Cours_C++

Table des matières

Chapitre 1 - Découverte de la programmation C++

CHAPITRE 1 : Découverte de la programmation C++

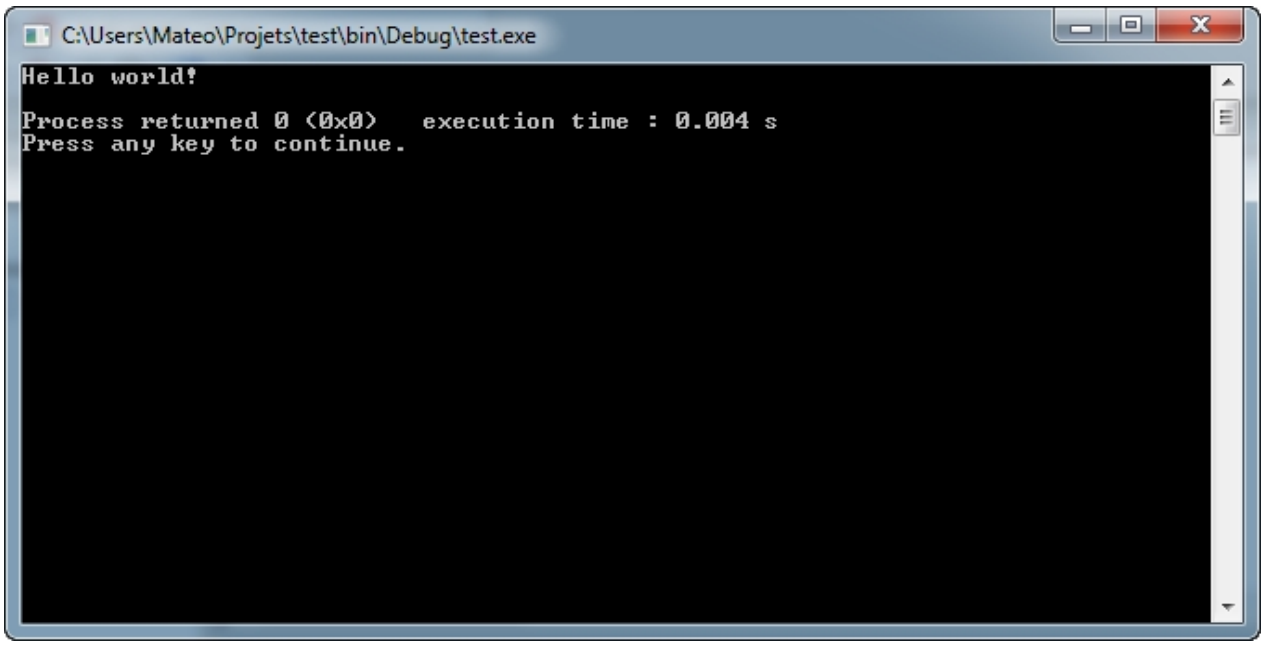
1 - "Hello World !"

Voici le programme de notre premier programme, un "Hello World !" :

```
#include <iostream>                          //Inclusion de la
bibliothèque d'affichage

using namespace std;                         //Espace de nom de la
bibliothèque standard
```

```
int main()                                          //Fonction principal
{
     cout << "Hello World!" << endl;                //Affichage du message
avec un saut de ligne
     return 0;                                      //Retour à 0 pour dire
que tous c'est bien passer
}
```

Ce programme doit nous retourner :



2 - Les variables

Les types de variables :

Type	Explication	Valeurs possible
`bool`	Booléan : 2 valeurs possible, `true` (vrai), `false` (faux).	`true` ou `false`
`char`	Un caractère.	`'a'`

`int`	Un nombre entier.	`-32768 à 32767`
`unsigned int`	Un nombre positif ou égal à 0.	`0 à 65535`
`double`	Un nombre à virgule	$-1.7*10^{-308}$ à $1.7*10^{308}$
`string`	Une chaîne de caractère	`"Hello World!"`
`enum`	Une énumération de valeurs	`Direction{ Nord, Sud}`

Déclaration de variables :

Il existe 2 façon de déclarer des variables :

- **TYPE NOM (VALEUR);**
- **TYPE NOM = VALEUR;**

Exemple :

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

int main()
{
    int ageUtilisateur(16);
    return 0;
}
```

Le cas des string :

Pour le cas des string, il faut en début de programme, inclure la bibliothèque des string en écrivant ceci : `#include <string>`.

Déclaration sans initialisation :

Il est aussi possible de déclarer une variable sans initialiser de valeur à l'intérieur comme ceci : **TYPE NOM;**

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

int main()
{
    string nomJoueur;
    int nombreJoueurs;
    bool aGagne;         //Le joueur a-t-il gagné ?

    return 0;
}
```

Attention !, il ne faut pas en cas de déclaration sans initialisation mettre des parenthèse !

Afficher la valeur d'une variable :

Pour afficher la valeur d'une variable, on utilise l'instruction `cout` comme dans l'exemple suivant :

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

int main()
{
    int qiUtilisateur(150);
    string nomUtilisateur("Albert Einstein");

    cout << "Vous vous appelez " << nomUtilisateur << " et votre QI vaut " << qiUtilisateur << endl;
    return 0;
}
```

Les références :

Les références permettent de créer un alias à une variable, ce qui permet à une variable d'avoir 2 nom, voici un exemple :

```cpp
int ageUtilisateur(16);  //Déclaration d'une variable.

int& maVariable(ageUtilisateur); //Déclaration d'une référence nommée maVariable qui est accrochée à la variable ageUtilisateur
```


3 - Interaction et arithmétiques

Demander des information à l'utilisateur :

Pour demander des informations à l'utilisateur il faut utiliser la commande `cin` et mettre le résultat de cette commande dans une variable du type de l'information demandé.

Exemple :

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

int main()
{
    cout << "Quel age avez-vous ?" << endl;

    int ageUtilisateur(0); //On prepare une case mémoire pour stocker un entier

    cin >> ageUtilisateur; //On fait entrer un nombre dans cette case

    cout << "Vous avez " << ageUtilisateur << " ans !" <<  endl; //Et on l'affiche

    return 0;
}
```

Attention !, Pour les string, il faut utiliser une méthode particulière pour assurer que le programme ne bug pas !

