Module ITC313 - Informatique

Partie C / C++

TD5 Creation d'objets, Genericite parametrique, STL

Benoît Darties - benoit.darties@u-bourgogne.fr
Université de Bourgogne

Année universitaire 2016-2017

---

1 Creation d'objets en C++

Exercice 1 : *Définition d'une premier classe, et interactions entre objets*

1. Définissez une classe `pokemon` comprenant les éléments suivants :
   — deux attributs privés `pointsVie` et `pointsAttaque` de type entier
   — les accesseurs en lecture et écriture correspondants
   — Un constructeur permettant d'initialiser `pointsVie` et `pointsAttaque` avec des valeurs passées en paramètre.
2. Créez ensuite une fonction `combat()`. Cette fonction prend en paramètres deux pokémon et simule un combat de pokémons de manière simpliste : à chaque tour de boucle, un pokémon tire un nombre aléatoire entre 1 et ses points d'attaque. Le pokemon avec la valeur la plus élevée "attaque" l'adversaire. Cette attaque ôte 1 unité aux points de vie de l'autre pokémon. Le combat continue jusqu'a ce que les points de vie d'un pokémon atteignent 0. Un message indique alors si le premier ou le second pokemon est vainqueur.

Créez enfin une fonction `main()` et testez votre fonction `combat()` sur plusieurs pokémons.

2 Une classe C++ interessante : la classe `string`

Dans cette section, nous étudierons les caractéristiques du type objet ***`string`*** *de la librairie standard* ***`C++`****. Cet objet permet de gérer beaucoup plus facilement des mots de plusieurs caractères, tandis qu'en* `c`*, il est nécessaire de stocker un mot dans un tableau de caractères, et de les comparer avec des fonctions spéciales..*

Exercice 2 : *Saisie de lignes*

Dans cet exercice, nous manipulons des premiers mots avec les objets de type ***`string`****, et mettons en avant la simplicité d'utilisation de cette classe.*

Nous allons créer facilement deux variables de type`string` et montrer que la manipulation de ces variables est très simple.

1. Recopiez ou récupérez le programme `concatMot.cpp` suivant :

```
#include <iostream>
#include <string>

using namespace std;

int main (void) {
        string s1, s2, s3;

        cout << "Tapez une première chaine :";
        cin >> s1;

        cout << "Tapez une seconde chaine :";
        cin >> s2;

        cout << "la premiere chaine tapee est : " << s1 << endl;
        cout << "la seconde chaine tapee est : " << s2 << endl;

        s3 = s1 + s2;

        cout << "l'union de ces chaines est : " << s3 << endl;

        cout << "chaines  identiques? "<<  (s1 == s2) << endl;
}
```

2. Compilez et testez plusieurs fois ce programmes en entrant successivement deux mots sans espaces, tantôt différents, tantôt identiques pour vérifier la condition du test `(s1 == s2)`. Bien que le code utilisé soit très simple, gardez en tête que ces opérations auraient été bien plus compliquées à mettre en place si on avait utilisé les chaînes de caractères spécifiques au C.
3. Que se passe-t'il si la première saisie contient un espace ? et la seconde ?
4. Modifiez votre code et effectuez les deux saisies de `s1` et `s2` en une seule fois au moyen de l'instruction :
   `cin >> s1 >> s2;`
   Que se passe-t'il si vous rentrez 2 mots séparés par un espace ? Si vous rentrez 1 seul mot ? Si vous rentrez 3 mots séparés par un espace ?
5. Pour prendre en compte la saisie d'espace, on utilisera la fonction `getline()` dont le prototype est :
   `getline ( istream &, string &) ;`
   Le premier paramètre est le flux d'entrée sur lequel on veut lire, et le second l'objet `string` dans lequel copier le résultat saisi. Modifiez votre programme pour que deux phrases soient successivement saisies et affectées dans 2 variables $s1$ et $s2$, et que leur contaténation soit affectée dans une troisième $s3$. Afficher chacun de ces objets à l'écran.

