C++

Table des matières

[Premier projet 7](#_Toc473789092)

[Les variables : 7](#_Toc473789093)

[Strings : 7](#_Toc473789094)

[Décaration : 8](#_Toc473789095)

[Afficher une variable : 8](#_Toc473789096)

[Les références : 8](#_Toc473789097)

[Une vrai calculatrice 10](#_Toc473789098)

[Intéragir avec l’utiisateur : 10](#_Toc473789099)

[Moyen mémotechnique : 10](#_Toc473789100)

[getLine : 11](#_Toc473789101)

[Déclarer une constante : 12](#_Toc473789102)

[L’incrémentation : 12](#_Toc473789103)

[Cmath : 12](#_Toc473789104)

[Pow : 13](#_Toc473789105)

[Les structures de contrôles 14](#_Toc473789106)

[Conditions : 14](#_Toc473789107)

[If : 14](#_Toc473789108)

[Switch : 14](#_Toc473789109)

[Boolean : 15](#_Toc473789110)

[Boucles : 15](#_Toc473789111)

[While : 15](#_Toc473789112)

[Do … while : 15](#_Toc473789113)

[For : 16](#_Toc473789114)

[Les fonctions 17](#_Toc473789115)

[Créer une fonction : 17](#_Toc473789116)

[Passage par valeur et par référence : 18](#_Toc473789117)

[Par valeur : 18](#_Toc473789118)

[Par référence : 18](#_Toc473789119)

[Utiliser plusieurs fichiers : 19](#_Toc473789120)

[Valeurs par défaut pour les arguments : 20](#_Toc473789121)

[Les tableaux : 21](#_Toc473789122)

[Les tableaux statiques : 21](#_Toc473789123)

[Décalrer : 21](#_Toc473789124)

[Les tableaux dans les fonctions : 21](#_Toc473789125)

[Tableaux dynamiques : 22](#_Toc473789126)

[Déclarer : 22](#_Toc473789127)

[Push\_back et pop\_back : 22](#_Toc473789128)

[Vector et fonction : 23](#_Toc473789129)

[Tableaux multidimensionnels : 23](#_Toc473789130)

[Déclaration : 23](#_Toc473789131)

[Aller plus loin : 24](#_Toc473789132)

[Les strings comme tableau : 24](#_Toc473789133)

[Accéder aux lettres : 24](#_Toc473789134)

[Les fonctions : 24](#_Toc473789135)

[Lire et modifier des fichiers 25](#_Toc473789136)

[Ecrire dans un fichier : 25](#_Toc473789137)

[Entête fstream : 25](#_Toc473789138)

[Lire un fichier : 26](#_Toc473789139)

[Ligne par ligne : 26](#_Toc473789140)

[Mot par mot : 26](#_Toc473789141)

[Lire par caractère : 27](#_Toc473789142)

[Exemple de lecture : 27](#_Toc473789143)

[Astuces : 27](#_Toc473789144)

[Conaitre sa position dans le fichier : 28](#_Toc473789145)

[Se déplacer : 28](#_Toc473789146)

[Connaître la taille du fichier : 28](#_Toc473789147)

[Générer un mot mystère : 29](#_Toc473789148)

[Les pointeurs 30](#_Toc473789149)

[Adresse d’une variable : 30](#_Toc473789150)

[Pointeurs : 30](#_Toc473789151)

[Déclarer un pointeur : 30](#_Toc473789152)

[Accéder à la valeur de la variable pointée : 31](#_Toc473789153)

[Allocation dynamique : 31](#_Toc473789154)

[Gestion automatique de la mémoire : 31](#_Toc473789155)

[Allouer de la mémoire (manuel) : 32](#_Toc473789156)

[Libérer la mémoire : 32](#_Toc473789157)

[Quand utiliser des pointeurs ? 33](#_Toc473789158)

[Les strings 34](#_Toc473789159)

[Créer et utiliser les strings : 34](#_Toc473789160)

[Concaténation : 34](#_Toc473789161)

[Comparer : 34](#_Toc473789162)

[Méthodes des strings : 34](#_Toc473789163)

[Size : 34](#_Toc473789164)

[Erase : 35](#_Toc473789165)

[Substr : 35](#_Toc473789166)

[C\_str : 35](#_Toc473789167)

[Les classes 36](#_Toc473789168)

[Créer une classe : 36](#_Toc473789169)

[Les attributs : 36](#_Toc473789170)

[Les méthodes : 36](#_Toc473789171)

[Droit d’accès et encapsulations : 37](#_Toc473789172)

[Droits d’accès : 37](#_Toc473789173)

[Séparer prototype et définition : 38](#_Toc473789174)

[Constructeur et destructeurs : 40](#_Toc473789175)

[Le constructeur : 40](#_Toc473789176)

[Liste d’initialisation : 40](#_Toc473789177)

[Surcharger le constructeur : 41](#_Toc473789178)

[Constructeur de copie : 41](#_Toc473789179)

[Destruceur : 41](#_Toc473789180)

[Méthodes constantes : 42](#_Toc473789181)

[Associer des classes entre elles : 42](#_Toc473789182)

[Surcharge d’opérateurs 46](#_Toc473789183)

[Préparatifs : 46](#_Toc473789184)

[Les opérateurs de comparaison : 47](#_Toc473789185)

[Implémenter l’opérateur : 47](#_Toc473789186)

[L’opérateur != : 48](#_Toc473789187)

[L’opérateur < : 48](#_Toc473789188)

[Opérateur arithmétiques : 49](#_Toc473789189)

[L’addition : 49](#_Toc473789190)

[+= : 49](#_Toc473789191)

[Les opérateurs de flux : 50](#_Toc473789192)