Exemple de demande d'information a l'utilisateur avec des string :

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

int main()
{
    cout << "Quel est votre nom ?" << endl;
    string nomUtilisateur("Sans nom"); //On crée une case mémoire pour contenir une chaine de caractères
    getline(cin, nomUtilisateur); //On remplit cette case avec toutela ligne que l'utilisateur a écrit

    cout << "Combien vaut pi ?" << endl;
    double piUtilisateur(-1.); //On crée une case mémoire pour stockerun nombre réel
    cin >> piUtilisateur; //Et on remplit cette case avec ce qu'écritl'utilisateur

    cout << "Vous vous appelez " << nomUtilisateur << " et vous pensez que pivaut " << piUtilisateur << "." << endl;

    return 0;
}
```

PS: Si vous vouliez demander d'abord des informations de type nombre et ensuite un string avec la fonction `getline()`, il faut ajouter après le demande des nombres l'instruction : `cin.ignore()`.

Opération arithmétiques :

Voici un tableau représentent toutes les opérations arithmétiques disponible :

Opération	Symboles	Exemple
Addition	+	`resultat = a + b`
Soustraction	-	`resultat = a - b`
Multiplication	*	`resultat = a * b`
Division	/	`resultat = a / b`
Modulo	%	`resultat = a % b`

Addition (raccourci)	`+=`	`resultat += a`
Soustraction (raccourci)	`-=`	`resultat -= a`
Multiplication (raccourci)	`*=`	`resultat *= a`
Division (raccourci)	`/=`	`resultat /= a`
Modulo (raccourci)	`%=`	`resultat %= a`
Incrémentation	`++`	`a++ ou ++a`
Décrémentation	`--`	`a-- ou --a`

Les constantes :

Les constantes sont des variables qui une fois déclarer ne peuvent pas changer de valeurs. Cela peut servir pour conserver des nombres qui ne changeront pas comme Π.

Exemple:

- `double const pi(3.14159);`
- `double const g(9.81);`

Encore plus de math avec `<cmath>` :

Nom de la fonction	Symbole	
Racine carré	$sqrtx$	
Sinus	$\sin(x)$	
Cosinus	$\cos(x)$	
Tangente	$\tan(x)$	
Exponentielle	e^x	
Logarithme népérien	$\ln x$	
Logarithme en base 10	$\log_{10} x$	
Valeur absolue	$\|x\|$	
Arrondi vers le bas	$\lfloor \times \rfloor$	
Arrondi vers le haut	$\lceil \times \rceil$	
Puissance	^	

4 - Les structures de contrôle

Les conditions

Condition `if, else` :

Voici ci après l'exemple d'une condition complète :

```cpp
#include <iostream>

using namespace std;

int main()
{
    int nbEnfants(2);

    if (nbEnfants == 0)
    {
        cout << "Eh bien alors, vous n'avez pas d'enfant ?" << endl;
    }
    else if (nbEnfants == 1)
    {
        cout << "Alors, c'est pour quand le deuxieme ?" << endl;
    }
    else if (nbEnfants == 2)
    {
        cout << "Quels beaux enfants vous avez la !" << endl;
    }
    else
    {
        cout << "Bon, il faut arreter de faire des gosses maintenant !" << endl;
    }

    cout << "Fin du programme" << endl;
    return 0;
}
```

Condition `switch` :

```cpp
#include <iostream>

using namespace std;

int main()
{
    int nbEnfants(2);

    switch (nbEnfants)
    {
        case 0:
            cout << "Eh bien alors, vous n'avez pas d'enfant ?" << endl;
            break;

        case 1:
            cout << "Alors, c'est pour quand le deuxieme ?" << endl;
            break;

        case 2:
            cout << "Quels beaux enfants vous avez la !" << endl;
            break;

        default:
            cout << "Bon, il faut arreter de faire des gosses maintenant !" << endl;
            break;
    }

    return 0;
}
```

Les boucles :

Les boucles permettent de répéter une action suivant si une condition est vrai ou fausse.

Boucle `while` :

```cpp
int main()
{
    int nbEnfants(-1); // Nombre négatif pour pouvoir entrer dans la boucle

    while (nbEnfants < 0)
    {
        cout << "Combien d'enfants avez-vous ?" << endl;
        cin >> nbEnfants;
    }

    cout << "Merci d'avoir indique un nombre d'enfants correct. Vous en avez " << nbEnfants << endl;

    return 0;
}
```

Boucle `do...while` **:**

```cpp
int main()
{
    int nbEnfants(0);

    do
    {
        cout << "Combien d'enfants avez-vous ?" << endl;
        cin >> nbEnfants;
    } while (nbEnfants < 0);

    cout << "Merci d'avoir indique un nombre d'enfants correct. Vous en avez " << nbEnfants << endl;

    return 0;
}
```

Boucle `for` **:**

```cpp
int main()
{
    int compteur(0);

    for (compteur = 0 ; compteur < 10 ; compteur++)
    {
        cout << compteur << endl;
    }

    return 0;
}
```

5 - Découper son programme en fonctions

Les types de fonction :

Il y a plusieurs type de fonction suivant si la fonction renvoie une variable ou pas.

Si la fonction ne renvoie rien, on écrit ceci :

Type de fonction	Exemple
void	`void maFonction() { //Éléments de la fonction }`

Si la fonction renvoie une variable le type de la fonction que vous voulez programmer est le type de la variable que vous voulez retourner.

Exemple :

Voici ci-dessous un exemple d'une fonction qui ne renvoie aucune variables mais qui prend 2 paramètre :

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

void dessineRectangle(int l, int h)
{
    for(int ligne(0); ligne < h; ligne++)
    {
        for(int colonne(0); colonne < l; colonne++)
        {
            cout << "*";
        }
        cout << endl;
    }
}

int main()
{
    int largeur, hauteur;
    cout << "Largeur du rectangle : ";
    cin >> largeur;
    cout << "Hauteur du rectangle : ";
    cin >> hauteur;

    dessineRectangle(largeur, hauteur);
    return 0;
}
```

Passage par valeur et passage par référence :

Exemple pour les explication : fonction qui ajoute 2 à un nombre passé en paramètre.

Passage par valeur :

```cpp
int ajouteDeux(int a)
{
    a+=2;
    return a;
}
```

Le passage par valeur veut dire que, quand un paramètre est passer à la fonction, la valeurs de la variable passé en paramètre est copié dans une seconde variable qui est à l'intérieur de la fonction. Ce qui veut dire que l'on occupe des cases mémoires supplémentaire. Par contre la variable de départ passé en paramètre ne change pas de valeurs, seule la variable de la fonction change de valeurs

Passage par référence :

```cpp
int ajouteDeux(int& a) //Notez le petit & !!!
{
    a+=2;
    return a;
}
```

Le passage par référence veut dire que, quand un paramètre est passé, l'ont lui attribut un alias qui est le nom de la variable de la fonction. Ce qui veut dire que le minimum de case mémoires sont utilisé. Mais **ATTENTION !** ceci modifie dans le cas de notre exemple la valeur du paramètre.

