

Introduction au langage C++

Plan du cours

- Apport du langage C++ par rapport au langage C
- Espace de noms
- Les références
- L'allocation dynamique
- La surcharge de fonctions
- Programmation Orientée Objet (POO)

Introduction

- Le langage C a été développé dans les années 70 par Ritchie puis dans le début des années 90, *Bjarne Stroustrup* a introduit de nouveaux concepts tels que l'orienté objet donnant ainsi naissance à un nouveau langage qui est le C++. Toutefois, bien que le C++ ait gardé un grand nombre de notions et de syntaxes du langage C, les deux langages sont deux langages différents.

Apports du C++ par rapport au C

Les notions rapportées par le C++ par rapport au langage C:

- ✓ Le concept d'orienté objets (classes, encapsulation, heritage, ...)
- ✓ les **références**, la vérification des types,
- ✓ Attributions de valeurs par défaut aux paramètres de fonctions,
- ✓ la **surcharge de fonctions** (des fonctions portant le même nom mais possédants un nombre de paramètres différent),
- ✓ la **surcharge des operateurs** (pour utiliser les operateurs avec les objets), les constantes typées.

Premier programme

```
#include <iostream>
using namespace std;
int main(int argc, char **argv)
{
    cout << "Donnez un entier : " << endl;
    cin >> n;
    for(int i = 0; i < n; i++)
        cout << "Hello world !" << endl;

    return 0;
}
```

- `#include <iostream>` : une directive de préprocesseur (même concept qu'en C), elle est utilisée pour inclure des bibliothèques et des fichiers. Ici `iostream` est la bibliothèque contenant les définitions des flux d'entrées et de sorties; Ces définitions sont incluses dans un espace de nommage appelé `std`.
- `using namespace std;`
 - Cette ligne permet d'utiliser l'espace de nom `std` (namespace: concept à définir plus tard).
- `cout << "Donnez un entier : " << endl;` permet d'afficher le message à l'écran.
- Le `endl`: marque la fin de la ligne et insère automatiquement un `\n`.
- `cin >> n;` permet de demander à l'utilisateur de saisir une Valeur pour `n`.

Namespace en C++

- Un espace de nom (namespace) est un espace désigné pour encapsuler des entités tel que des constantes, des fonctions, des variables, ... afin de lever une ambiguïté sur un terme (des termes) utilisé dans des contextes différents (ex: un meme nom donné pour deux fonctions différentes).
- Un espace de nom est matérialisé par un préfixe identifiant la signification d'un terme.

Namespace en C++

- Exemple:

```
#include <iostream>

using namespace std;

namespace spaceA
{
    int tab[5] = {1, 3, 4, 7, 9};
}

namespace spaceB
{
    int tab[5] = {10, 33, 45, 77, 99};
}

int main()
{
    cout << "tab[0] = " << spaceA::tab[0] << endl;

    cout << "tab[0] = " << spaceB::tab[0] << endl;
    return 0;
}
```



```

#include <iostream>
using namespace std;
namespace spaceA
{
    int tab[5] = {1, 3, 4, 7, 9};
    void affiche(int tab[5])
    {
        for(int i =0; i< 5; i++)
        {
            cout << tab[i]*i <<endl;
        }
    }
}
namespace spaceB
{
    int tab[5] = {10, 33, 45, 77, 99};

    void affiche(int tab[5])
    {
        for(int i =0; i< 5; i++)
        {
            cout << tab[i]+i <<endl;
        }
    }
}

int main()
{
    cout << "appel de de la fonction affiche de l'espace de nom spaceA" << endl;
    spaceA::affiche(spaceA::tab);
    cout << "appel de de la fonction affiche de l'espace de nom spaceB" << endl;
    spaceB::affiche(spaceB::tab);
    return 0;
}