[Opérateur << : 50](#_Toc473789193)

[TP, ZFraction 51](#_Toc473789194)

[Préparatifs : 51](#_Toc473789195)

[Classes et pointeurs 52](#_Toc473789196)

[Pointeur d’une classe vers une autre classe : 52](#_Toc473789197)

[Gestion de l’allocation dynamique : 53](#_Toc473789198)

[Désallouer de la mémoire pour l’objet : 53](#_Toc473789199)

[Le pointeur this : 54](#_Toc473789200)

[Constructeur de copie : 54](#_Toc473789201)

[Création du constructeur de copie : 55](#_Toc473789202)

[La surcharge d’affectation : 56](#_Toc473789203)

[L’héritage 57](#_Toc473789204)

[Héritage simple : 57](#_Toc473789205)

[La dérivation de type : 58](#_Toc473789206)

[Héritage et constructeur : 58](#_Toc473789207)

[Appeler le constructeur de la classe mère : 58](#_Toc473789208)

[Transmission des paramètres : 58](#_Toc473789209)

[Portée protected : 59](#_Toc473789210)

[Le masquage : 60](#_Toc473789211)

[La masquage : 60](#_Toc473789212)

[Le démasquage : 61](#_Toc473789213)

[Le polymorphisme 62](#_Toc473789214)

[La résolution des liens : 62](#_Toc473789215)

[Résolution statique des liens : 62](#_Toc473789216)

[Résolution dynamique des liens : 63](#_Toc473789217)

[Les fonctions virtuelles : 63](#_Toc473789218)

[Déclarer une méthode virtuelle : 63](#_Toc473789219)

[Utiliser une référence : 64](#_Toc473789220)

[Les méthodes spéciales : 64](#_Toc473789221)

[Les collections hétérogènes : 66](#_Toc473789222)

[Retour des pointeurs : 66](#_Toc473789223)

[Les fonctions virtuelles pures : 67](#_Toc473789224)

[Les classes abstraites : 68](#_Toc473789225)

[Eléments statiques et amitié 69](#_Toc473789226)

[Les méthodes statiques : 69](#_Toc473789227)

[Les attributs statiques : 69](#_Toc473789228)

[L’amitité : 70](#_Toc473789229)

[L’amitié et la responsabilité : 72](#_Toc473789230)

[Introduction à Qt 73](#_Toc473789231)

[Les fenêtres : 73](#_Toc473789232)

[GUI : 73](#_Toc473789233)

[Présentation : 73](#_Toc473789234)

[Compiler une première fenêtre 73](#_Toc473789235)

[Affichage d’un widget : 74](#_Toc473789236)

[Diffuser le programme : 74](#_Toc473789237)

[Modifier les propriétés d’un widget 75](#_Toc473789238)

[Modifier et personnaliser : 75](#_Toc473789239)

[Text : 75](#_Toc473789240)

[toolTip() l’infobulle : 75](#_Toc473789241)

[Font : la police : 75](#_Toc473789242)

[Cursor : curseur de la souris : 77](#_Toc473789243)

[Icon : l’icône du cursor : 77](#_Toc473789244)

[Qt et l’héritage : 77](#_Toc473789245)

[QObject la classe de base : 77](#_Toc473789246)

[Un widget peut en contenir un autre : 79](#_Toc473789247)

[Contenant et contenu : 79](#_Toc473789248)

[Créer une fenêtre contenant un bouton : 80](#_Toc473789249)

[Tout widget peut en stocker un autre : 81](#_Toc473789250)

[Hériter un widget : 82](#_Toc473789251)

[MaFenetre.h 83](#_Toc473789252)

[MaFenetre.cpp 83](#_Toc473789253)

# Premier projet

#include <iostream>

using namespace std;

*int* main()

{

cout << "Hello world!" << endl;

return 0;

}

On a :

* #include : directive du préprocesseur
* Using namespace std permet de spécifier l’espace de nom.

Quand on utilise Qt et Iostream, on ne sait pas où afficher le message, on le spécifie en expliquant ce qu’on utilise dans le using namespace.

# Les variables :

On peut utiliser la synthaxe C ou C++ :

Syntaxe d'initialisation d'une variable en C++

Syntaxe d'initialisation d'une variable en C++

On peut aussi utiliser la même syntaxe que dans le langage C (figure suivante).

Syntaxe d'initialisation d'une variable, héritée du C

Syntaxe d'initialisation d'une variable, héritée du C

### Strings :

ne.

#include <iostream>

#include <string>

using namespace std;

*int* main()

{

string nomUtilisateur("Albert Einstein");

return 0;

}

### Décaration :

On peut déclarer plusieurs variables de même type sur une même ligne :

*int* a(2),b(4),c(-1); //On déclare trois cases mémoires nommées a, b et c et qui contiennent respectivement les valeurs 2, 4 et -1

string prenom("Albert"), nom("Einstein"); //On déclare deux cases pouvant contenir des chaînes de caractères

Si on ne définit une variable sans l’initialiser :

#include <iostream>

#include <string>

using namespace std;

*int* main()

{

string nomJoueur;

*int* nombreJoueurs;

*bool* aGagne; //Le joueur a-t-il gagné ?

return 0;

}

## Afficher une variable :

#include <iostream>

using namespace std;

*int* main()

{

*int* ageUtilisateur(16);

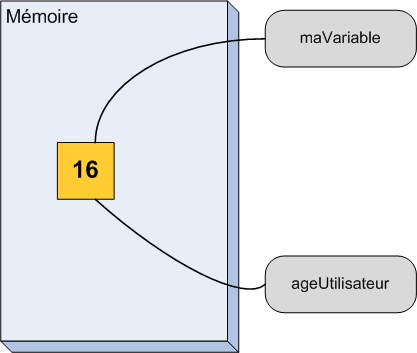
cout << "Votre age est : ";

cout << ageUtilisateur;

return 0;

}

## Les références :



Au niveau du code, on utilise une esperluette (**&**) pour déclarer une référence sur une variable. Voyons cela avec un petit exemple.

