Langages Orientés Objet : C++ moderne
TP N° 1 – Revoir les bases

Objectifs :
Ce premier TP vise d'un côté à vous rafraîchir la programmation du C++ moderne, abordée pendant le semestre précédent. Nous mettrons en pratique la théorie présentée dans le support de cours *Premiers concepts* qui est disponible sur Moodle → S5A → Langages Orientés Objet. Si malheureusement ce TP a lieu avant le cours, votre encadrant sera là pour vous aider. Les transparents et la page Moodle dédiée à OBJ fournissent aussi des liens vers d'autre documentation. N'hésitez pas non plus à chercher par vous même.
Ce TP contient aussi des points facultatifs (écrits entre **[ ]**) ou des exercices entiers facultatifs pour les programmeurs les plus intéressés et téméraires.

En particulier ce TP traitera :
- Fonction *main()*.
- Écriture dans la console.
- Programmation modulaire.
- Espace de noms (*namespace*).
- Utilisation des types élaborés (*string*, *array*, *vector*).
- Passage des arguments par valeur et par référence (constante ou pas).
- Retour de fonctions par valeur d'une ou plusieurs données.
- Arguments par défaut.
- Exceptions.
- Recopie et déplacement.
- Surcharge des opérateurs.
- *Template* (un très bref aperçu) et fonctions *inline*.

1. Fonction *main()* et écriture du *makefile*.
Commençons par le commencement. Créez un dossier de travail, nommé *s5loo*, dans lequel vous créerez un sous-dossier dédié à ce TP, nommé *TP1_basics*. Ouvrez n'importe quel éditeur de texte, *kate* par exemple, et écrivez la fonction principale dans sa version avec des paramètres (pour rappel, *argc* et *argv*).
Le *main()* devra simplement afficher son nom (utilisez la variable *__func__* implicitement définie) et ses paramètres.
Sauvegardez le fichier dans le dossier *TP1_basics* en l'appelant *prog.cpp*.

Pour compiler, vous devez utiliser le fichier *GNUmakefile* fourni sur Moodle dans la section « IMPORTANT – fonctionnement des Labos ». Le *GNUmakefile* permet de compiler n'importe quel ensemble de fichiers *.c* ou *.cpp* sous plusieurs plate-formes (Linux, Windows et Mac). Cela ne signifie pas que vous avez le droit d'oublier comment écrire un *makefile* !
Pour rappel, ici : https://www.enib.fr/~harrouet/dev_tools/ vous trouverez les informations nécessaires pour créer l'environnement de travail sur votre machine personnelle.

2. Programmation modulaire, *array*, passage des arguments et espace de noms.
Ajoutez maintenant un fichier d'en-tête *moduleArray.hpp*. Rappelez vous que tout fichier d'en-tête doit être encadré par les balises *#ifndef/#define/#endif*. Dans ce fichier vous allez déclarer des fonctionnalités pour manipuler 15 *string*s contenues dans un *array*. Pour vous simplifier la vie renommez le type *array* en utilisant le mot clef *using* comme indiqué dans le support de cours ; utilisez par exemple le nom *str15*. Toute fonctionnalité devra être définie dans le fichier *moduleArray.cpp*.

La première fonctionnalité s'appelle *fillArray()*, elle ne renvoie rien et reçoit l'*array* à remplir. Pour commencer affectez simplement à l'*array* les valeurs suivantes :
{"hi", "hello", "anna", "kayak", "21:12", "goodbye"}.
Les 9 *string*s restantes seront automatiquement initialisées à "".
Attention, la modification de l'*array* reçu en paramètre doit être visible aussi dans le contexte appelant. Pour

cela un **passage par référence** s'impose.

La deuxième fonctionnalité s'appelle *printArray()*, elle doit afficher à l'écran les éléments contenus dans l'*array* entre accolades et séparés par une espace. Remarquez que l'*array* est juste lu mais jamais modifié, dans ce cas il faudra faire un **passage par référence constante**.