```

Alias d'espace de noms

- Il est possible d'avoir des espaces de noms imbriqués (encapsuler plusieurs espaces de noms dans un seul espace de noms). L'utilisation d'espaces de noms imbriqués (surtout si on a plusieurs niveaux) dans le programme principale peut le rendre non lisible:

```
#include <iostream>
using namespace std;
namespace spaceA
{
    namespace fonctions
    {
        int tab[5] = {1, 3, 4, 7, 9};
        void multiplication(int tab[5])
        {
            for(int i =0; i< 5; i++)
            {
                cout << tab[i]*i <<endl;
            }
        }
    }
    namespace constantes
    {
        const int A =200;
        const int B = 150;
    }
}
int main()
{
    spaceA::fonctions::multiplication(spaceA::fonctions::tab);

    return 0;
}
```

Alias d'espace de noms

- Pour rendre le code plus lisible, on utilise des alias.
- Un alias permet de ramener la portée d'un concept à un seul nom.
- *Exemple*: dans le code suivant, on crée un alias *A1* qui va remplacer la notation *spaceA::fonctions*

```
#include <iostream>
using namespace std;
namespace spaceA
{
    namespace fonctions
    {
        int tab[5] = {1, 3, 4, 7, 9};
        void multiplication(int tab[5])
        {
            for(int i =0; i< 5; i++)
            {
                cout << tab[i]*i <<endl;
            }
        }
    }
    namespace constantes
    {
        const int A =200;
        const int B = 150;
    }
}

namespace A1 = spaceA::fonctions;
int main()
{
    A1::multiplication(A1::tab);
    return 0;
}
```

Alias d'espace de noms

- Il est aussi possible de créer un alias sous forme d'une fonction pour ramener la portée à un seul mot.
- Dans l'exemple suivant, nous avons créé une fonction *multip()* qui va remplacer l'appel à la fonction *multiplication()* de l'espace de noms *fonctions* défini dans l'espace de noms *spaceA*.

```
#include <iostream>
using namespace std;
namespace spaceA
{
    namespace fonctions
    {
        int tab[5] = {1, 3, 4, 7, 9};
        void multiplication(int tab[5])
        {
            for(int i =0; i< 5; i++)
            {
                cout << tab[i]*i <<endl;
            }
        }
    }
    namespace constantes
    {
        const int A =200;
        const int B = 150;
    }
}

void multip()
{
    namespace A1 = spaceA::fonctions;
    A1::multiplication(A1::tab);
}

int main()
{
    multip();
    return 0;
}
```

Directive *using*

- La directive *using* permet d'utiliser les entités d'un espace de nommage sans spécifier à chaque fois le nom de ce dernier.

Syntaxe:

```
using namespace nom_espace;
```

```
#include <iostream>
using namespace std;
namespace spaceA
{
    const int A =900;
    const char B[] = "bonjour";
}
using namespace spaceA;

int main()
{
    cout << "A = " << A << endl;
    return 0;
}
```

Directive *using*

- Il est possible d'étendre un espace de nommage après l'utilisation de la directive `using`.

```
#include <iostream>
using namespace std;
namespace spaceA
{
    const int A = 900;
    const char B[] = "bonjour";
}
using namespace spaceA;

namespace spaceA
{
    void affiche()
    {
        cout << B << endl;
    }
}

int main()
{
    cout << "A = " << A << endl;
    affiche();
    return 0;
}
```

Directive using

- L'utilisation de la directive using peut causer un conflit lorsque deux espace de nommages contiennent deux entités portant un meme nom.

```
#include <iostream>
using namespace std;
namespace spaceA
{
    const char B[] = "bonjour";
}

namespace spaceB
{
    const char B[] = "Bonsoir";
}
using namespace spaceA;
using namespace spaceB;

int main()
{
    cout << B << endl;

    return 0;
}
```

\\main.cpp	22	error: reference to 'B' is ambiguous
------------	----	--------------------------------------