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

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

# Une vrai calculatrice

## Intéragir avec l’utiisateur :

**On utilise cin et plus cout et on inverse les chevrons :**

#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

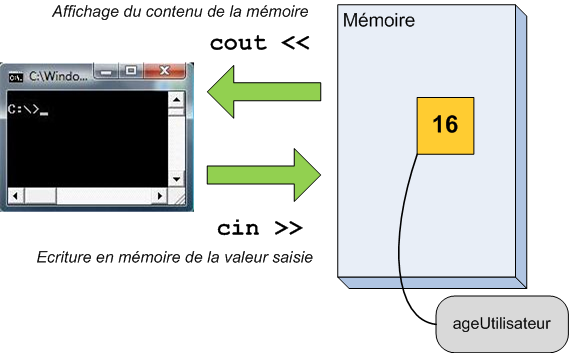
cin >> ageUtilisateur; //On fait entrer un nombre dans cette case

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

return 0;

}

### Moyen mémotechnique :



## getLine :

Quand onr rempli du texte dans la console, la lecture s’arrête aux espace ou retour à la ligne … Il faut donc indiquer de récupérer toute la ligne avec un getLine :

Il faut remplacer la ligne cin >> nomUtilisateur; par un getline().

#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;

}

Si on demande un cin avant un getLine, le code ne fonctionnera plus.

Pour pallier ce problème, il faut ajouter la ligne cin.ignore() après l'utilisation des chevrons.

#include <iostream>

#include <string>

using namespace std;

*int* main()

{

cout << "Combien vaut pi ?" << endl;

*double* piUtilisateur(-1.); //On crée une case mémoire pour stocker unnombre réel

cin >> piUtilisateur; //Et on remplit cette case avec ce qu'écritl'utilisateur

cin.ignore();

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 toute la ligne que l'utilisateur a écrit

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

return 0;

}

Avec cela, plus de souci.   
Quand on mélange l'utilisation des chevrons et de getline(), il faut toujours placer l'instruction cin.ignore() après la ligne cin>>a. C'est une règle à apprendre.

### Déclarer une constante :

Il est important de déclarer constant tout ce qui doit l’être pour éviter les erreurs.

string const motDePasse("wAsTZsaswQ"); //Le mot de passe secret

*double* const pi(3.14);

*unsigned* *int* const pointsDeVieMaximum(100); //Le nombre maximal de points de vie

### L’incrémentation :

**On incrémente en C++ en mettant AVANT le nom de variable les ++ :**

*int* nombreJoueur(4); //Il y a 4 joueurs dans la partie

++nombreJoueur;

//À partir d'ici, il y a 5 joueurs

## Cmath :

| **Nom de la fonction** | **Symbole** | **Fonction** | **Mini-exemple** |
| --- | --- | --- | --- |
| Racine carrée | sqrtx | sqrt() | resultat = sqrt(valeur); |
| Sinus | sin(x) | sin() | resultat = sin(valeur); |
| Cosinus | cos(x) | cos() | resultat = cos(valeur); |
| Tangente | tan(x) | tan() | resultat = tan(valeur); |
| Exponentielle | ex | exp() | resultat = exp(valeur); |
| Logarithme népérien | lnx | log() | resultat = log(valeur); |
| Logarithme en base 10 | log10x | log10() | resultat = log10(valeur); |
| Valeur absolue | |x| | fabs() | resultat = fabs(valeur); |
| Arrondi vers le bas | ⌊×⌋ | floor() | resultat = floor(valeur); |
| Arrondi vers le haut | ⌈×⌉ | ceil() | resultat = ceil(valeur); |

#include <iostream>

#include <cmath> //Ne pas oublier cette ligne

using namespace std;

*int* main()

{

*double* const nombre(16); //Le nombre dont on veut la racine

//Comme sa valeur ne changera pas on met 'const'

*double* resultat; //Une case mémoire pour stocker le résultat

resultat = sqrt(nombre); //On effectue le calcul !

cout << "La racine de " << nombre << " est " << resultat << endl;

return 0;

}

**Sin et cos utilisent les radians !**

### Pow :

:

#include <iostream>

#include <cmath> //Ne pas oublier !

using namespace std;

*int* main()

{

*double* a(0), b(0); //Déclaration des variables utiles

cout << "Bienvenue dans le programme de calcul de a^b !" << endl;

cout << "Donnez une valeur pour a : "; //On demande le premier nombre

cin >> a;

cout << "Donnez une valeur pour b : "; //On demande le deuxième nombre

cin >> b;

*double* const resultat(pow(a, b)); //On effectue l'opération

//On peut aussi écrire comme avant :

//double const resultat = pow(a,b);

//Souvenez-vous des deux formes possibles

//De l'initialisation d'une variable

cout << a << " ^ " << b << " = " << resultat << endl;

//On affiche le résultat

return 0;

}

# Les structures de contrôles

## Conditions :

### If :

#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;

}

### Switch :

#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;

}

### Boolean :

*bool* adulte(true);

if (adulte)

{

cout << "Vous etes un adulte !" << endl;

}

## Boucles :

### While :

*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;

}

### Do … while :

*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;

}

### For :

*int* main()

{

*int* compteur(0);

for (compteur = 0 ; compteur < 10 ; compteur++)

{

cout << compteur << endl;

}

return 0;

}

# Les fonctions

## Créer une fonction :

On peut utiliser void en type de retour come en Java.