Exemple d'une fonction `echange()` avec un passage par rérérence :

```cpp
void echange(double& a, double& b)
{
    double temporaire(a); //On sauvegarde la valeur de 'a'
    a = b;                //On remplace la valeur de 'a' par celle de 'b'
    b = temporaire;       //Et on utilise la valeur sauvegardée pour mettre l'ancienne valeur de 'a' dans 'b'
}

int main()
{
    double a(1.2), b(4.5);

    cout << "a vaut " << a << " et b vaut " << b << endl;

    echange(a,b);   //On utilise la fonction

    cout << "a vaut " << a << " et b vaut " << b << endl;
    return 0;
}
```

Passage par référence constante :

Le passage par référence constante permet d'avoir l'avantage de la référence évitant ainsi la copie de la variable et de ne pas pouvoir modifier l'argument de la fonction. Voici un code d'exemple :

```cpp
void f1(string const& texte);  //Pas de copie et pas de modification possible
{
}
```

Utiliser plusieurs fichier pour fragmenter son programme :

Nous allons créer notre fonction `ajouteDeux()` dans un fichier séparé.

```
int ajouteDeux(int a)
{
    a+=2;
    return a;
}
```

Contenue du fichier `math.h` :

```
#ifndef MATH_H_INCLUDED
#define MATH_H_INCLUDED

int ajouteDeux(int nombreRecu);

#endif // MATH_H_INCLUDED
```

Dans le fichier source, il suffit de mettre le code de notre fonction et tous en haut du fichier d'inclure le **prototype** : `#include "math.h"`

Arguments avec des valeurs par défaut :

Pour mettre des valeurs par défaut au arguments d'une fonction, il suffit de mettre : `= valeur`.

```
int nombreDeSecondes(int heures, int minutes = 0, int secondes = 0);
```


6 - Les tableaux

Les tableaux statiques :

Les tableaux permettent de regrouper plusieurs informations de même type dans une seule variable et ainsi de gagner du temps sur l'affichage ou le traitement.

Déclaration d'un tableau statique :

Un tableau statique ce déclare de cette manière : **TYPE NOM [TAILLE] ;**

Voici un exemple de déclaration de tableau statique :

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

int main()
{
    int meilleurScore[5];       //Déclare un tableau de 5 int

    double anglesTriangle[3];   //Déclare un tableau de 3 double

    return 0;
}
```

Attention !, si vous souhaiter déclarer la taille du tableaux avec une variable, il faut **ABSOLUMENT** que la variable soit une constante !

Accès aux éléments du tableau :

Pour accéder aux éléments du tableau, il suffit d'appelé la variable en rajoutant des crochets et la position désirer que ce soit pour le remplissage ou le lecture de la case que vous souhaiter sélectionner.

Exemple :

```cpp
meilleurScore[2] = 5;
```

Pour le parcourir, nous pouvons utiliser une boucle `for` comme ceci :

```cpp
double moyenne(0);
for(int i(0); i<nombreNotes; ++i)
{
    moyenne += notes[i];   //On additionne toutes les notes
}
//En arrivant ici, la variable moyenne contient la somme des notes (79.5)
//Il ne reste donc qu'à diviser par le nombre de notes
moyenne /= nombreNotes;
```

Parcourir son tableau avec une fonction :

Pour parcourir son tableau avec une fonction, il nous faut savoir qu'il y as quelque restriction :

- Une fonction ne peut pas renvoyer de tableau statique.
- Un tableau statique ne peut passer en argument de fonction qu'en temps que référence.

Bien sur pour cela il n'est pas nécessaire de mettre le & après le type de variable, ceci est fait automatiquement. Par contre il faut rajouter un second argument qui contiendra la taille du tableau.

Voici un exemple pour illustrer mes propos :

```cpp
/*
 *  Fonction qui calcule la moyenne des éléments d'un tableau
 *  - tableau : Le tableau dont on veut la moyenne
 *  - tailleTableau : La taille du tableau
 */
double moyenne(double tableau[], int tailleTableau)
{
    double moyenne(0);
    for(int i(0); i<tailleTableau; ++i)
    {
        moyenne += tableau[i];   //On additionne toutes les valeurs
    }
    moyenne /= tailleTableau;

    return moyenne;
}
```

Les tableaux dynamiques :

Les tableaux dynamique ce base sur le même principe que les tableaux statiques mais eux peuvent prendre toutes les tailles. Il peuvent par exemple contenir que 3 valeurs, ou bien 8, ou 19...

Déclaration d'un tableau dynamique :

Pour déclarer un tableau dynamique, il faut commencer par importer une biothèque de cette manière : `#include <vector>`.

Ensuite pour la déclaration à proprement dite, il faut faire comme ceci : **vector<TYPE> NOM (TAILLE) ;**

Voici un exemple de déclaration de tableau dynamique :

```cpp
#include <iostream>
#include <vector> //Ne pas oublier !
using namespace std;

int main()
{
    vector<int> tableau(5);

    return 0;
}
```

REMARQUE : Avec ce type de tableaux, on peut directement remplir toutes les cases du tableau avec une donnée.

Voici un exemple :

```
vector<int> tableau(5, 3);  //Crée un tableau de 5 entiers valant tous 3
vector<string> listeNoms(12, "Sans nom");
//Crée un tableau de 12 strings valant toutes « Sans nom »
```

Accès aux éléments du tableau dynamique :

Pour accéder aux éléments d'un tableaux dynamique, la méthode est la même que pour accéder aux éléments d'un tableau statique.

Changer la taille :

Pour changer la taille du tableau on peut faire plusieurs chose.

Commençons par ajouter une valeur en fin de tableau :

```
vector<int> tableau(3,2);  //Un tableau de 3 entiers valant tous 2
tableau.push_back(8);
//On ajoute une 4ème case au tableau qui contient la valeur 8
```

Maintenant supprimons la dernière valeur du tableau :

```
vector<int> tableau(3,2); //Un tableau de 3 entiers valant tous 2
tableau.pop_back(); //Et hop ! Plus que 2 cases
tableau.pop_back(); //Et hop ! Plus que 1 case
```

Pour finir, voici la dernière méthode que je vous présenterai, il s'agit de la méthode permettant de savoir la taille du tableau :

```
vector<int> tableau(5,4); //Un tableau de 5 entiers valant tous 4
int const taille(tableau.size());
//Une variable qui contient la taille du tableau
//La taille peut varier mais la valeur de cette variable ne changera pas
//On utilise donc une constante
//À partir d'ici, la constante 'taille' vaut donc 5
```

Les tableaux dynamiques et les fonctions :

Dans le cas d'un tableau dynamique, le traitement pour une fonction est plus simple, il suffit de passer le tableau en passage par référence constante comme ceci :

```
1 //Une fonction recevant un tableau d'entiers en argument
2 void fonction(vector<int> const& a)
3 {
4     //…
5 }
```

Les tableaux multi-dimensionnel :

Les tableaux multi-dimensionnel ce déclare de la même manière que les tableaux statique, ce sont des tableaux statique mais avec 2 dimension.

Il ce déclare de la manière suivante :

```
int tableau[5][4];
//tableau[tailleX][tailleY];
```

Ce tableau contiendra en résumer 4 tableaux statique de 5 cases.

7 - Lire et modifier des fichiers

Écrire dans un fichier :

Dans un premier temps, pour manipuler les fichiers, il faut importer un module. Ce module s'importe comme d'habitude avec un `#include` et s'intitule `<fstream>`.

Pour l'importer, il faut faire comme ceci : `#include <fstream>`.

Ouvrir un fichier en écriture :

Pour ouvrir un fichier en écriture, il suffit de faire comme dans l'exemple ci-dessous :

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

int main()
{
   ofstream monFlux("C:/Nanoc/scores.txt");
   //Déclaration d'un flux permettant d'écrire dans le fichier
   // C:/Nanoc/scores.txt
   return 0;
}
```

