

TP 10 – Références & pointeurs

Exercice 0 Mise en place du TP

- 📁 Récupérez sur l'ENT le fichier *VEtudiant.zip*. Il contient une solution Visual Studio à compléter au fur et à mesure de ce TP.

Exercice 1 Passage par référence vs passage par valeur

- 📁 Ce premier exercice a pour objectif de valider l'éventuel gain en ressources entre un passage de paramètre par référence et un passage par valeur.

1.1 Prise en main du code fourni

Le code fourni permet de réaliser des benchmarks de fonctions. C'est-à-dire qu'il permet d'analyser le temps d'exécution des fonctions. Le principe pour mesurer le plus précisément le temps d'exécution d'une fonction est d'appeler en boucle cette fonction durant un certain temps (ici, nous avons choisi 6 secondes) et de compter le nombre de fois qu'elle a pu s'exécuter. Une simple division permet ensuite de déterminer le temps moyen d'exécution de la fonction.

- 📁 Dans le code fourni, activez le projet *Exercice1* comme projet de démarrage.

Ce code s'architecture dans la fonction principale **main()** dans le fichier *Exercice1.cpp*. Cette fonction se décompose en 6 parties clairement identifiées dans le code par des commentaires :

1. **Données de test** : il s'agit de la déclaration de deux variables **texte** et **d** qui seront utilisées pour être passées aux fonctions.
2. **Initialisation du benchmark** : il s'agit ici de deux tâches. La première permet de « faire chauffer » le processeur afin que le système se mette à 100% de ses capacités de traitement, certains processeurs ayant la capacité de ralentir lorsqu'il n'y a pas de traitement lourd à réaliser. La seconde mesure le temps passé par le processeur pour gérer une boucle de mesure. N'est donc pris en compte ici que le temps passé à tester le critère d'arrêt de la boucle et faire l'incréméntation du compteur.
3. **Quatre sections d'analyse** pour analyser 4 fonctions dans la suite de cet exercice. Ces quatre sections sont construites sur le même principe et sont pour le moment commentées.

- 📁 Compilez et exécutez ce programme.

- ➡ Le programme va passer 6s à faire chauffer le processeur et 6s à analyser le temps de gestion d'une boucle d'analyse.

1.2 Passage d'une chaîne de caractères

1.2.1 Passage par valeur

- 📁 Dans les fichiers *fonctions.h* et *fonctions.cpp*, créez une fonction **passStringByValue()** prenant en paramètre une chaîne de caractères de type **std::string** par valeur. Cette fonction ne fera rien et ne retournera rien.

caractère. La fonction retournera un tableau d'indices des caractères de début de sous-chaine.

- ➡ Chaque élément du tableau de retour sera l'indice de début de chaque sous-chaine.
- ➡ Comment **DEVEZ**-vous passer la chaîne de caractères **str** ?
- ➡ L'illustration ci-dessous explique ce qui est attendu :

str :	B	O	N	J	O	U	R		L	E	S		G	E	N	S		!
Tableau de retour :	0								8				12				17	18

- 📖 Dans le programme principal, testez cette fonction en lui passant la variable **strTexte** préexistante, et le caractère *espace* en délimiteur.
 - ➡ Affichez chaque sous-chaine sur des lignes séparées afin de vérifier vos résultats.
- 📖 Décommentez et complétez le code restant dans la fonction **main()** afin de mettre en place le benchmark de ces fonctions.
 - ➡ Que conclure ?
- 📖 Passer le mode de compilation de *Debug* à *Release* et relancez le benchmark.
 - ➡ Que constatez-vous ?

Exercice 3 Paramètres optionnels

- 📖 Dans le code fourni, activez le projet *Exercice3* comme projet de démarrage.

3.1 Apprentissage

Les pointeurs sont souvent utilisés comme paramètres optionnels aux fonctions. C'est-à-dire qu'une fonction peut fonctionner avec ou sans un paramètre particulier. Pour cela, le paramètre est passé par un pointeur qui est **nullptr** si le paramètre ne doit pas être utilisé par la fonction.

- 📖 Afin de tester ce style de programmation, créez une fonction **afficherInt()** prenant en paramètre un pointeur vers un entier.
 - ➡ Si le pointeur est différent de **nullptr**, la fonction affichera la valeur pointée par le paramètre
 - ➡ Sinon, la fonction affichera « *Non défini* »
- 📖 Testez cette fonction dans le programme principal avec les deux cas de figure.