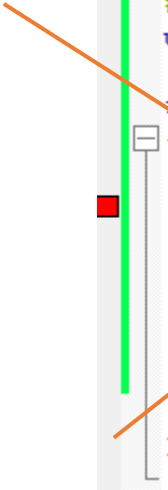
Les references en C++

- Une reference doit etre obligatoirement initialisée au moment de sa déclaration.
- Une reference ne peut pas etre initialisée qu'au moment de sa declaration alors elle ne peut pas etre utilisée comme référence à une autre variable.

```
#include <iostream>
using namespace std;

int main()
{
    int A = 10;
    int &B = A;

    return 0;
}
```

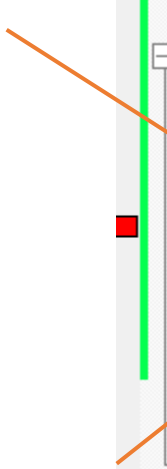


```
#include <iostream>
using namespace std;

int main()
{
    int A = 10;
    int &B;
    &B = A;

    return 0;
}
```

The diagram shows a vertical stack representing memory. A green bar indicates the stack's growth. A red square marks the position of variable B, which is declared before being assigned a value. An orange 'X' is drawn over the code, indicating it is invalid.



```
int main()
{
    int A = 10;
    int &B = A;

    int C = 100;
    &B = C;

    return 0;
}
```

The diagram shows a vertical stack representing memory. A green bar indicates the stack's growth. A red square marks the position of variable B, which is assigned the address of variable C. Since C has not yet been declared, this is invalid. An orange 'X' is drawn over the code.

Les references en C++

- Soit une variable A et une reference
&B vers A. L'affectation d'une nouvelle
Valeur à &B (B = 2) est l'équivalent de
A = 2.

```
#include <iostream>
using namespace std;

int main()
{
    int A = 10;
    int &B = A;

    B = 20;

    cout << "B = " << B << " A = " << A << endl;

    return 0;
}
```

Les references en C++

- L'utilisation de reference permet d'éviter de créer des copies pour une variable (principalement lors de passage de parametres à des fonctions)

```
#include <iostream>
using namespace std;

void affectation (char &x)
{
    char y = 'N';
    cout << "x = " << x << endl;

    x = y;
    cout << "x = " << x << endl;
}

int main()
{
    char var = 'B';
    char &r = var;
    affectation(r);
    cout << "var = " << var << endl;

    return 0;
}
```

Allocation dynamique de la mémoire

- L'allocation dynamique de la mémoire se fait en C++ en utilisant l'opérateur new.
- L'opérateur renvoie un pointeur vers le type voulu

```
#include <iostream>
using namespace std;

int main()
{
    int taille = 25;
    int *tab = new int (taille);

    for(int i = 0; i < taille; i++)
    {
        *(tab+i) = i;
        cout << "tab[" << i << "] = " << *(tab+i) << endl;
    }

    return 0;
}
```

Allocation dynamique de la mémoire

- La libération de la mémoire allouée par un pointeur peut être libérée en utilisant la fonction *delete*.

```
#include <iostream>
using namespace std;

int main()
{
    int *pt = new int (10);
    cout << "*pt = " <<*pt << endl;

    delete pt;

    int taille = 25;
    int *tab = new int (taille);
    for(int i = 0; i< taille; i++)
    {
        *(tab+i) = i;
        cout << "tab[" <<i <<"] = " << *(tab+i) << endl;
    }

    delete [] tab;

    return 0;
}
```

Surcharge de fonctions

- La surcharge de fonctions fait en sorte que plusieurs fonctions déclarées dans le même endroit portent exactement le même nom mais diffèrent dans le type des paramètres qu'elles acceptent.

```
#include <iostream>
using namespace std;

void afficher(int a)
{
    cout << "a = " << a << endl;
}

void afficher(int a, int b)
{
    cout << "a = " << a << "b = " << endl;
}

void afficher(int a, int b, int c)
{
    cout << "a = " << a << "b = " << "c = " << c << endl;
}

int main()
{
    int a = 2, b = 3, c = 10;
    afficher(a);
    afficher(a, b);
    afficher(a, b, c);
    return 0;
}
```