Mais avec cette technique qui est la base, il peut y avoir des problèmes, alors on teste si le fichier est bien ouvert avant de faire quoi que ce soit :

```cpp
ofstream monFlux("C:/Nanoc/scores.txt");  //On essaye d'ouvrir le fichier

if(monFlux)  //On teste si tout est OK
{
    //Tout est OK, on peut utiliser le fichier
}
else
{
    cout << "ERREUR: Impossible d'ouvrir le fichier." << endl;
}
```

Écrire dans un flux :

Maintenant que notre fichier est ouvert, nous pouvons écrire à l'intérieur. La technique est la même que pour afficher du texte dans la console. Il faut utiliser les chevron :

```cpp
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main()
{
    string const nomFichier("C:/Nanoc/scores.txt");
    ofstream monFlux(nomFichier.c_str());

    if(monFlux)
    {
        monFlux << "Bonjour, je suis une phrase écrite dans un fichier." << endl;
        monFlux << 42.1337 << endl;
        int age(23);
        monFlux << "J'ai " << age << " ans." << endl;
    }
    else
    {
        cout << "ERREUR: Impossible d'ouvrir le fichier." << endl;
    }
    return 0;
}
```

Attention !, si vous souhaiter ajouter des informations à la fin du documents s'il existe, il faut écrire dans la déclaration: `ofstream monFlux("Chemin", ios::app);` !

Lire dans un fichier :

Déclaration du flux :

Pour lire dans un fichier la méthode est la même concernent la déclaration du flux. Sauf que dans le cas d'une ouverture, la classe à appeler est : `ifstream`.

Lecture dans le flux :

Pour lire le contenu d'un fichier, il y a 3 manière de le lire :

- Ligne par ligne, avec `getline()`;
- Mot par mot, avec les chevrons `>>`;
- Caractère par caractère, avec `get()`.

Ligne par ligne :

```
string ligne;
getline(monFlux, ligne); //On lit une ligne complète
```

Mot par mot :

```
double nombre;
monFlux >> nombre; //Lit un nombre à virgule depuis le fichier
string mot;
monFlux >> mot;    //Lit un mot depuis le fichier
```

Caractère par caractère :

```
char a;
monFlux.get(a);
```

Si plusieurs méthode de lecture sont employé :

```
ifstream monFlux("C:/Nanoc/C++/data.txt");

string mot;
monFlux >> mot;          //On lit un mot depuis le fichier

monFlux.ignore();        //On change de mode

string ligne;
getline(monFlux, ligne); //On lit une ligne complète
```

Lire un fichier dans sa globalité :

```cpp
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main()
{
    ifstream fichier("C:/Nanoc/fichier.txt");

    if(fichier)
    {
        //L'ouverture s'est bien passée, on peut donc lire

        string ligne; //Une variable pour stocker les lignes lues

        while(getline(fichier, ligne)) //Tant qu'on n'est pas à la fin, on lit
        {
            cout << ligne << endl;
            //Et on l'affiche dans la console
            //Ou alors on fait quelque chose avec cette ligne
            //À vous de voir
        }
    }
    else
    {
        cout << "ERREUR: Impossible d'ouvrir le fichier en lecture." << endl;
    }

    return 0;
}
```

Quelques astuces :

Fermer prématurément un fichier :

Pour refermer un fichier prématurément, il suffit d'écrire : `monFlux.close()`.

TP1 - Le mot mytère

```cpp
#include <iostream>
#include <string>
#include <ctime>
#include <cstdlib>

using namespace std;

string melangerLettres(string mot)
{
    string melange;
    int position(0);

    //Tant qu'on n'a pas extrait toutes les lettres du mot
    while (mot.size() != 0)
    {
        //On choisit un numéro de lettre au hasard dans le mot
        position = rand() % mot.size();
        //On ajoute la lettre dans le mot mélangé
        melange += mot[position];
        //On retire cette lettre du mot mystère
        //Pour ne pas la prendre une deuxième fois
          mot.erase(position, 1);
     }

    //On renvoie le mot mélangé
    return melange;
}

int main()
{
    string motMystere, motMelange, motUtilisateur;

    //Initialisation des nombres aléatoires
    srand(time(0));

    //1 : On demande de saisir un mot
    cout << "Saisissez un mot" << endl;
    cin >> motMystere;

    //2 : On récupère le mot avec les lettres mélangées dans motMelange
    motMelange = melangerLettres(motMystere);

    //3 : On demande à l'utilisateur quel est le mot mystère
    do
    {
        cout << endl << "Quel est ce mot ? " << motMelange << endl;
        cin >> motUtilisateur;

        if (motUtilisateur == motMystere)
        {
            cout << "Bravo !" << endl;
        }
        else
        {
            cout << "Ce n'est pas le mot !" << endl;
        }
    }while (motUtilisateur != motMystere);
    //On recommence tant qu'il n'a pas trouvé

     return 0;
}
```

8 - Les pointeurs

Les pointeurs sont des variables qui sont capable de contenir que des adresse de variables déclaré.

Afficher l'adresse d'une variable :

Pour afficher l'adresse d'une variable, il suffit de rajouter le préfixe & au nom de la variable. Voici ci-après un exemple :

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

int main()
{
   int ageUtilisateur(16);
   cout << "L'adresse est : " << &ageUtilisateur << endl;
   //Affichage de l'adresse de la variable
   return 0;
}
```

L'adresse retourné sera codé en hexadécimal sous le format suivant : `0x22ff1c`.

Les pointeurs :

Déclarer un pointeur :

Pour déclarer un pointeur, il faut ajouter le préfixe * au nom du pointeur. Pour savoir quelle type mettre au pointeur, il faut mettre le même type que l'adresse de la variable pointer. Voici un exemple :

Attention !, il ne faut JAMAIS déclarer un pointeur vide, il faut toujours le déclarer avec la valeur 0 !

```cpp
int *pointeur(0);
double *pointeurA(0);
unsigned int *pointeurB(0);
string *pointeurC(0);
vector<int> *pointeurD(0);
int const *pointeurE(0);
```

Stocker une adresse dans un pointeur :

Pour stocker une adresse dans un pointeur, il faut procéder comme ceci :

```cpp
int main()
{
    int ageUtilisateur(16);
    //Une variable de type int
    int *ptr(0);
    //Un pointeur pouvant contenir l'adresse d'un nombre entier

    ptr = &ageUtilisateur;
    //On met l'adresse de 'ageUtilisateur' dans le pointeur 'ptr'

    return 0;
}
```

Afficher l'adresse du pointeur :

Pour visualiser l'adresse du pointer par le pointeur, il suffit de la montrer dans le flux de sortie standard comme ci-dessous :

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

int main()
{
    int ageUtilisateur(16);
    int *ptr(0);

    ptr = &ageUtilisateur;

    cout << "L'adresse de 'ageUtilisateur' est : " << &ageUtilisateur << endl;
    cout << "La valeur de pointeur est : " << ptr << endl;

    return 0;
}
```

Accéder à la valeur pointer :