*int* ajouteDeux(*int* nombreRecu)

{

*int* valeur(nombreRecu + 2);

//On crée une case en mémoire

//On prend le nombre reçu en argument, on lui ajoute 2

//Et on met tout cela dans la mémoire

return valeur;

//On indique que la valeur qui sort de la fonction

//Est la valeur de la variable 'valeur'

}

*int* addition(*int* a, *int* b)

{

return a+b;

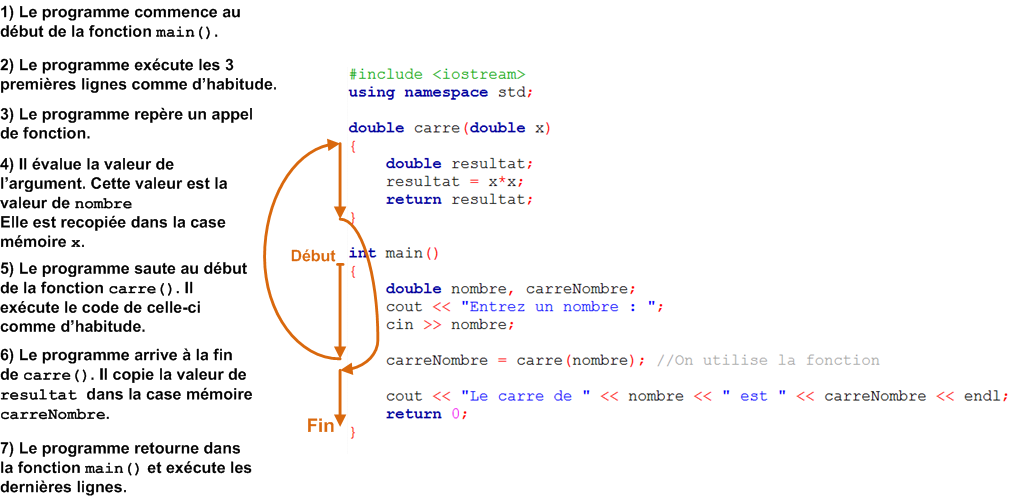
}

*double* multiplication(*double* a, *double* b, *double* c)

{

return a\*b\*c;

}



## Passage par valeur et par référence :

### Par valeur :

#include <iostream>

using namespace std;

*int* ajouteDeux(*int* a)

{

a+=2;

return a;

}

*int* main()

{

*int* nombre(4), resultat;

**resultat = ajouteDeux(nombre);**

cout << "Le nombre original vaut : " << nombre << endl;

cout << "Le resultat vaut : " << resultat << endl;

return 0;

}

### Par référence :

**int ajouteDeux(int& a) //Notez le petit & !!!**

{

a+=2;

return a;

}

**Exemple :**

***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;

}

a vaut 1.2 et b vaut 4.5

a vaut 4.5 et b vaut 1.2

Le passage par référence permet donc de modifier l’argument passé en paramètre, ce qui peut être risqué.

La solution est d'utiliser ce que l'on appelle un **passage par référence constante**. On évite la copie en utilisant une référence et l'on empêche la modification de l'argument en le déclarant constant.

*void* f1(string const& texte); //Pas de copie et pas de modification possible

{

}

**On met le & sur le const et plus sur le string.**

## Utiliser plusieurs fichiers :

Pour utiliser plusieurs fichiers il nous faut le fichier source (.cpp) ainsi que le fichier header (.h) qui ne contient que le **PROTOTYPE** de la fonction.

Le header et le fichier source doivent avoir le même nom. On inclut dans le fichier main, le fichier source avec ges guillements et non des accolades !

#include <iostream>

#include "math.h"

using namespace std;

*int* main()

{

*int* a(2),b(2);

cout << "Valeur de a : " << a << endl;

cout << "Valeur de b : " << b << endl;

b = ajouteDeux(a); //Appel de la fonction

cout << "Valeur de a : " << a << endl;

cout << "Valeur de b : " << b << endl;

return 0;

}

Pour le header, si on utilise des strings ou des maths par exemples, il faut inclure la bobliothèque.

#ifndef MATH\_H\_INCLUDED

#define MATH\_H\_INCLUDED

#include <string>

*void* afficherMessage(std::string message);

#endif // MATH\_H\_INCLUDED

## Valeurs par défaut pour les arguments :

**On peut déclarer les prototypes de fonction avant le main si on déclare nos fonctions APRES le main. Le cas normal est de créer les fonctions est de créer le main seulement après à cause de la lecture du code de haut en bas.**

Les valeurs par défaut sont dans le prototypes, donc ici avant le main. Si on utilisait cette fonctions dans un autre fichier, on auré donc modifié le .h et non le .cpp.

#include <iostream>

using namespace std;

// Prototype avec les valeurs par défaut

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

// Main

*int* main()

{

cout << nombreDeSecondes(1, 10, 25) << endl;

return 0;

}

// Définition de la fonction, SANS les valeurs par défaut

*int* nombreDeSecondes(*int* heures, *int* minutes, *int* secondes)

{

*int* total = 0;

total = heures \* 60 \* 60;

total += minutes \* 60;

total += secondes;

return total;

}

*Les paramètres facultatifs doivent obligatoirement se trouver à la fin* (à droite).