En C++, il est possible de spécifier des paramètres par défaut. C'est-à-dire que si le paramètre n'est pas passé lors de l'appel de la fonction, il prend automatiquement une valeur prédéfinie. Pour cela, il suffit d'ajouter « = **valeur_par_défaut** » à la suite de la déclaration du paramètre dans la déclaration de la fonction. Attention, cela n'est possible que pour les derniers paramètres d'une fonction. Voici quelques exemples de ce qui est possible et ce qui ne l'est pas :

```
//OK, pas de paramètre par défaut :
void fonction1(int a, int b, int c);
//fonction1 peut être appelée par :
fonction1(4, 5, 6);

//OK, c a la valeur 3 par défaut :
void fonction2(int a, int b, int c = 3);
//fonction2 peut être appelée par :
fonction2(4, 5, 6);
fonction2(4, 5); //Ici, c = 3

//OK, b et c ont des valeurs par défaut :
void fonction3(int a, int b = 2, int c = 3);
//fonction3 peut être appelée par :
fonction3(4, 5, 6);
fonction3(4, 5); //Ici, c = 3
```

- Utilisez cette technique pour que le pointeur passé en paramètre à la fonction **afficherInt()** ait la valeur **nullptr** par défaut.
- Testez cette fonction dans le programme principal en appelant la fonction **afficherInt()** sans paramètre.

3.2 Cas un peu plus utile

Dans cet exercice, vous allez afficher du texte stylé dans la console. Grâce à des séquences de contrôle, il est possible de paramétrer différents aspects de la console, dont les couleurs d’affichage. Ces séquences de contrôle répondent à la norme VT-100, dont une référence succincte est disponible ici : <https://www2.ccs.neu.edu/research/gpc/VonaUtils/vona/terminal/vtansi.htm>. Par exemple, pour commencer à afficher le texte en rouge, il suffit de sortir dans la console la séquence de caractères « `\x1B[1;31m` » (« `\x1B`’ est le code ASCII de caractère correspondant à la touche « *escape* »). Pour redéfinir la couleur par défaut, il faut alors utiliser la séquence « `\x1B[0m` ».

- Testez cette fonctionnalité de la console en affichant le texte « **Bouh !** » en rouge. Assurez-vous que la console reprend un affichage standard à la fin de votre programme.

Nous allons utiliser cette fonctionnalité pour afficher du texte stylé.

- Créez une fonction **afficherTexteStyle()** prenant en paramètres un tableau **tabTexte** de chaînes de caractères et un pointeur **pStyles** vers un tableau de chaîne de caractères.
 - La fonction affichera chaque chaîne de caractères en appliquant un style défini dans le tableau pointé par **pStyles**, seulement si ce pointeur est différent de **nullptr**.
 - Chaque chaîne de caractères sera constituée avec un premier caractère qui sera l’indice du style.
 - L’indice référencera un élément dans le tableau pointé par **pStyles**. Chaque élément dans **pStyle** sera une séquence de contrôle VT-100.
 - Enfin, la fonction doit rétablir le style standard de la console à la fin de son travail.
- Testez cette fonction avec un programme capable d’afficher la sortie suivante (le programme n’a le droit d’appeler votre fonction que 2 fois) :

```
Sortie non stylée :
Différents types de messages :
Un message qui fait peur.
Un message qui te prévient.
Un message plutôt sympa.
C'est pas trop cool ça ?



Sortie stylée :
Différents types de messages :
Un message qui fait peur.
Un message qui te prévient.
Un message plutôt sympa.
C'est pas trop cool ça ?
```

Exercice 4 Exploration de la mémoire

 Dans le code fourni, activez le projet *Exercice4* comme projet de démarrage.



Chaque donnée, quel que soit son type se retrouve stockée sous forme d'octets dans la mémoire. Il est donc nécessaire de savoir comment passer d'une donnée typée (un **double**, un **int**, ...) vers sa représentation binaire sous forme d'octets. C'est notamment nécessaire pour la lecture / écriture des données binaires dans les fichiers via les fonctions **read** et **write** des objets **std::fstream**. L'objectif de cet exercice est d'apprendre à passer d'une représentation binaire brute à une représentation typée d'une même donnée.