Pour accéder à la valeur pointer, rien de plus simple, il suffit de mettre le pointeur dans le flux de sortie standard en rajoutant le préfixe * comme ci-après :

```
int main()
{
   int ageUtilisateur(16);
   int *ptr(0);

   ptr= &ageUtilisateur;

   cout << "La valeur est :  " << *ptr << endl;

   return 0;
}
```

L'allocation dynamique de la mémoire :

L'allocation dynamique de la mémoire permet d'avoir le contrôle complet de la mémoire dans le programme. Voici un exemple pour vous faire comprendre le fonctionnement :

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

int main()
{
   int* pointeur(0);
   pointeur = new int;

   cout << "Quel est votre age ? ";
   cin >> *pointeur;
   //On écrit dans la case mémoire pointée par le pointeur 'pointeur'

   cout << "Vous avez " << *pointeur << " ans." << endl;
   //On utilise à nouveau *pointeur
    delete pointeur;   //Ne pas oublier de libérer la mémoire
   pointeur = 0;       //Et de faire pointer le pointeur vers rien

   return 0;
}
```

Chapitre 2 - La POO (Programmation orienté objet)

CHAPITRE 2 : La programmation orienté objet

1 - La vérité sur les string

Les méthode de l'objet `string` :

Méthode `.size()` :

La méthode `.size()` permet de connaître la longueur de la chaîne de caractère.

Méthode `.erase()` :

Permet de supprimer tout le contenu de la chaîne de caractère.

Méthode `.c_str()` :

Permet de renvoyer un pointeur vers le tableau de *`char`* que contient l'objet `string`.

2 - Les classes

Créer une classe :

Pour créer une classe, il faut déjà lui trouver un nom comme pour une variable, mais ATTENTION, le nom d'une classe doit toujours commencer par une majuscule. Dans cette classe nous allons mettre des attribut qui eux on comme particularité de toujours commencer par `"m_"`. Ensuite vous pourrez mettre vos méthodes qui elle reprennent tous les concepts des fonctions.

Voici un exemple de prototype de classe :

```cpp
class Personnage
{
    // Méthodes
    void recevoirDegats(int nbDegats)
    {

    }

    void attaquer(Personnage &cible)
    {

    }

    void boirePotionDeVie(int quantitePotion)
    {

    }

    void changerArme(string nomNouvelleArme, int degatsNouvelleArme)
    {

    }

    bool estVivant()
    {

    }

    // Attributs
    int m_vie;
    int m_mana;
    string m_nomArme;
    int m_degatsArme;
};
```

Attention !, il faut penser à mettre un point virgule après la class !!!!

Droits d'accès et encapsulation :

Dans la POO, il existe deux type de droits d'accès :

- public : les méthodes et attributs peuvent être appeler de l'extérieur;
- private : les méthodes et attributs ne peuvent être appeler de nulle par, par défaut tous ce qui est dans une classe est private.

Maintenant voici le code précédant mais en tenant compte des droits d'accès :

```cpp
class Personnage
{
    // Tout ce qui suit est public (accessible depuis l'extérieur)
    public:

    void recevoirDegats(int nbDegats)
    {

    }

    void attaquer(Personnage &cible)
    {

    }

    void boirePotionDeVie(int quantitePotion)
    {

    }

    void changerArme(string nomNouvelleArme, int degatsNouvelleArme)
    {

    }

    bool estVivant()
    {

    }

    // Tout ce qui suit est privé (inaccessible depuis l'extérieur)
    private:

    int m_vie;
    int m_mana;
    string m_nomArme;
    int m_degatsArme;
};
```

Séparer prototype et définitions :

Le but de séparer prototype et définitions est pour les mêmes raisons que les fonction pour que le programmes soit découper en plusieurs fichier et ainsi portable.

Personnage.h :

```cpp
#ifndef DEF_PERSONNAGE
#define DEF_PERSONNAGE

#include <string>

class Personnage
{
    public:

    void recevoirDegats(int nbDegats);
    void attaquer(Personnage &cible);
    void boirePotionDeVie(int quantitePotion);
    void changerArme(std::string nomNouvelleArme, int degatsNouvelleArme);
    bool estVivant();

    private:

    int m_vie;
    int m_mana;
    std::string m_nomArme; //Pas de using namespace std, il faut donc mettrestd:: devant string
    int m_degatsArme;
};

#endif
```

Personnage.cpp :

```cpp
#include "Personnage.h"

using namespace std;

void Personnage::recevoirDegats(int nbDegats)
{
    m_vie -= nbDegats;
    //On enlève le nombre de dégâts reçus à la vie du personnage

    if (m_vie < 0) //Pour éviter d'avoir une vie négative
    {
        m_vie = 0; //On met la vie à 0 (cela veut dire mort)
    }
}

void Personnage::attaquer(Personnage &cible)
{
    cible.recevoirDegats(m_degatsArme);
    //On inflige à la cible les dégâts que cause notre arme
}

void Personnage::boirePotionDeVie(int quantitePotion)
{
    m_vie += quantitePotion;

    if (m_vie > 100) //Interdiction de dépasser 100 de vie
    {
        m_vie = 100;
    }
}

void Personnage::changerArme(string nomNouvelleArme, int degatsNouvelleArme)
{
    m_nomArme = nomNouvelleArme;
    m_degatsArme = degatsNouvelleArme;
}

bool Personnage::estVivant()
{
    return m_vie > 0;
}
```

Constructeurs et destructeurs :

Le rôle du constructeur est d'initialiser les attributs de la classe et le rôle du destructeurs est de supprimer les attribut allouer dynamiquement.

Constructeur :

```cpp
#include <string>

class Personnage
{
    public:

    Personnage(); //Constructeur
    void recevoirDegats(int nbDegats);
    void attaquer(Personnage &cible);
    void boirePotionDeVie(int quantitePotion);
    void changerArme(std::string nomNouvelleArme, int degatsNouvelleArme);
    bool estVivant();


    private:

    int m_vie;
    int m_mana;
    std::string m_nomArme;
    int m_degatsArme;
};
```

```cpp
Personnage::Personnage()
{
    m_vie = 100;
    m_mana = 100;
    m_nomArme = "Épée rouillée";
    m_degatsArme = 10;
}
```

OU

```cpp
Personnage::Personnage() : m_vie(100), m_mana(100), m_nomArme("Épée rouillée"),
m_degatsArme(10)
{
    //Rien à mettre dans le corps du constructeur, tout a déjà été fait !
}
```

Surcharger le constructeur :

```cpp
Personnage(std::string nomArme, int degatsArme);
```

```cpp
Personnage::Personnage(string nomArme, int degatsArme) : m_vie(100),
m_mana(100),m_nomArme(nomArme), m_degatsArme(degatsArme)
{

}
```

Le destructeur :

