisant le retour de `rand` par `RAND_MAX` :

```
float val = rand()/RAND_MAX; // attention
```

Mais cette division donne quasiment toujours 0, parce que l'opération est faite avec des valeurs entières (pas de virgule). Elle nécessite l'opérateur de cast présenté un peu plus loin pour donner une valeur en réel.

1.4 Opérations bit à bit

En C, nous avons six opérateurs qui permettent des opérations au niveau des bits. Ils s'utilisent uniquement avec les types entiers signés ou non : `char`, `short`, `int`, `long`.

1.4.1 ET : opérateur &

L'opérateur ET fonctionne bit à bit de la façon suivante :

```
i & 0  donne toujours 0
i & 1  donne i, c'est-à-dire soit 0 soit 1
```

Par exemple, le résultat de 12 & 10 donne 8 :

```
         1 1 0 0     (12 en binaire)
    &    1 0 1 0     (10 en binaire)
donne    1 0 0 0     (8 en binaire)
```

Il sert pour accéder à un ou plusieurs bits particuliers sur un entier. Pour ce faire, nous prenons une valeur de masque dans laquelle seuls les bits à 1 sont visibles. Par exemple :

Pour connaître la valeur du premier bit d'un entier, masque à 1 :

```
          10111001
    &     00000001
donne     00000001
```

Pour connaître la valeur du second bit, masque à 2 :

```
          10111001
    &     00000010
donne     00000000
```

Pour connaître la valeur du troisième bit, masque à 4 :

```
          10111001
    &     00000100
donne     00000000
```

Pour connaître la valeur du quatrième bit, masque à 8 :

```
          10111001
    &     00001000
donne     00001000
```

Pour connaître la valeur résultante des quatre premiers bits de l'octet :

```
          10111001
    &     00001111
donne     00001001
```

Pour connaître la valeur résultante des quatre derniers bits de l'octet :

```
          10111001
    &     11110000
donne     10110000
```

L'intérêt est de pouvoir disposer sur un seul entier de plusieurs interrupteurs booléens distincts ou de pouvoir le décomposer en plusieurs plages de bits.

1.4.2 OU exclusif : opérateur ^

Cet opérateur met à 1 ce qui diffère et à 0 ce qui est identique :

```
0^0    donne 0
1^0    donne 1
0^1    donne 1
1^1    donne 0
```

Il est utilisé dans différentes occasions. Par exemple, dans le programme suivant, les valeurs des deux variables sont échangées :

```
#include <iostream>

int main()
{
    int a = 345, b = 678;

    std::cout << "a: " << a << "b: " << b << '\n';
    a ^= b;
    b ^= a;
    a ^= b;
    std::cout << "a: " << a << "b: " << b << '\n';

    std::cin.get();
    return 0;
}
```

1.4.3 OU inclusif : opérateur |

Met à 1 ce qui est à 1 :

```
0 | 0        donne 0
1 | 0        donne 1
0 | 1        donne 1
1 | 1        donne 1
```