IMAC
INGÉNIEUR
ESIPE

Travaux dirigés C++ n°11

Informatique

—IMAC 2e année—

C++ légèrement avancé

Le but de ce TD est de se familiariser avec les concepts légèrement avancés du C++.

▸ Exercice 1. *Expressions constantes*

Objectif de l'exercice : calculs durant la compilation

1. *Faire une fonction calculant la valeur absolue d'un nombre. Votre fonction pourra prendre le prototype suivant :*

```
template<typename T>
T my_abs(const T x);
```

2. *Si vous utilisez des constantes comme* 0 *ou* 1 *ou* π*, il est souvent nécessaire de les convertir (par une opération de cast) dans le type* `T`*, en remplaçant* `if( a > 0 )` *par* `if( a > static_cast<T>(0) )`*.*

3. *Le cas échéant, remplacer votre* `if` *de la façon suivante :*

```
if(condition) return a;
else return b;
```

devient :

```
return condition ? a : b;
```

4. *Avec le mot clé* `constexpr`*, certaines fonctions peuvent être calculées à la compilation. Pour cela, il faut que toutes les données nécessaires à leur déroulement soient connues à la compilation. Il faut également que votre fonction n'ait qu'un seul* `return`*. Ces fonctions ne contiendront dans leur corps uniquement des opérations* `constexpr` *(attention au* `if`*). Par exemple, la fonction qui calcule le signe d'un nombre peut prendre la forme suivante:*

```
template<typename T>
constexpr int sgn(const T x)
noexcept
{
  return x > static_cast<T>(0) ? 1 : (x < static_cast<T>(0) ? -1 : 0);
}
```

Pour vous assurer que la fonction a bien été évaluée à la compilation (et qu'elle donne le bon résultat, vous pouvez ajouter un test :

```
static_assert(sgn(-5) == -1, "test of 'sng' at compile time");
```

En vous inspirant de la fonction `sgn`*, écrire la fonction* `my_abs` *permettant de calculer la valeur absolue d'un nombre à la compilation en utilisant le mot clé* `constexpr`*.*

▸ **Exercice 2.** ***Expressions constantes et récursivité***

Objectif de l'exercice : calculs récursifs durant la compilation

1. *Faire une fonction récursive calculant* x^n *où* $x \in \mathbb{R}$ *et* $n \in \mathbb{N}$*. Votre fonction pourra prendre le prototype suivant :*

```
double pow1(const double &x, const unsigned int n);
```

2. *Réécrire cette fonction, nommée* `pow2`*, pour la rendre générique à l'aide de template.*

3. *En vous inspirant de la fonction factorielle présentée ci-dessous, écrire la fonction* `pow3` *permettant de calculer* x^n *à la compilation en utilisant le mot clé* `constexpr`*.*

```
#include <iostream>

constexpr unsigned int fact(const unsigned int n){
  return (n == 0) ? 1 : n * fact(n-1);
}

int main(){
  static_assert(fact(5) == 120, "test of 'fact' at compile time");
  std::cout << fact(5) <<  std::endl;
  return 0;
}
```

▸ **Exercice 3.** ***Variadics***

Objectif de l'exercice : gérer les arguments à nombres variables.

Une autre façon de gérer simultanément la fonction factorielle et la fonction puissance pour les cas les plus simples consiste à faire une fonction qui multiplie ensemble tous ses arguments. Cette fonction pourrait être appelée de cette façon :

- `std::cout << product<unsigned int>(1,2,3,4,5) << std::endl;`
- `std::cout << product<double>(5,5,5) << std::endl;`

1. *Chercher dans votre cours ou sur le net comment écrire une fonction variadic. Ecrire la fonction* `product` *qui multiplie entre eux tous ses arguments.*
2. *Rendez vos fonctions* `constexpr` *et vérifiez le calcul à la compilation avec un* `static_assert`*.*

▸ **Exercice 4.** ***Fonction*** λ

Objectif de l'exercice : gérer les fonctors.

Si le nombre d'éléments à multiplier entre eux devient grand, la méthode précédente devient inutilisable et il est préférable de stocker les éléments dans un `std::vector`*.*

1. *Pour commencer, nous créerons nos vecteurs de la façon suivante :*
   `std::vector<int> v = {1,2,3,4,5};`
   Ecrire le code permettant de multiplier entres eux tous les éléments d'un `std::vector<T>` *d'éléments de type* `T`*.*
2. *Réécrire cette fonction en utilisant l'instruction* `std::for_each` *associée à une fonction lambda.*
3. *Réécrire la même fonction à l'aide de* `std::accumulate` *associé à* `std::multiplies`*.*
4. *Pour les vecteurs de grande taille, il est préférable de générer le contenu automatiquement. Définir un* `std::vector` *de taille* $n = 500$*. A l'aide de la fonction* `std::generate` *et d'une fonction lambda, remplir ce vecteur tel que* `v[i] = i`*.*