Ce code ne compilera donc pas :

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

//Erreur, les paramètres par défaut doivent être à droite

# Les tableaux :

## Les tableaux statiques :

### Décalrer :

Déclaration d'un tableau statique

#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;

}

Pour accéder aux éléments du tableau :

*int* const nombreMeilleursScores(5); //La taille du tableau

*int* meilleursScores[nombreMeilleursScores]; //Déclaration du tableau

meilleursScores[0] = 118218; //Remplissage de la première case

meilleursScores[1] = 100432; //Remplissage de la deuxième case

meilleursScores[2] = 87347; //Remplissage de la troisième case

meilleursScores[3] = 64523; //Remplissage de la quatrième case

meilleursScores[4] = 31415; //Remplissage de la cinquième case

### Les tableaux dans les fonctions :

*void* fonction(*double* tableau[])

{

//…

}

Exemple :

/\*

\* 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;

}

## Tableaux dynamiques :

### Déclarer :

Déclaration d'un vector

Par exemple, pour un tableau de 5 entiers, on écrit :

#include <iostream>

#include <vector> //Ne pas oublier !

using namespace std;

*int* main()

{

vector<*int*> tableau(5);

return 0;

}

On peut préremplir les champs :

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 »

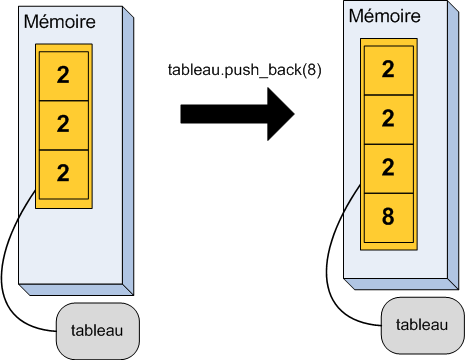
### Push\_back et pop\_back :

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

Ici on ajoute une nouvelle case contenant 8 :



On peut récupérer la taille du tableau ave **size** :

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

### Vector et fonction :

//Une fonction recevant un tableau d'entiers en argument

*void* fonction(vector<*int*> a)

{

//…

}

Pour reprendre ce qu’on a appris :

Si on doit copier les éléments du tableau SANS les modifier :

//Une fonction recevant un tableau d'entiers en argument

*void* fonction(vector<*int*> const& a)

{

//…

}

Si on découpe le programme en plusieurs fichiers, il faut ajouter vector à l’entête et ajouter std comme pour les strings :

#ifndef TABLEAU\_H\_INCLUDED

#define TABLEAU\_H\_INCLUDED

#include <vector>

*void* fonctionAvecTableau(std::vector<*int*>& tableau);

#endif // TABLEAU\_H\_INCLUDED

## Tableaux multidimensionnels :

### Déclaration :

*int* const tailleX(5);

*int* const tailleY(4);

*int* tableau[tailleX][tailleY];

### Aller plus loin :

On peut créer des tableaux à 2,3,4, … dimenssions.

Notez qu'il est aussi possible de créer des tableaux multi-dimensionnels de taille variable en utilisant les vectors. Pour une grille 2D d'entiers, on devra écrire :

vector<vector<*int*> > grille;

## Les strings comme tableau :

### Accéder aux lettres :

#include <iostream>

#include <string>

using namespace std;

*int* main()

{

string nomUtilisateur("Julien");

cout << "Vous etes " << nomUtilisateur << "." <<endl;

nomUtilisateur[0] = 'L'; //On modifie la première lettre

nomUtilisateur[2] = 'c'; //On modifie la troisième lettre

cout << "Ah non, vous etes " << nomUtilisateur << "!" << endl;

return 0;

}

Testons pour voir :

Vous etes Julien.

Ah non, vous etes Lucien!

### Les fonctions :

On peut également utiliser size() pour connaître le nombre de lettres et push\_back() pour ajouter des lettres à la fin. La encore, c'est comme avec vector.

string texte("Portez ce whisky au vieux juge blond qui fume."); //46 caractères

cout << "Cette phrase contient " << texte.size() << " lettres." << endl;

Mais contrairement aux tableaux, on peut ajouter *plusieurs lettres* d'un coup. Et on utilise le **+=**.

#include <iostream>

#include <string>

using namespace std;

*int* main()

{

string prenom("Albert");

string nom("Einstein");

string total; //Une chaîne vide

total += prenom; //On ajoute le prénom à la chaîne vide

total += " "; //Puis un espace

total += nom; //Et finalement le nom de famille

cout << "Vous vous appelez " << total << "." << endl;

return 0;

}

# Lire et modifier des fichiers

## Ecrire dans un fichier :

### Entête fstream :

On doit d’abords ajouter cette fonctionnalité : #include <fstream>.

D’abords on ouvre le chemin d’accès au fichier avec un flux. Pour ce faire on doit utiliser un objet et créer une variable de type ofstream.

#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;

}

On peut écrire un chemin relatif si on met le fichier au même endroit que le code, exemple : Fichier :fichier.txt.

string const nomFichier("C:/Nanoc/scores.txt");

ofstream monFlux(nomFichier.c\_str());

//Déclaration d'un flux permettant d'écrire dans un fichier.

**On doit toujours tester que tout s’est bien passé avec un if(monFlux) avant de mettre le code.**

#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 on utilise une variable pour stocker le chemin du flux, on doit rajouter.c\_ctr après.**

Pour écrire sans écraser ce qui est déjà écrit :

ofstream monFlux("C:/Nanoc/scores.txt", ios::app);.