```
~Personnage();
```

```
Personnage::~Personnage()
{
    /* Rien à mettre ici car on ne fait pas d'allocation dynamique
    dans la classe Personnage. Le destructeur est donc inutile mais
    je le mets pour montrer à quoi cela ressemble.
    En temps normal, un destructeur fait souvent des delete et quelques
    autres vérifications si nécessaire avant la destruction de l'objet. */
}
```

Les méthodes constantes :

Les méthodes constantes sont des méthodes qui sont que en "lecture seule". Exemple : si vous avez une méthode qui ce contente juste d'afficher quelque chose a l'écran, c'est une méthode constante. Les méthodes constante sont, en résumé, les méthodes qui ne modifie aucun attribut.

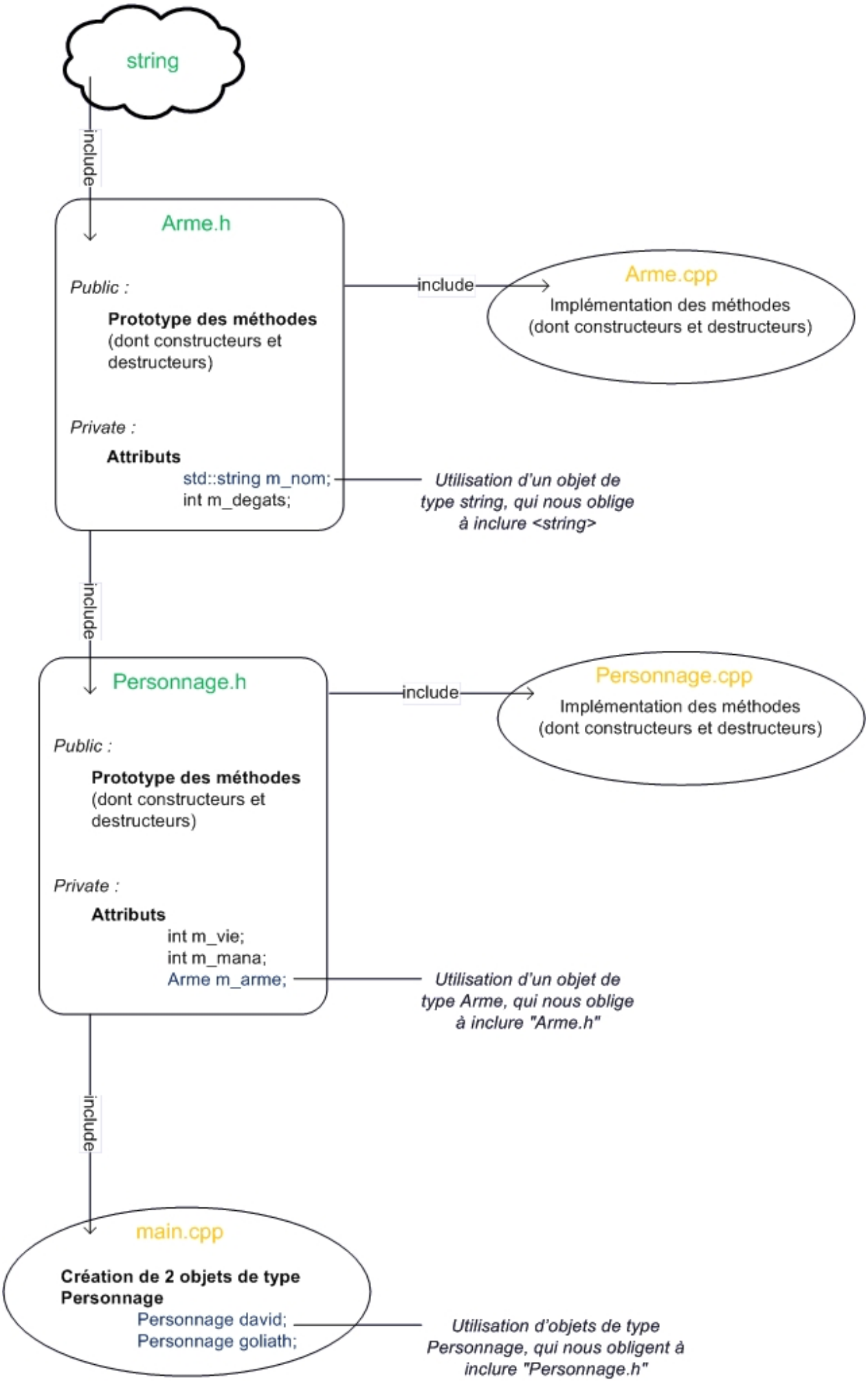
Exemple de méthode constante :

```
//Prototype de la méthode (dans le .h) :
void maMethode(int parametre) const;


//Déclaration de la méthode (dans le .cpp) :
void MaClasse::maMethode(int parametre) const
{

}
```

Associer des classe entre elles :

Pour faire cela, il suffit d'appeler une autre classe dans notre classe et de mettre un attribut avec notre classe.

MEGA RESUME :


string
include
Arme.h
Public :
Prototype des méthodes
(dont constructeurs et
destructeurs)
Private :
Attributs
std::string m_nom;
int m_degats;
include
Arme.cpp
Implémentation des méthodes
(dont constructeurs et destructeurs)
Utilisation d'un objet de
type string, qui nous oblige
à inclure <string>
include
Personnage.h
Public :
Prototype des méthodes
(dont constructeurs et
destructeurs)
Private :
Attributs
int m_vie;
int m_mana;
Arme m_arme;
include
Personnage.cpp
Implémentation des méthodes
(dont constructeurs et destructeurs)
Utilisation d'un objet de
type Arme, qui nous oblige
à inclure "Arme.h"
include
main.cpp
Création de 2 objets de type
Personnage
Personnage david;
Personnage goliath;
Utilisation d'objets de type
Personnage, qui nous obligent à
inclure "Personnage.h"

3 - Les surcharges d'opérateurs

Notre classe type pour ce chapitre :

Durée.h :

```
#ifndef DEF_DUREE
#define DEF_DUREE

class Duree
{
    public:

    Duree(int heures = 0, int minutes = 0, int secondes = 0);

    private:

    int m_heures;
    int m_minutes;
    int m_secondes;
};

#endif
```

Durée.cpp:

```
#include "Duree.h"

Duree::Duree(int heures, int minutes, int secondes) : m_heures(heures), m_minutes(minutes),
m_secondes(secondes)
{
}
```

Les surcharge d'opérateur permettent dans la cas de notre classe durée de définir comment notre objet va réagir aux opérateur de comparaison et arithmétique.

Les opérateurs de comparaisons :

Exemple avec l'opérateur == :

```
bool operator==(Duree const& a, Duree const& b);
```

Cette fonction est à mettre dans le fichier de la classe pour plus de lisibilité mais pas dans la classe car ce n'est pas une méthode de la classe.

Dans le fichier `.cpp`, il faut ajouter une méthode car cette surcharge ne peut être directement du faite qu'elle est hors de la classe et que les attributs de la classe sont privé.