Exercice 3 : *Quelques methodes de la classe* ***`string`***

Dans cet exercice, nous poursuivons l'étude de la classe ***`string`****, en étudiant quelques méthodes proposées dans cette classe.*

Nous allons maintenant utiliser deux méthodes de la classe `string`, permettant de récupérer la taille d'un mot, et de récupérer un caractère situé à une position précise.

1. créez une fonction main() demandant à saisir 2 phrases, et rajoutez les instructions suivantes :
   ```
   cout « "total s1 : " « s1.size() « endl;
   cout « "total s2 : " « s2.size() « endl;
   ```
   Exécutez votre programme. A quoi sert alors la méthode `size()` de la classe string ?

Remarques / questions : benoit.darties@u-bourgogne.fr

2. Rajoutez les instructions suivantes :

```
cout « "s1.at(3) = " « s1.at(3) « endl;
cout « "s1.at(4) = " « s1.at(4) « endl;
cout « "s1.at(6) = " « s1.at(6) « endl;
```

et exécutez votre programme en saisissant une chaîne d'au moins 7 caractères. A quoi sert la méthode `at(int)` de la classe `string` ? Que se passe-t'il si l'on tente d'afficher un caractère d'indice supérieur à la taille de la chaîne ?

Exercice 4 : *conversion de type*

Afin d'assurer une pleine compatibilité avec la gestion des chaînes de caractères en C, la classe string possède ses propres outils de conversion.

En langage `C`, les mots sont stockés dans des chaines de caractères, c'est à dire des tableaux de caractères. On peut créer un tableau de caractères `char tabCaract []` sans préciser sa taille, à condition que l'on l'initialise dès sa définition. Dans la suite, nous regardons comment convertir un mot en un format orienté `C` vers un format orienté `C++` et réciproquement.

1. Recopiez ou récupérez, puis analysez le code du fichier `convert.cpp` :

```
#include <iostream>
#include <string>

using namespace std;

int main() {
        /* conversion char -> string */
        char tabCaract [] = "hello bidule";
        string s1 = tabCaract;
        cout << "conv char -> string :"<< tabCaract << endl;

        /* conversion string -> char */
        string s2;
        char *c2;
        s2 = "hello machin";
        c2 = s2.c_str();
        cout << "conv string -> char :"<< c2 << endl;
}
```

2. A quoi sert alors la méthode `c_str()` de la classe `string` ? Quel type retourne-t'elle ?
3. Pour obtenir un correspondant à une chaîne de caractères donnée, nous avions vu que nous utilisions des fonctions telles que `atoi()`, `atol()`, `atof()`, ... Lorsque l'on souhaite obtenir la valeur numérique correspondant à un objet string donné, ces fonctions peuvent être utilisées à condition de convertir dans un premier temps l'objet string en char * (comme précédemment). Mais la librairie `string` offre également un autre outil dédié au `C++` très puissant sous la forme d'une classe nommée `istringstream`. Cette classe définit un flux d'entrée de caractères . Il suffit alors de lire un entier à partir de ce flot, tout comme on l'aurait fait avec le flux d'entrée standard `cin`. L'instruction :

```
istringstream istr (s);
```

permet de définir un flux de caractères à partir d'un objet string nommé s. Pour lire un entier sur ce flux , on utilise simplement l'instruction :

```
istr >> i ;
```

où i est une variable de type entier ou flottant. Tant que le flux comporte des éléments, les appels successifs à cette instruction affectent i à la valeur actuelle lue dans le flot. Si l'élément

Remarques / questions : benoit.darties@u-bourgogne.fr

lu par le flux ne peut pas être vu comme un entier, la valeur de i n'est pas affectée, et le résultat du test sur (istr » i) renvoie 0.

A partir de ces éléments, créez un programme `sommeIntLus` qui lit sur l'entrée standard une ligne contenant une liste d'entiers séparés par des espaces, et initialise un flux de caractères à partir de la chaîne lue. A partir de ce flux, le programme lit un entier, l'ajoute à un total, et boucle tant que le flux n'est pas vide. Une fois le flux vidé, il affiche le résultat calculé à l'écran.