## Lire un fichier :

Il y a trois manières différentes de lire un fichier :

1. Ligne par ligne, en utilisant getline() ;
2. Mot par mot, en utilisant les chevrons >> ;
3. Caractère par caractère, en utilisant 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

### Lire par caractère :

*char* a;

monFlux.get(a);

Comme pour cin, on doit faire .ignore() :

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

### Exemple de lecture :

#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;

}

## Astuces :

On peut gérer l’ouverture et la fermeture des flux :

*void* f()

{

ofstream flux; //Un flux sans fichier associé

flux.open("C:/Nanoc/data.txt"); //On ouvre le fichier C:/Nanoc/data.txt

//Utilisation du fichier

flux.close(); //On referme le fichier

//On ne peut plus écrire dans le fichier à partir d'ici

}

### Conaitre sa position dans le fichier :

| **Pour ifstream** | **Pour ofstream** |
| --- | --- |
| tellg() | tellp() |

Elles s'utilisent toutes les deux de la même manière.

ofstream fichier("C:/Nanoc/data.txt");

*int* position = fichier.tellp(); //On récupére la position

cout << "Nous nous situons au " << position << "eme caractere du fichier." << endl;

### Se déplacer :

flux.seekp(nombreCaracteres, position);

Les trois positions possibles sont :

* le début du fichier : ios::beg ;
* la fin du fichier : ios::end ;
* la position actuelle : ios::cur.

Si, par exemple, je souhaite me placer 10 caractères après le début du fichier, j'utilise flux.seekp(10, ios::beg);. Si je souhaite aller 20 caractères plus loin que l'endroit où se situe le curseur, j'utilise flux.seekp(20, ios::cur);.

### Connaître la taille du fichier :

#include <iostream>

#include <fstream>

using namespace std;

*int* main()

{

ifstream fichier("C:/Nanoc/meilleursScores.txt"); //On ouvre le fichier

fichier.seekg(0, ios::end); //On se déplace à la fin du fichier

*int* taille;

taille = fichier.tellg();

//On récupère la position qui correspond donc a la taille du fichier !

cout << "Taille du fichier : " << taille << " octets." << endl;

return 0;

}

## Générer un mot mystère :

#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;

}

# Les pointeurs

## Adresse d’une variable :

#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;

}

## Pointeurs :

**Un pointeur correspond à l’adresse d’une variable.**

### Déclarer un pointeur :

*double* \*pointeurA;

//Un pointeur qui peut contenir l'adresse d'un nombre à virgule

*unsigned* *int* \*pointeurB;

//Un pointeur qui peut contenir l'adresse d'un nombre entier positif

string \*pointeurC;

//Un pointeur qui peut contenir l'adresse d'une chaîne de caractères

vector<*int*> \*pointeurD;

//Un pointeur qui peut contenir l'adresse d'un tableau dynamique de nombres entiers

*int* const \*pointeurE;

//Un pointeur qui peut contenir l'adresse d'un nombre entier constant

Il faut TOUJOURS déclarer un pointeur avec l’adresse 0 entre parenthèse, pour éviter que celle-ci soit créée à n’importe quelle endroit et risquerait de modifier une variable windows.

Dans l’exemple suivant, ptr pointe sur ageUtilisateur.

*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;

}

### Accéder à la valeur de la variable pointée :

On utilise : \*

*int* main()

{

*int* ageUtilisateur(16);

*int* \*ptr(0);

ptr= &ageUtilisateur;

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

return 0;

}

**On crée un pointeur en mettant \* devzant le nom de variable et & devant la variable pointé. Pour accéder à la valeur de la variable pointé on utilise \* devant le pointeur.**

**Pour une variable int nombre :**

* **nombre permet d'accéder à la valeur de la variable ;**
* **&nombre permet d'accéder à l'adresse de la variable.**

**Sur un pointeur int \*pointeur :**

* **pointeur permet d'accéder à la valeur du pointeur, c'est-à-dire à l'*adresse de la variable pointée* ;**
* **\*pointeur permet d'accéder à la valeur de la variable pointée.**

## Allocation dynamique :

### Gestion automatique de la mémoire :

Le programme fait tout tout seul, il alloue la mémoire et l’initialise. Quand il a fini il la libère.

### Allouer de la mémoire (manuel) :

Pour demander d’allouer de l’espace mémoire on doit utiliser new :

*int* \*pointeur(0);

pointeur = new *int*;

*int* \*pointeur(0);

pointeur = new *int*;

\*pointeur = 2; //On accède à la case mémoire pour en modifier la valeur

### Libérer la mémoire :

Quand on libère de la mémoire, on doit indiquer que le pointeur ne pointe plus vers une varialbe !

*int* \*pointeur(0);

pointeur = new *int*;

delete pointeur; //On libère la case mémoire

pointeur = 0; //On indique que le pointeur ne pointe plus vers rien

Exemple complet :

#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;

}

## Quand utiliser des pointeurs ?

Il y a en réalité trois cas d'application :

* gérer soi-même le moment de la création et de la destruction des cases mémoire ;
* partager une variable dans plusieurs morceaux du code ;
* sélectionner une valeur parmi plusieurs options.

Exemple d’utilisation :

#include <iostream>

#include <string>

using namespace std;

*int* main()

{

string reponseA, reponseB, reponseC;

reponseA = "La peur des jeux de loterie";

reponseB = "La peur du noir";

reponseC = "La peur des vendredis treize";

cout << "Qu'est-ce que la 'kenophobie' ? " << endl; //On pose la question

cout << "A) " << reponseA << endl; //Et on affiche les trois propositions

cout << "B) " << reponseB << endl;

cout << "C) " << reponseC << endl;

*char* reponse;

cout << "Votre reponse (A,B ou C) : ";

cin >> reponse; //On récupère la réponse de l'utilisateur

string \*reponseUtilisateur(0); //Un pointeur qui pointera sur la réponse choisie

switch(reponse)

{

case 'A':

reponseUtilisateur = &reponseA; //On déplace le pointeur sur la réponse choisie

break;

case 'B':

reponseUtilisateur = &reponseB;

break;

case 'C':

reponseUtilisateur = &reponseC;

break;

}

//On peut alors utiliser le pointeur pour afficher la réponse choisie

cout << "Vous avez choisi la reponse : " << \*reponseUtilisateur << endl;

return 0;

}

# Les strings

## Créer et utiliser les strings :

### Concaténation :

*int* main()

{

string chaine1("Bonjour !");

string chaine2("Comment allez-vous ?");

string chaine3;

chaine3 = chaine1 + chaine2; // 3... 2... 1... Concaténatioooooon

cout << chaine3 << endl;

return 0;

}

On concatène avec le + comme en java.