La troisième fonctionnalité s'appelle *isPalindrome()*, elle reçoit une *string* et doit renvoyer *true* si le mot contenu dans le paramètre est un palindrome, *false* autrement. Pour rappel, un mot est un palindrome si l'on peut le lire indifféremment de gauche à droite ou de droite à gauche. Une *string* vide est un palindrome. Étant donné que les *string*s que nous manipulons ici sont très courtes et que nous ne devons pas les modifier pour le contexte appelant, un simple **passage par valeur** sera suffisant.

Pour compléter proprement le module composé par les fichiers *moduleArray.hpp* et *moduleArray.cpp*, englobez le nouveau nom du type *array* (*str15*) et les fonctionnalités dans l'espace de noms (*namespace*) *s5loo*.

Testez ces fonctionnalités. Pour cela, incluez le fichier *moduleArray.hpp* dans *prog.cpp* auquel ajoutez une fonction *test_array()*. Cette fonction ne retourne rien et ne prend rien en paramètre. Elle déclare une variable de type *array* de *string*s de 15 éléments, l'initialise en appelant la fonction *fillArray()*, l'affiche grâce à la fonction *printArray()* et enfin, pour chaque élément du tableau affiche s'il est un palindrome ou pas. Rappelez vous d'utiliser le nom simplifié que vous avez donné au type *array* (*str15*) et du fait que tout ce qui a été déclaré dans le fichier *moduleArray.hpp* appartient au *namespace s5loo*.

Compilez et testez votre application pour vous assurer que tout fonctionne comme prévu.

(facultatif) [Modifiez la fonction *fillArray()* pour générer le contenu de l'*array* automatiquement. Le tableau doit être entièrement rempli par des mots de dimension variable (à générer aléatoirement entre 3 et 6 caractères) composés par des lettres 'a', 'b', 'c' et 'e', toujours à tirer au hasard.
Vous pouvez utiliser les instructions suivantes pour créer *lengthDistrib* un générateur de valeurs entières pseudo-aléatoires entre 3 et 6 et *charDistrib*, un générateur de caractères pseudo-aléatoires entre *a* et *e* :

```
std::default_random_engine rndGen{std::random_device{}()};
std::uniform_int_distribution<int> lengthDistrib{3, 6};
std::uniform_int_distribution<int> charDistrib{'a', 'e'};
```

Maintenant pour obtenir une valeur entière aléatoire il faut simplement écrire :

```
int x = lengthDistrib(rndGen);
```

et pour obtenir un caractère aléatoire :

```
char y = charDistrib(rndGen);
```

Pour utiliser le générateur aléatoire il faut inclure le ficher d'en-tête standard <*random*>.
]

3. *Vector*, retour de fonctions par valeur d'une ou plusieurs données, arguments par défaut, exceptions.
Ajoutez un nouveau module composé par les fichiers *moduleVector.hpp* et *moduleVector.cpp*. Il nous permettra de déclarer et définir des fonctionnalités pour manipuler un *vector* d'entiers.
Il faut suivre la même démarche appliquée dans la conception du module précédent et commencer par renommer le type *vector* d'entiers avec un nom plus simple. Nous utiliserons *intvec*. Puis déclarez et définissez les fonctions suivantes :

makeVector() : elle reçoit trois entiers en paramètres, *count*, *nMin* et *nMax* et retourne par valeur un *vector* d'entiers qu'elle doit remplir avec *count* entiers aléatoires tirés dans l'intervalle [*nMin*, *nMax*]. La fonction doit pouvoir être appelée sans spécifier une valeur pour *nMin* et *nMax* ; dans ce cas ces paramètres doivent prendre les valeurs par défaut 10 et 100 respectivement. Remarquez que cette fonction aurait pu recevoir en paramètre le *vector* à remplir mais que nous avons fait un choix différent : le *vector* est créé dans la fonction et **renvoyé par valeur**. Cette opération en effet ne coûte rien, vu que le compilateur optimise le code en créant le *vector* a renvoyer directement dans le contexte appelant.

printVector() : elle doit afficher les éléments contenus dans le *vector* d'entiers passé en paramètre entre

accolades et séparés par une espace. Remarquez que le *vector* ne doit pas être modifié, mais seulement lu.