3 Généricité paramétrique et redirection d'opérateurs

Les exercices de cette question abordent deux notions essentielles en C++, que sont la généricité paramétrique, et la redirection d'opérateur. La généricité paramétrique permet de compacter du code en le rendant indépendant des types des variables utilisées : plutôt que d'écrire plusieurs fois une fonction en l'adaptant pour différents types de variables, on crée un modèle de fonctions, et on le décline en autant d'exemplaires que l'on souhaite. La redirection d'opérateurs permet de rajouter des fonctionnalités aux opérateurs courants (`+, -, << , ...`) afin de pouvoir les utiliser sur des classes

Exercice 5 : *Genericite parametrique sur fonctions*

Dans cet exercice, nous définissons une fonction de recherche dichotomique sur les éléments d'un tableau ordonné. Ce tableau sera de type entier. L'objectif est d'utiliser la généricité paramétrique afin de proposer un modèle de fonction, de sorte à pouvoir dériver la fonction sur des tableaux contenant autre chose que des entiers

1. Proposez une fonction `rechercheDico()` dont les trois paramètres sont respectivement un tableau `T` d'éléments de type `int`, une taille de tableau, et une valeur entière à chercher. Cette fonction effectue une recherche dichotomique de la valeur entrée en paramètre dans le tableau, et renvoie 1 si cette dernière a été trouvée, et 0 autrement.
2. En utilisant la généricité paramétrique, modifiez votre fonction de sorte que cette dernière puisse désormais rechercher une valeur dans un tableaux dont le type d'élément est un paramètre formel.
3. Ecrivez une fonction `main()` qui initialise deux tableaux dont les éléments sont de type différents
4. Reprenez les questions précédentes avec la mise en place des fonctions suivantes :
   (a) une fonction `affcheTab()` qui affiche les éléments d'un tableau
   (b) une fonction `ordoTab()` qui ordonne les éléments d'un tableau (méthode de tri de votre choix)

Exercice 6 : *Redirection d'opérateur sur classe*

L'objectif de cet exercice est de comprendre les mécanismes de redirection des opérateurs, c'est à dire la possibilité d'utiliser des opérateurs connus sur autre chose que des types primitifs

Dans un premier temps, nous définissons une classe `Date`, puis nous redéfinissons les opérateurs de comparaison, afin de pouvoir comparer des dates. Une fois des dates comparables, on peut les stocker dans des tableaux et ordonner les éléments de ces tableaux avec les fonctions de tri précédemment définies

1. Définition de la classe Date :
   (a) Ecrivez une classe `Date` dont les éléments sont les suivants :

Remarques / questions : benoit.darties@u-bourgogne.fr

i. trois attributs `jour`, `mois` et `annee` dont vous définirez le type,

ii. un constructeur initialisant ces trois attributs,

iii. une méthode `nombreJours()` qui renvoie le nombre de jours séparant la date définie par les attributs d'une autre date passée en paramètres.

iv. une méthode `estPlusAncien()` qui renvoie vrai si et seulement si la date représentée par les attributs de l'objet est plus ancienne que la date passée en paramètres.

2. Redirection d'opérateurs, puis ordonnancement

   (a) En ré-utilisant la classes précédemment définie, redirigez les opérateurs `<`, `>`, `<=`, `>=` et `=` de sorte à pouvoir comparer deux dates en utilisant ces derniers (on se basera sur le retour des fonctions `nombreJours()` et `estPlusAncien()`.

   (b) Redéfinissez l'opérateur `<<` afin de pouvoir afficher facilement un objet de type `date` avec l'instruction `cout`.

   (c) Reprenez enfin les fonctions définies dans l'exercice *Genericite parametrique sur fonctions*, et modifiez éventuellement votre code en créant une nouvelle instance de ces fonctions pour des objets de type `Date`. Vérifiez que les fonctions définies marchent bien avec des tableaux dont les valeurs sont des objets de type `Date`.

