Ce cours présente la notion de type structuré que vous serez amené à utiliser dans la suite des exercices. Nous ne proposons pas encore d'exercice de découverte lié.

Simples enregistrements

Lorsque l'on a beaucoup de données, il peut s'avérer intéressant de les regrouper sous un seul nom. Imaginons par exemple que l'on ait de nombreux dessins de rectangle à gérer ; on souhaiterait alors pouvoir regrouper ensemble leur nombre de lignes et de colonnes, ainsi que le caractère de remplissage.

Il faut alors définir un type : une *structure* ou *enregistrement*. Nous définissons ci-dessous un type *DessinRectangle*. Ce code peut être placé n'importe où : dans une fonction, dans une autre structure, ou à l'extérieur de tout bloc. En général, nous serons dans ce dernier cas, afin de pouvoir utiliser notre structure dans tout notre programme, sans le moindre souci.

```
struct DessinRectangle
{
   int nbLig, nbCol;
   char motif;
};
```

Maintenant, on peut définir des variables (des « instances ») de ce type. Nous créons ci-dessous trois rectangles :

```
DessinRectangle rectLong = { 3, 12, '>' };
DessinRectangle rectHaut = { 8, 2, 'I' };
DessinRectangle rectGros = { 10, 15, '0' };
```

Comme pour les tableaux, les accolades peuvent être utilisées pour indiquer les valeurs des attributs d'une variable de type *DessinRectangle*.

Dans la suite, on utilise le point `.` pour accéder aux attributs d'une variable de type *DessinRectangle*. Les attributs fonctionnent comme des variables normales : on peut lire leur valeur ou la modifier.

Voici un exemple d'utilisation :

```
void dessinerRectangle(DessinRectangle rect)
{
   for (int iLig = 1; iLig <= rect.nbLig; iLig = iLig + 1)
   {
      for (int iCol = 1; iCol <= rect.nbCol; iCol = iCol + 1)
         cout << rect.motif;
      cout << endl;
   }
}

dessinerRectangle(rectLong);
dessinerRectangle(rectHaut);
dessinerRectangle(rectGros);
```

↳

```
>>>>>>>>>>>>
>>>>>>>>>>>>
>>>>>>>>>>>>
II
II
II
II
```

```
II
II
II
II
000000000000000
000000000000000
000000000000000
000000000000000
000000000000000
000000000000000
000000000000000
000000000000000
000000000000000
000000000000000
```

Affectations entre enregistrements

Lorsque l'on affecte une instance à une autre :

```
DessinRectangle rectX = { 5, 4, 'x' };
DessinRectangle copie = rectX;
```

les valeurs sont copiées, et on peut alors traiter deux variables indépendantes avec des attributs.

En effet, en C++, lorsqu'une variable possède un type, elle est rattachée à un emplacement mémoire stockant une instance de ce type. Ainsi, lorsque vous prenez une structure en paramètre d'une fonction, la valeur donnée à l'appel de la fonction est copiée dans l'emplacement mémoire du paramètre. Le paramètre a ainsi son propre emplacement mémoire. Si la structure est très grosse, appeler la fonction va donc nécessiter un espace supplémentaire, et une copie qui peut prendre du temps. Pour éviter cela, on pourra donner à la fonction l'adresse de la structure à la place.

Dans un bon nombre de langages plus modernes, les variables ne sont pas rattachées à une instance, mais à une adresse ; en somme, elles sont comme des pointeurs. Une variable n'est donc pas lourde ; seule la valeur l'est, et celle-ci peut être la cible de plusieurs variables.

Méthodes

Certains langages permettent d'écrire des fonctions à l'intérieur des types structurés ; on appelle ces fonctions des *méthodes*. Une méthode est équivalente à une fonction, sauf qu'au lieu de faire `fonction(instance, ...)`, elle permet d'écrire `instance.fonction(...)`, donc de faire ressortir l'élément dans l'écriture. Dans le corps de la méthode, l'instance est accessible par un mot-clé particulier.

Voici par exemple la structure précédente, avec une méthode pour le dessin :

```
struct DessinRectangle
{
   int nbLig, nbCol;
   char motif;

   void dessinerRectangle()
   {
      for (int iLig = 1; iLig <= nbLig; iLig = iLig + 1)
      {
         for (int iCol = 1; iCol <= nbCol; iCol = iCol + 1)
            cout << motif;
         cout << endl;
```

```
        }
    }
};
```

Avec cette définition, on peut appeler la méthode en écrivant `rectangle.dessinerRectangle()`. Dans la méthode, l'adresse de l'instance appelante est accessible via le mot-clé `this`. On peut ainsi accéder au motif de l'instance en écrivant `this.motif`. Toutefois, les attributs et les méthodes sont aussi accessibles directement dans la méthode : on peut donc écrire tout simplement `motif`.

Classes et autres notions d'orienté objet

C++ implante le paradigme orienté objet, c'est-à-dire qu'il permet les choses suivantes :

- écrire des classes plutôt que des structures, et définir que certains attributs et méthodes de la classe ne sont pas accessibles à l'extérieur de la classe ;
- indiquer qu'une structure hérite des attributs et méthodes d'une autre, ce qui permet d'utiliser une instance de différents types de la même manière.

Rendre des attributs inaccessibles ne vous sera jamais utile dans la résolution de problèmes comme sur notre plateforme ; nous n'abordons donc pas ce point ici. De même, l'héritage est une notion complexe qui ne rentre pas dans le cadre de notre enseignement.