### Comparer :

*int* main()

{

string chaine1("Bonjour !");

string chaine2("Comment allez-vous ?");

if (chaine1 == chaine2) // Faux

{

cout << "Les chaines sont identiques." << endl;

}

else

{

cout << "Les chaines sont differentes." << endl;

}

return 0;

}

## Méthodes des strings :

### Size :

*int* main()

{

string maChaine("Bonjour !");

cout << "Longueur de la chaine : " << maChaine.size();

return 0;

}

### Erase :

*int* main()

{

string chaine("Bonjour !");

chaine.erase();

cout << "La chaine contient : " << chaine << endl;

return 0;

}

### Substr :

Extrait une sous-chaîne.

*int* main()

{

string chaine("Bonjour !");

cout << chaine.substr(3) << endl;

return 0;

}

La méthode possède deux paramètres :

* Le premier obligatoire : à partir d’où on coupe.
* Le deuxième : combien de caractère on prend. Par défaut c’est npos qui prend tout le reste du string.

*int* main()

{

string chaine("Bonjour !");

cout << chaine.substr(3, 4) << endl;

return 0;

}

### C\_str :

Renvoie un pointeur vers le tableau de char.

On l’a utilisé pour créer un flux :

string const nomFichier("C:/Nanoc/scores.txt");

ofstream monFlux(nomFichier.c\_str());

# Les classes

## Créer une classe :

*class* Personnage

{

}; // N'oubliez pas le point-virgule à la fin !

**Par convention, le nom des classes commencent toujours par une majuscule (comparé aux variables qui commencent par une minuscule).**

### Les attributs :

*class* Personnage

{

*int* m\_vie;

*int* m\_mana;

string m\_nomArme;

*int* m\_degatsArme;

};

### Les méthodes :

*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;

};

## Droit d’accès et encapsulations :

Avec la POO il y a bien 2 étapes :

* La création de la classe
* L’utilisation de la classe

On a déjà crée la classe, on va l’utiliser :

*int* main()

{

Personnage david, goliath;

//Création de 2 objets de type Personnage : david et goliath

goliath.attaquer(david); //goliath attaque david

david.boirePotionDeVie(20); //david récupère 20 de vie en buvant une potion

goliath.attaquer(david); //goliath réattaque david

david.attaquer(goliath); //david contre-attaque... c'est assez clair non ?

goliath.changerArme("Double hache tranchante vénéneuse de la mort", 40);

goliath.attaquer(david);

return 0;

}

### Droits d’accès :

**Par défaut tous les éléments d’une classe sont private. On donc rajouter public si on le souhaite.**

*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;

};

**Encapsulation : TOUS LES ATTRIBUTS D’UNE CLASSE SONT PRIVATE.**

## Séparer prototype et définition :

Pour éviter la méthode bourin, on crée un nouveau .cpp et un .h

#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

Dans les .h, il est recommandé de ne jamais mettre la directive using namespace std; car cela pourrait avoir des effets néfastes, par la suite, lorsque vous utiliserez la classe.  
Par conséquent, il faut rajouter le préfixe std:: devant chaque string du .h. Sinon, le compilateur vous sortira une erreur du type string does not name a type.

Dans le .cpp on peut utiliser using namespace std.

Si on implémente lers méthodes en dehors de la classe, on doit préciser de quelle classe il s’agit :

*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()

{

if (m\_vie > 0) //Plus de 0 de vie ?

{

return true; //VRAI, il est vivant !

}

else

{

return false; //FAUX, il n'est plus vivant !

}

}

Pour tester le code : On ne peut pas encore le tester cas on n’a pas les getters et les setters.

#include <iostream>

#include "Personnage.h" //Ne pas oublier

using namespace std;

*int* main()

{

Personnage david, goliath;

//Création de 2 objets de type Personnage : david et goliath

goliath.attaquer(david); //goliath attaque david

david.boirePotionDeVie(20); //david récupère 20 de vie en buvant une potion

goliath.attaquer(david); //goliath réattaque david

david.attaquer(goliath); //david contre-attaque... c'est assez clair non ?

goliath.changerArme("Double hache tranchante vénéneuse de la mort", 40);

goliath.attaquer(david);

return 0;

}

## Constructeur et destructeurs :

### Le constructeur :

On doit aussi mettre le constructeur dans le header.

#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;

};

Personnage::Personnage()

{

m\_vie = 100;

m\_mana = 100;

m\_nomArme = "Épée rouillée";

m\_degatsArme = 10;

}

### Liste d’initialisation :

A utiliser le plus souvent, fait la même chose que le constructeur :

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 :

On crée un constructeur mais cette fois, avec une arme par défaut.

Personnage::Personnage(string nomArme, *int* degatsArme) : m\_vie(100), m\_mana(100),m\_nomArme(nomArme), m\_degatsArme(degatsArme)

{

}

### Constructeur de copie :

A la création de la classe, on a un constructeur par défaut, mais aussi un constructeur de copie. On peut par exemple copier un objet dans un autre.