4 Généricité paramétrique sur classe

Les exercices de cette section vous permettent de mettre en pratique les mécanismes de bases sur la généricité paramétrique. L'objet de ces exercices consiste à mettre en place des structures permettant de gérer des quantités (listes, piles, ...) d'éléments, pour plusieurs types possibles.

Exercice 7 : *Mise en place d'une liste chainee*

Avant d'appliquer la généricité paramétrique sur une liste, nous créons les fonctions nécessaires à la gestion d'une liste d'entiers. Puis nous les adapterons pour qu'elles puissent gérer des éléments dont le type sera formel et défini a postériori.

1. **Définition d'une liste chainée contenant des entiers :**

   (a) Reprenez le code que vous avez , définissez les différentes fonctions nécessaires à la gestion de listes chaînées contenant des entiers dans deux fichiers `ListeChaineInt.cpp` et `ListeChaineInt.h` :

      i. Une structure de type `struct noeud` permettra de définir un maillon de chaîne. Elle contiendra une valeur `val` de type int et un pointeur vers le maillon suivant nommé `next` de type `struct noeud *`.

      ii. la fonction `afficherListe()` prend en paramètre un pointeur de tête de liste (type `struct noeud *`.), et affiche la liste des éléments séparés par un point-virgule. Elle termine l'affichage avec un retour chariot ;

      iii. la fonction `ajouterElement()` permet d'ajouter un élément dont la valeur `val` de type `int` est passée en paramètre en tête d'une liste désignée par un pointeur de tête de liste également donné en paramètre. Faites attention aux points soulevés par les questions suivantes :
      — dans quelle zone mémoire le noeud doit-il être créé ?
      — La **valeur** même du pointeur est-elle modifiée ?
      — Que faire si la liste est vide au départ ? (pointeur de tête de liste égal à NULL) ;

      iv. La fonction `supprimerElement()` supprime le premier noeud rencontré dont la valeur est égale à une valeur `val` passée en paramètre (libère la mémoire), sans briser la liste chaînée. Faites attention aux points soulevés par les questions suivantes :
      — Que faire si le noeud à supprimer est le premier de la liste ?

Remarques / questions : benoit.darties@u-bourgogne.fr

— Le pointeur de tête de liste doit-il être mis à jour ?

(b) Testez vos fonctions par des manipulations conséquentes dans la fonction `main()` :
— Création d'une liste vide
— Ajout et suppression de multiples éléments
— Mise en évidence des situations problématiques (suppression du seul noeud existant, du premier noeud, ...)

2. **Mise en oeuvre de la généricité paramétrique dans une liste chaînée**

(a) Copiez les fichiers `ListeChaineInt.cpp / .h` précédemment créés vers les fichiers `ListeChaine.cpp / .h`, et videz au besoin la `fonction main()` de toute instruction.

(b) En utilisant la généricité paramétrique, modifiez la définition de la structure `noeud` de sorte que le type de valeur stockée soit un type formel que vous nommerez `typeValNoeud`.

(c) A partir de là, les types `struct noeud` et `struct noeud *` n'existent plus, et sont remplacés par `struct noeud<typeValNoeud>` et `struct noeud<typeValNoeud> *`. De ce fait, chacune des fonctions qui utilise ces types doit être précédée de l'annonce du paramètre formel. Modifiez votre code en conséquence en faisant précéder chaque fonction de l'annonce des template utilisés. Adaptez au besoin votre code afin de réaliser correctement la généricité paramétrique.

(d) Dans la fonction `main()` créez deux listes indépendantes, la première composée d'entiers et la seconde composée de flottants. Vérifiez que vous pouvez utiliser chacune des fonctions précédemment créées tantôt avec l'une ou l'autre des listes.

(e) Pouvez-vous ajouter un noeud contenant une valeur flottant dans la liste chaînée composée d'entiers ? Ce lien est possible en utilisant le transtypage, mais les variables de type formel peuvent être mal interprétées.