(facultatif) [*numEven()* : elle reçoit un *vector* d'entiers en paramètre et retourne le nombre de valeurs paires contenues dans la collection. Remarquez que le *vector* ne doit pas être modifié, mais seulement lu.**]**

dataVector() : elle reçoit un *vector* d'entiers pour en calculer la moyenne, l'écart type et la médiane, lorsque la taille du *vector* est différent de zéro (si la taille vaut zéro vous pouvez lever une exception).
Utilisez les formules suivantes.

Moyenne :

$$\overline{x} = \frac{1}{n}\sum_{i=0}^{n-1} x_i$$

Écart type :

$$\sigma = \sqrt{\frac{1}{n}\sum_{i=0}^{n-1} (x_i - \overline{x})^2}$$

où *n* correspond au nombre d'éléments dans le *vector* et x_i correspond à l'i-ème élément du *vector*.
Pour trouver la **médiane** il faut trier le *vector* et prendre la valeur du milieu si la taille du *vector* est impaire. Si la taille est paire il faut prendre la moyenne de deux valeurs au centre du *vector*. Pour trier le *vector* utilisez la fonction *std::sort()* (http://en.cppreference.com/w/cpp/algorithm/sort). Attention, le *vector* d'origine ne doit pas être modifié !
La fonction doit **renvoyer** ces trois valeurs **à travers un *tuple*** (voir exemples sur le support de cours). Remarquez que sur le support de cours, pour créer un *tuple*, on suggère d'utiliser la fonction s*td::move()*. La sémantique *move* a été largement expliquée en S4, n'hésitez pas à reprendre la documentation du semestre précédent si vous ne vous en souvenez plus.

Comme pour le module *moduleArray*, toutes les fonctionnalités doivent appartenir au *namespace s5loo*.

Testez ces fonctionnalités. Pour cela, incluez le fichier *moduleVector.hpp* dans *prog.cpp* auquel ajoutez une fonction *test_vector()*. Cette fonction ne retourne rien et prend en paramètre une chaîne de caractères (*const char* arg*). Premièrement, elle doit essayer de lire un entier dans le paramètre *arg* (utilisez *std::stoi()* pour ça), si cela n'est pas possible une exception est levée par la fonction *std::stoi()*. Renseignez alors l'utilisateur que la fonction *test_vector()* doit recevoir un entier et interrompre l'application en appelant *std::exit(0)*. Revoir sur le support de cours l'utilisation d'un bloc *try{}catch{}* si vous ne vous souvenez plus.
Ensuite, déclarez un *vector* d'entiers et initialisez-le avec la valeur renvoyée par la fonction *makeVector()*. Imprimez le *vector* à l'aide de la fonction *printVector()*.

(facultatif) [Affichez le nombre de valeurs paires dans le *vector* que vous aurez obtenu en appelant la fonction *numEven()*.**]**

Enfin, récupérez les valeurs calculées par la fonction *dataVector()* et affichez-les à l'écran. Le support de cours vous donne des exemples sur comment accéder aux valeurs contenues dans un *tuple*.

La fonction *test_vector()* doit maintenant être appelée par le *main()*. Les moins audacieux entre vous peuvent juste passer la chaîne de caractères "25" à la fonction *test_vector()*.
(facultatif) [Les autres modifieront le code afin que l'utilisateur puisse choisir quel test lancer (*array* ou *vector*) en l'indiquant dans la ligne de commande. Cela veut dire que l'application pourra être lancée comme ceci :

```
./prog array
```

ou comme cela :

```
./prog vector 24
```

Dans le premier cas, le test sur l'*array* de *string*s implémenté précédemment sera exécuté. Dans le deuxième cas, la fonction *test_vector()* sera appelée en lui passant le troisième paramètre de l'application (24 dans cet exemple). Contrôlez que l'application soit lancée avec les paramètres qu'il faut, si non un message qui

explique comment lancer l'application doit être affiché à l'écran.**]**

Testez votre application et assurez-vous qu'elle marche comme prévu.