```
bool Duree::estEgal(Duree const& b) const
{
    //Teste si a.m_heure == b.m_heure etc.
    if (m_heures == b.m_heures && m_minutes == b.m_minutes && m_secondes == b.m_secondes)
        return true;
    else
        return false;
}
```

```
bool operator==(Duree const& a, Duree const& b)
{
    return a.estEgal(b);
}
```

Bien sur ne pas oublier dans le surcharge de passer les arguments en références constante pour ne pas qu'il y ai de copie, ni de modification.

Tableau des opérateurs de comparaison et arithmétiques :

==	operator==
!=	operator!=
<	operator<
>	operator>

<=	operator<=
>=	operator>=
+	operator+
+=	operator+=
-	operator-
-=	operator-=
*	operator*
*=	operator*=
/	operator/
/=	operator/=
%	operator%
%=	operator%=

Les opérateurs de flux :

```
1 bool operator==(Duree const& a, Duree const& b);
```

```cpp
void Duree::afficher(ostream &flux) const
{
    flux << m_heures << "h" << m_minutes << "m" << m_secondes << "s";
}
```

```cpp
bool operator==(Duree const& a, Duree const& b);
```


TP2 - La POO en pratique avec ZFraction

4 - Les classes et les pointeurs

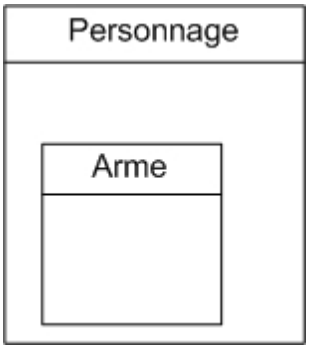
Pointeur d'une classe vers une autre :

Ici, nous allons voir comment un personnage peut changer d'arme.

Maintenant, notre code ressemble à celui-ci :

```cpp
class Personnage
{
    public:

    //Quelques méthodes…

    private:

    Arme m_arme; // L'Arme est "contenue" dans le Personnage
    //…
};
```

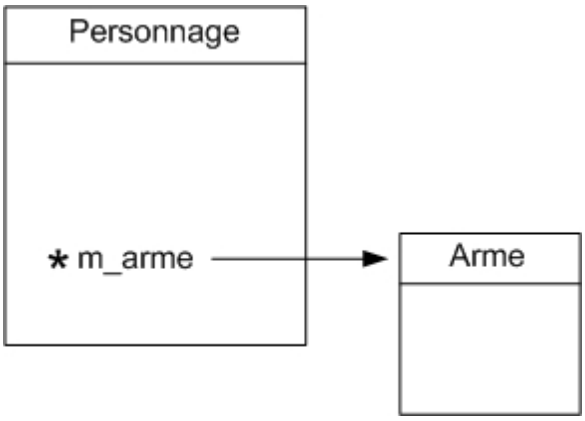
Voici le schéma que l'on peut associer à ce code:



Maintenant, mettons un pointeur à la place d'une instance de classe classique :

```
class Personnage
{
    public:

    //Quelques méthodes…

    private:

    Arme *m_arme;
    //L'Arme est un pointeur, l'objet n'est plus contenu dans le Personnage
    //…
};
```

Voici son schéma :



Maintenant, le personnage peut changer d'arme ou ne plus en avoir car l'objet `Arme` ne lui appartient pas, il pointe vers lui.

Gestion de l'allocation dynamique :

Allocation de la mémoire pour l'objet :

Maintenant, il faut déclarer le pointeur pour qu'il pointe vers quelque chose, sinon sa ne sert à rien. Il faudra écrire le code suivant dans le constructeur de notre classe `Personnage` :

```
1 Personnage::Personnage(string nomArme, int degatsArme) : m_arme(0), m_vie(100), m_mana(100)
2 {
3     m_arme = new Arme(nomArme, degatsArme);
4 }
```

Ensuite, on à le choix de mettre ou non des arguments dans la déclaration suivant quelle constructeur on souhaite appeler.

Attention !, par sécurité, il faut penser à initialiser le pointeur à 0 !!!!

Désallocation de la mémoire pour l'objet :

Il faut penser à supprimer notre pointeur lors de la destruction de `Personnage`, sinon il occupera de la mémoire pour rien. Il faut mettre le code suivant :

```
1 Personnage::~Personnage()
2 {
3     delete m_arme;
4 }
```

Avec ce code dans le destructeur de la classe `Personnage`, tous sera supprimer correctement.

Penser que `m_arme` est un pointeur ! :

Maintenant, il faut penser que `m_arme` est un pointeur, donc remplacer les éléments en rapport avec `m_arme` par `->`.

```
1 void Personnage::attaquer(Personnage &cible)
2 {
3     cible.recevoirDegats(m_arme->getDegats());
4 }
```

Le pointeur this :

Le pointeur this, sert à pointer la classe vers elle même. Utiliser comme sa on peut avoir l'adresse de la classe et utiliser avec *, on peut pointer vers la classe elle même.

Le constructeur de copie :

Le constructeur de copie sert quand vous voulez créer un objet à partie d'un objet.

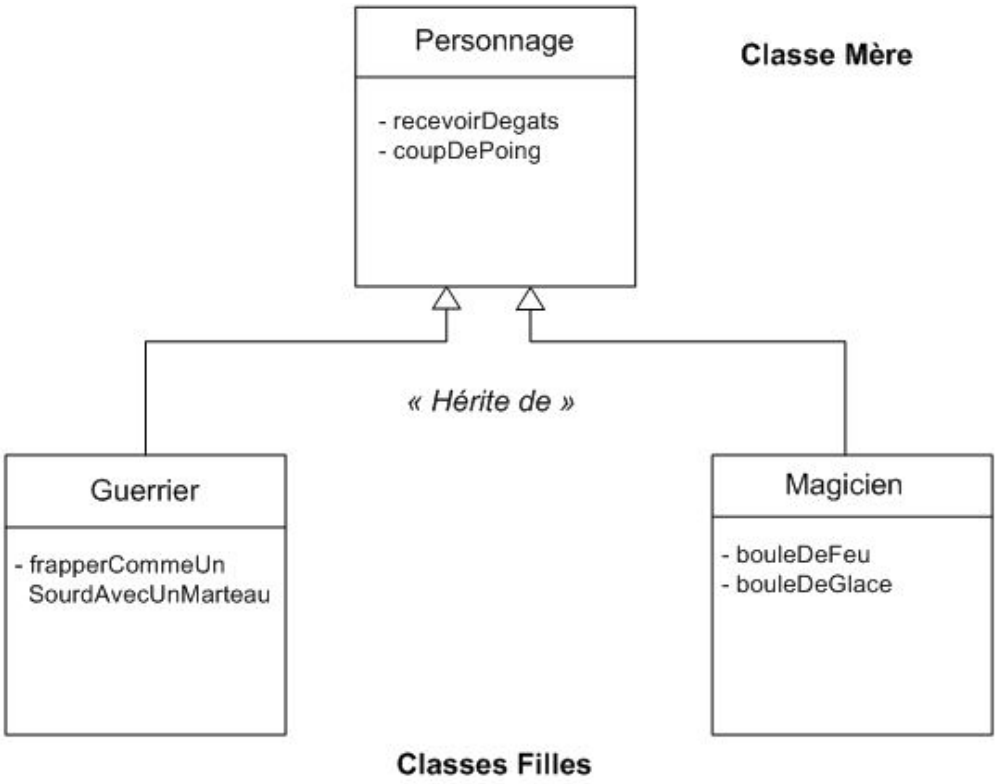
Exemple de code de constructeur de copie :