Exercice 8 : *Mise en place d'une pile de lettres*

Cet exercice synthétise les connaissances vues sur les listes chainées et la généricité paramétrique, afin de créer une structure de type pile de lettres.D'abord nous créons une classe `Pile` *sur des entiers, puis nous créons une classe* `Lettre`*. Ensuite nous modifions* `Pile` *pour que les éléments empilés soient de type formels, et nous empilerons enfin des objets de type* `Lettre`

Pour cet exercice, nous réutiliserons une partie du code produit dans l'exercice précédent.

1. **Définition d'une classe Pile sur un type primitif :**

(a) Nous allons mettre en place une classe nommée `Pile`, dont l'objectif est d'empiler des valeurs entières. Définissez la classe `Pile` à partir des indications suivantes. L'implémentation de `Pile` sera effectuée dans deux fichiers séparés `Pile.h` (donné ci-après) et `Pile.cpp`, et doit respecter les indications suivantes :
— Les valeurs empilées seront stockées dans une liste chaînée, dont le pointeur de tête sera un attribut de la classe `Pile`.
— La classe ne possède qu'un constructeur sans paramètre, qui crée une pile d'entiers vide.
— La méthode `empiler()` doit permettre d'ajouter un élément en début de pile, c'est à dire en début de liste chaînée.
— La méthode `dépiler()` doit permettre d'extraire le premier élément de la pile en renvoyant sa valeur, et en mettant à jour le pointeur de début de liste.
— La méthode `estVide()` renseigne sur l'état de la pile.
— La méthode `afficherPile()` doit permettre l'affichage des différents éléments de la pile.
— La méthode `prochain()` affiche le prochain élément qui sera extrait de la pile.
Pour vous aider, voici le contenu du fichier `Pile.h` :

Remarques / questions : benoit.darties@u-bourgogne.fr

```
#ifndef _PILE
#define _PILE

struct noeud {
        int val;
        struct noeud *next;
};

class Pile {
        private :noeud * pileList;
        public :
                Pile();
                void empiler(int p_valeur);
                int depiler();
                bool estVide();
                void afficherPile();
                void prochain();
};
#endif
```

2. **Définition d'une classe `Lettre` :**
   (a) Dans deux fichiers `Lettre.cpp` et `Lettre.h`, définissez une classe `Lettre` comportant les éléments suivants :
   — un attribut `auteur` de type `string`,
   — un attribut `contenu` de type `string`,
   — un constructeur ayant deux paramètres, chacun initialisant les attributs précédents.
   (b) Vérifier que vous pouvez bien construire des objets de type `Lettre` dans une fonction `main()` située dans un autre fichier
3. **Mise en oeuvre de la généricité paramétrique sur la pile :**
   (a) modifiez l'ensemble de votre code de sorte à pouvoir créer des piles d'objet de type formel `T` au moyen de la généricité paramétrique. Pour cela, utiliser l'instruction `template<typename T>` à chaque fois que nécessaire et en remplacant le type `int` par le type formel `T` aux endroits adéquats.
4. **Définition et utilisation d'une pile de lettres :**
   (a) Avant de pouvoir utiliser les méthodes de la classe `Pile` sur des objets de type `Lettre`, il faut que le compilateur génère le code nécessaire des méthodes de `Pile` sur des objets de type `Lettre`. Pour cela, reprenez le fichier `Pile.cpp` et ajoutez-y :
   — en début de code : la description d'un objet de type `Lettre` avec l'instruction `#include <Lettre.h>`
   — en fin de code : l'instruction permettant de définir une classe template `Pile` sur des objets de type `Lettre`, avec l'instruction `template class Pile<Lettre>;`
   (b) Testez enfin votre classe templatée en créant une pile de lettres et en ajoutant et retirant des lettres au sein d'une fonction `main()`

Remarques / questions : benoit.darties@u-bourgogne.fr