4.1 Cas simple, un entier

-  Dans le programme principal, commencez par créer un entier égal à 42, par exemple.
-  En utilisant les manipulateurs de flux **std::hex**, **std::setfill** et **std::setw** (de la bibliothèque *iomanip*), affichez cette valeur sous forme hexadécimale sur un nombre de chiffres correspondant à la taille mémoire du type d'entier choisi.

Il est possible d'observer une adresse mémoire en l'interprétant par un autre type que la donnée qui y est stockée. Pour cela, on utilise le mot clé **reinterpret_cast<>()** en spécifiant entre les signes < et > le type de pointeur qui réinterprètera l'adresse passée entre parenthèses. Voici un exemple :

```
double donnee = 42.0;
unsigned char* pRaw = reinterpret_cast<unsigned char*>(&donnee);
//pRaw pointe vers la variable donnee, mais vue comme un "tableau"
//d'unsigned char. Le tableau a la "taille" d'un double en
//mémoire
```

-  En utilisant ces nouvelles connaissances, affichez tous les octets sous forme hexadécimale constituant votre variable entière précédemment créée.
-  Qu'observez-vous ?

Selon les architectures d'ordinateur, l'ordre des octets n'est pas toujours le même. Cet ordre est appelé le **boutisme** ou **endianness** en anglais. On distingue principalement 2 types d'architecture :

les architectures *gros-boutistes* et *petit-boutistes*. En gros-boutisme, les octets de poids fort (les plus significatifs dans la valeur numérique) sont placés au début du stockage, en petit-boutisme, c'est l'inverse. Quel est le boutisme de votre ordinateur ?

Le boutisme pose des problèmes lorsqu'il n'est pas pris en compte et que des données binaires sont transférées entre différents ordinateurs d'architectures différentes. Il est donc nécessaire de choisir un boutisme pour le stockage des données, de connaître le boutisme de l'architecture sur laquelle le programme s'exécute et enfin de savoir inverser le boutisme s'il ne correspond pas.



4.2 Connaître le boutisme de l'architecture

Il est possible de connaître le boutisme de l'architecture courante en utilisant le test suivant, après inclusion de la bibliothèque `<bit>` :

```
if constexpr (std::endian::native == std::endian::big)
    std::cout << "gros-boutiste\n";
else if constexpr (std::endian::native == std::endian::little)
    std::cout << "petit-boutiste\n";
else std::cout << "mi-boutiste\n";
```





Il y a plein de choses à dire sur ce petit extrait de code. Premièrement, il ne compile que depuis la norme C++20 du C++, car la bibliothèque `<bit>` n'existait pas avant. Deuxièmement, le `constexpr` situé à côté du `if` signifie que cette condition peut être évaluée à la compilation (libérant alors le processeur de réaliser ce test à l'exécution du programme). C'est ici possible car la comparaison porte sur des constantes intégrées dans la bibliothèque `<bit>`. Enfin, le cas « *mi-boutiste* » est un cas suffisamment rare pour pouvoir être ignoré.

Comment faire si vous n'avez pas accès à la bibliothèque `<bit>` ? Il suffit de prendre une constante numérique entière positive inférieure à 256 (par exemple 42), de la réinterpréter sa mémoire sous forme d'octets bruts, puis de comparer la valeur du premier octet avec la valeur de la constante initiale. Si le premier octet a la même valeur que la constante initiale, alors l'architecture est petit-boutiste, sinon elle est gros-boutiste.

-  Sans utiliser la bibliothèque `<bit>`, créez une fonction `estPetitBoutiste()` retournant `true` si l'architecture est petit-boutiste.
-  Testez cette fonction dans votre programme principal.

4.3 Enregistrement de données binaires en forçant le boutisme

Nous allons mettre en place l'enregistrement de données au format binaire de façon portable sur toutes les architectures.

-  Créez une fonction `rawPrint()` prenant un paramètre de type `double` et affichant sa représentation binaire brute en mémoire.
 -  Affichez les octets sur deux caractères hexadécimaux séparés par un espace.
 -  Testez la fonction dans le programme principal.
-  Créez une fonction `reverseEndianness()` prenant un paramètre de type `double` en entrée / sortie et qui retournera sa représentation binaire brute en mémoire. Ainsi un petit-boutiste deviendra gros, et inversement.