```
1 Objet(Objet const& objetACopier);
```

```
1 Personnage::Personnage(Personnage const& personnageACopier)
2    : m_vie(personnageACopier.m_vie), m_mana(personnageACopier.m_mana), m_arme(0)
3 {
4
5 }
```


5 - L'héritage

Exemple d'héritage :




```hpp
#ifndef DEF_MAGICIEN
#define DEF_MAGICIEN

#include <iostream>
#include <string>
#include "Personnage.h"

class Magicien : public Personnage
{
    public:
        void bouleDeFeu() const;
        void bouleDeGlace() const;

    private:
        int m_mana;
};

#endif
```

6 - Éléments statiques et amitié

Méthodes statiques :

Une méthode statique est une méthode qui est en réalité une fonction rangé dans une classe. Par exemple, dans la classe `QDate` de la bibliothèque Qt, il y a une méthode statique `QDate::currentDate()` qui renvoie la date actuelle. Pour cela pas besoin de créer une instance de classe juste pour ça, alors on l'a déclarer comme méthode statique.

Voici comment on déclare une méthode statique:

```
class MaClasse
{
    public:
    MaClasse();
    static void maMethode();
};
```

Ensuite, pour l'écriture du code C++ dans le fichier cpp, pas besoin d'écrire le mot clé statique.

Une fois le code écrit, pour appeler cette méthode, il suffit d'écrire ceci dans le main:

```
int main()
{
    MaClasse::maMethode();

    return 0;
}
```

Attribut statique :

De la même façon, il existe des attribut statique, il peuvent être déclarer dans l'espace `private` sans soucis, c'est une exeption.

Attention !, SUBTILITER, il faut déclarer ces attribut dans l'espace globale !!! Pas dans la classe !!!

Voici comment on les initialise:

```cpp
//Initialiser l'attribut en dehors de toute fonction ou classe (espace global)
int MaClasse::monAttribut = 5;
```

Ces attribut sont considérer comme des variables globale, ils sont accessible partout dans le code.

L'amitié :

Le but de l'amitié est d'appeler une méthode privée ou un attribut placé dans la partie privée de la classe par une fonction déclarer comme amis de la classe.

Par exemple dans le cas de la surcharge d'opérateur, la méthode de la classe appeler lors de la surcharge de l'opérateur est mieux dans la partie privé de la classe pour ne pas être utiliser a mauvais hessiens. Mais si la méthode est dans la partie privée de la classe, alors la surcharge de l'opérateur n'a plus accès à cette méthode, pour cela, il faut déclarer la fonction surcharge comme étant amis avec la classe qui contient la méthode.
Pour cela, il faut écrire le mot clé `friend` devant la fonction concerné et la mettre dans la classe, comme ceci :

```cpp
class Duree
{
    public:

    Duree(int heures = 0, int minutes = 0, int secondes = 0);

    private:

    void affiche(ostream& out) const;  //Permet d'écrire la durée dans un flux

    int m_heures;
    int m_minutes;
    int m_secondes;

    friend std::ostream& operator<< (std::ostream& flux, Duree const& duree);
};
```

Chapitre 3 - Créer vos propres fenêtres avec Qt

CHAPITRE 3 : Créer vos propres fenêtres avec Qt

1 - Compiler votre première fenêtre avec Qt

Code du fichier .pro :

```
QT += widgets

SOURCES += \
    main.cpp
```

Voici le code à mettre dans le fichier .pro de votre projet pour importer les composants graphiques de Qt.

Code minimal :

```cpp
#include <QApplication>
#include <QPushButton>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    QPushButton bouton("Salut les Zéros, la forme ?");
    bouton.show();

    return app.exec();
}
```

Ci-dessus, nous voyons le code minimal à mettre dans notre `main.cpp` pour avoir une fenêtre.
Avec ce code, nous avons une fenêtre avec un bouton qui contient le texte "Salut les Zéros, la forme ?".

2 - Personnaliser les widgets

Personnalisé les widgets :

Pour personnalisé les widgets, rien de plus simple, il faut utiliser des méthodes (accesseur ou mutateur) pour modifier le comportement des attributs du widget. Pour cela, il faut pour récupérer une valeurs, mettre le préfixe `get` devant et pour modifier une propriété du widget mettre le préfixe `set`.

Quelque propriété utile :
- toolTip -> infobulle
- font -> `button.setFont(QFont("Courrier"))`
- cursor -> le curseur
- icon -> `setIcon(QIcon("smile.png"))`

Créer une fenêtre dans une classe :

Voici ci-dessous un exemple de fenêtre dans une classe. Cette méthode est très fortement conseiller car en cas de plusieurs fenêtres, il suffit de créer plusieurs classes.

```h
#ifndef DEF_MAFENETRE
#define DEF_MAFENETRE

#include <QApplication>
#include <QWidget>
#include <QPushButton>

class MaFenetre : public QWidget // On hérite de QWidget (IMPORTANT)
{
    public:
    MaFenetre();

    private:
    QPushButton *m_bouton;
};

#endif
```

```cpp
#include "MaFenetre.h"

MaFenetre::MaFenetre() : QWidget()
{
    setFixedSize(300, 150);

    // Construction du bouton
    m_bouton = new QPushButton("Pimp mon bouton !", this);

    m_bouton->setFont(QFont("Comic Sans MS", 14));
    m_bouton->setCursor(Qt::PointingHandCursor);
    m_bouton->setIcon(QIcon("smile.png"));
    m_bouton->move(60, 50);
}
```

```cpp
#include <QApplication>
#include "MaFenetre.h"


int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    MaFenetre fenetre;
    fenetre.show();

    return app.exec();
}
```

3 - Utilisez les signaux et les slots

Le principe des signaux et slots

Les signaux et les slots permettent de connecter des actions faites sur les widgets avec des méthodes de notre classe pour par exemple fermer le programme quand on clique sur un bouton.

Les signaux sont les actions faite sur le widget. Par exemple: cliquer, relachement...

Les slots sont les méthodes qui sont connecter avec signaux. Dans l'exemple, le slot est la méthode qui ferme l'application.

Connexion entre les signaux et les slots:

La ligne de code à écrire est: `QObject::connect(m_bouton, SIGNAL(clicked()), qApp, SLOT(quit()))`

Ici, `m_bouton` est le widget concerné, `clicked` est le type de signal, qApp est la classe qui contient le slot et `quit()` est le slot.

Chapitre 4 - Utiliser la bibliothèque standard

CHAPITRE 4 : Utiliser la bibliothèque standard

1 - Les conteneurs

Chapitre 5 - Notions avancés

CHAPITRE 5 : Notions avancés

1 - Les exeptions

Suppléments Qt

Utiliser le presse papier avec Qt

Sécurisation d'un chat avec Qt

Le hachage avec Qt

Annexes

Logiciel à télécharger (IDE)