4. Surcharge des opérateurs (facultatif, seulement pour qui ne s'en souvient plus).
Reprenez l'exemple donné dans le support de cours pour la surcharge des opérateurs (structure *Vec3D*). Créez le fichier *Vec3D.hpp* et ajoutez aux opérateurs déjà surchargés les opérations ^ et * combinant deux Vec3D (comme dans le cas de l'addition) pour en calculer respectivement le produit vectoriel et le produit scalaire.
Testez en implémentant une fonction *test_operator()* dans le fichier *prog.cpp*. Cette fonction doit déclarer deux *Vec3D* (*v1{1,2,3}* et *v2{4,5,6}*) et en calculer la somme, la multiplication par un scalaire, le produit vectoriel, le produit scalaire et afficher les *Vec3D* résultants à l'écran.

Maintenant modifiez le *main()* pour ajouter l'appel à la fonction *test_operator()*. Les plus audacieux entre vous qui ont fait aussi le troisième point facultatif de la troisième partie de ce TP, veilleront à que l'utilisateur puisse lancer l'application comme suit :
`./prog operator`
et que dans ce cas la fonction *test_operator()* soit appelée.

~~~~~~~~~~~~~~~~~~~~ **Conclusion : ce qu'il faut retenir de cette expérience** ~~~~~~~~~~~~~~~~~~~~
Ce premier TP vous a permis revoir les concepts de base du C++ moderne.
Dans les TPs qui suivront, il faudra systématiquement appliquer la démarche de la programmation modulaire et toujours englober les fonctionnalités dans un espace de noms.

Nous nous imposerons de ne jamais plus utiliser les chaînes de caractères, les tableaux et l'allocation dynamique "à la C", mais nous nous servirons de *string* (pour les chaînes de caractères), *array* (pour les tableaux de dimension connue à la compilation) et de *vector* (pour les tableaux dynamiques). Le C++ moderne rend l'utilisation de ces types élaborés aussi simple que celle des type de base, grâce au fait que l'opération de recopie reste cohérente indifféremment du type sur lequel elle est appliquée. Comme vous le constaterez, l'utilisation systématique de ces types élaborés nous permettra de limiter au maximum l'allocation dynamique, qui est souvent cause d'erreurs et de fuites de mémoire même chez les programmeurs les plus expérimentés.

Lors de l'écriture d'une fonction, pour le passage des paramètres nous suivrons toujours les règles suivantes :
- lorsque la donnée doit être uniquement consultée :
  - passage par valeur, pour les types de base et les données de petite taille (telles que les structures avec quelques champs de type de base, les *array* contenant peu d'éléments et les *string*s courtes),
  - par référence constante, pour tout le reste,
- lorsque la donnée doit être modifiée dans le contexte appelant :
  - par référence,
- on n'utilisera le passage par pointeur que pour les données qui pourraient valoir *nullptr*, ce qui est assez rare.

Et pour le retour d'une fonction nous utiliserons toujours :
- retour par valeur,
  - si plusieurs données doivent être retournées, nous nous servirons d'un *tuple.*

Si nous voulons donner une valeur par défaut aux paramètres d'une fonction, cela se fera uniquement dans sa déclaration et toujours pour les paramètres le plus à droite de la liste.

Nous avons revu aussi la sémantique *move*. Cette opération nous permet de déplacer les données sans avoir à les dupliquer (ce qui peut s'avérer très coûteux). Elle sera explicitement utilisée à chaque fois où le dépouillement de la donnée d'origine est envisageable, c'est à dire lorsqu'on sait quelle ne sera plus utilisée. L'opération de déplacement est faite implicitement sur les données temporaires, telles que les valeurs
~~~~~~~~~~~~~~~~~~~~

renvoyées par une fonction, par exemple.
Prendre l'habitude d'utiliser la fonction *std::move()* à bon escient ne nous coûte rien et le compilateur sera en mesure d'optimiser le code pour nous fabriquer des applications plus performantes (moins de mémoire occupée et moins d'instruction à exécuter pour dupliquer les données).

Nous avons revu aussi la surcharge des opérateurs.