- ➡ Vous pouvez vous-même implémenter cette mise à l'envers ou bien utiliser la fonction `std::reverse`.
- ➡ Testez la fonction dans le programme principal en vous aidant de la fonction `rawPrint()`.
- 📁 Créez une fonction `saveToFile()` prenant en paramètre un nom de fichier et une valeur de type **double**. Cette fonction enregistrera le double au format binaire gros-boutiste dans le fichier spécifié. C'est cette fonction qui doit se charger de retourner les octets de la donnée si l'architecture est petit-boutiste.
 - ➡ L'enregistrement binaire se fait en ouvrant un flux `std::ofstream` en mode binaire et en écrivant les données à l'aide de la fonction `write()`. Cette fonction prend en premier paramètre un pointeur de type **const char*** à la base des données à enregistrer et un second paramètre entier spécifiant la taille en octets de la donnée à écrire.
 - ➡ Vérifiez l'enregistrement en ouvrant le fichier dans un éditeur hexadécimal. Par exemple, si votre fichier porte une extension inconnue comme « *.bin* », Visual Studio l'ouvrira dans un éditeur binaire. Vérifiez que la donnée est bien dans l'ordre gros-boutiste.
- 📁 Créez une fonction `readFromFile()` effectuant le travail inverse.
 - ➡ Cette fois-ci, la fonction s'appelle `read()` et prend les mêmes types de paramètres.
 - ➡ Vérifiez le fonctionnement de la lecture en relisant la donnée précédemment écrite. Vérifiez que la donnée est correctement lue.

Exercice 5 Pointeurs vers fonction – générateurs de parler creux









- 📁 Dans le code fourni, activez le projet *Exercice5* comme projet de démarrage.

Nous allons produire dans cet exercice 3 générateurs de phrases vides de sens, mais qui paraissent à première vue très intelligentes. Elles sont structurées autour d'un sujet, d'un verbe transitif et d'un complément d'objet direct constitué d'un nom, d'un adjectif et d'un complément du nom. Chacune de ces 5 parties de la phrase est tirée aléatoirement parmi une liste de possibilités.

Les trois générateurs concerneront trois thématiques différentes : le management, la pédagogie et la programmation informatique. Dans le code fourni, vous trouverez dans le fichier *fonctions.cpp* 3 lots de 5 tableaux de chaînes de caractères correspondant aux 3 thèmes. Ces tableaux vous permettront de générer les phrases en prenant successivement un groupe de mots dans chaque tableau. Une fonction `aleatEntreBornes()` vous est également fournie.

Un programme principal vous est proposé avec la gestion d'un menu, mais il est pour le moment commenté.

- 📁 Commencez par créer une fonction `generateurManagement()` prenant en paramètre un booléen **style** et retournant une chaîne de caractères.
 - ➡ La chaîne retournée sera une concaténation de 5 groupes de mots choisis aléatoirement dans les 5 tableaux correspondant au thème du management.
 - ➡ N'oubliez pas d'ajouter des espaces entre les groupes de mots et un point à la fin de la phrase.
 - ➡ Pour le moment, le paramètre **style** peut être ignoré.
- 📁 Testez cette fonction dans le programme principal en l'appelant 5 fois (ne décommentez pas le code pour le moment, faites votre propre programme principal).
 - ➡ Affichez à chaque fois le résultat.

-  Recommencez ces deux dernières étapes pour les 2 autres thèmes.
-  Dans le programme principal, supprimez tous vos tests et décommentez le code fourni.
-  Créez un tableau de pointeurs vers fonction du type des générateurs. Stockez-y des pointeurs vers les 3 fonctions de génération en respectant l'ordre des thèmes : management, pédagogie, programmation.
-  À l'endroit adéquat du **switch** principal, appelez la fonction correspondante à l'indice **indice** et affichez son résultat.
 -  Testez votre programme, il doit être fonctionnel à l'exception de la gestion du style.
-  Modifiez les fonctions de génération afin de prendre en compte le paramètre **style**. Lorsque le **style** est activé, la fonction doit permettre d'afficher en couleurs différentes les 5 groupes de mots. Utilisez pour cela des séquences VT-100.
-  Dans le programme principal, dans le cas du **switch** correspondant, basculez la valeur du booléen **style**. N'oubliez pas de passer ce booléen à vos fonctions de génération.
 -  Testez votre programme.