Personnage goliath("Épée aiguisée", 20); //On crée goliath en utilisant un constructeur normal

Personnage david(goliath); //On crée david en copiant tous les attributs de goliath

Si toute fois, vous désirez changer le comportement du constructeur de copie, il faut simplement le déclarer dans votre classe de la manière suivante:

Personnage(Personnage const& autre);

et de définir son implémentation comme suit:

Personnage::Personnage(Personnage const& autre): m\_vie(autre.m\_vie), m\_mana(autre.m\_mana), m\_nomArme(autre.m\_nomArme), m\_degatsArme(autre.m\_degatsArme)

{

}

### Destruceur :

* Un destructeur est une méthode qui commence par un tilde (~) suivi du nom de la classe.
* Un destructeur ne renvoie aucune valeur, pas même void (comme le constructeur).
* Et, nouveauté : le destructeur ne peut prendre aucun paramètre. Il y a donc toujours un seul destructeur, il ne peut pas être surchargé.

Dans Personnage.h, le prototype du destructeur sera donc :

~Personnage();

Dans Personnage.cpp, l'implémentation sera :

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. \*/

}

## Méthodes constantes :

Méthode avec « const » à la fin. Elle est en lecture seule, signifie qu’elle ne modifie pas les attributs de la classe.

//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

{

}

Exemple :

*bool* Personnage::estVivant() const

{

return m\_vie > 0;

}

## Associer des classes entre elles :

##### Arme.h

#ifndef DEF\_ARME

#define DEF\_ARME

#include <iostream>

#include <string>

*class* Arme

{

public:

Arme();

Arme(std::string nom, *int* degats);

*void* changer(std::string nom, *int* degats);

*void* afficher() const;

private:

std::string m\_nom;

*int* m\_degats;

};

#endif

Arme.cpp

#include "Arme.h"

using namespace std;

Arme::Arme() : m\_nom("Épée rouillée"), m\_degats(10)

{

}

Arme::Arme(string nom, *int* degats) : m\_nom(nom), m\_degats(degats)

{

}

*void* Arme::changer(string nom, *int* degats)

{

m\_nom = nom;

m\_degats = degats;

}

*void* Arme::afficher() const

{

cout << "Arme : " << m\_nom << " (Dégâts : " << m\_degats << ")" << endl;

}

Voici mon nouveau Personnage.h :

#ifndef DEF\_PERSONNAGE

#define DEF\_PERSONNAGE

#include <iostream>

#include <string>

#include "Arme.h" //Ne PAS oublier d'inclure Arme.h pour en avoir la définition

*class* Personnage

{

public:

Personnage();

Personnage(std::string nomArme, *int* degatsArme);

~Personnage();

*void* recevoirDegats(*int* nbDegats);

*void* attaquer(Personnage &cible);

*void* boirePotionDeVie(*int* quantitePotion);

*void* changerArme(std::string nomNouvelleArme, *int* degatsNouvelleArme);

*bool* estVivant() const;

private:

*int* m\_vie;

*int* m\_mana;

Arme m\_arme; //Notre Personnage possède une Arme

};

#endif

##### Personnage.cpp

Nous n'avons besoin de changer que les méthodes qui utilisent l'arme, pour les adapter.  
On commence par les constructeurs :

Personnage::Personnage() : m\_vie(100), m\_mana(100)

{

}

Personnage::Personnage(string nomArme, *int* degatsArme) : m\_vie(100), m\_mana(100),m\_arme(nomArme, degatsArme)

{

}

On conseille généralement de rajouter le mot-clé const aux accesseurs pour en faire des méthodes constantes, puisqu'elles ne modifient pas l'objet.

*int* Arme::getDegats() const

{

return m\_degats;

}

N'oubliez pas de mettre à jour Arme.h avec le prototype, qui sera le suivant :

*int* getDegats() const;

Vous pouvez maintenant retourner dans Personnage.cpp et écrire :

*void* Personnage::attaquer(Personnage &cible)

{

cible.recevoirDegats(m\_arme.getDegats());

}

getDegats renvoie le nombre de dégâts, qu'on envoie à la méthode recevoirDegats de la cible. Pfiou !

Le reste des méthodes n'a pas besoin de changer, à part changerArme de la classe Personnage :

*void* Personnage::changerArme(string nomNouvelleArme, *int* degatsNouvelleArme)

{

m\_arme.changer(nomNouvelleArme, degatsNouvelleArme);

}

#### Prototype et include

On rajoute le prototype, tout bête, dans le .h :

*void* afficherEtat() const;

#### Implémentation

Implémentons ensuite la méthode. C'est simple, on a simplement à faire des cout. Grâce aux attributs, on peut faire apparaître toutes les informations relatives au personnage :

*void* Personnage::afficherEtat() const

{

cout << "Vie : " << m\_vie << endl;

cout << "Mana : " << m\_mana << endl;

m\_arme.afficher();

}

Comme vous pouvez le voir, les informations sur l'arme sont demandées à l'objet m\_armevia sa méthode afficher(). Encore une fois, les objets communiquent entre eux pour récupérer les informations dont ils ont besoin.

#### Appel de afficherEtat dans le main()

Bien, tout cela c'est bien beau mais, tant qu'on n'appelle pas la méthode, elle ne sert à rien   
Je vous propose donc de compléter le main() et de rajouter à la fin les appels de méthode :

*int* main()

{

//Création des personnages

Personnage david, goliath("Épée aiguisée", 20);

//Au combat !

goliath.attaquer(david);

david.boirePotionDeVie(20);

goliath.attaquer(david);

david.attaquer(goliath);

goliath.changerArme("Double hache tranchante vénéneuse de la mort", 40);

goliath.attaquer(david);

//Temps mort ! Voyons voir la vie de chacun…

cout << "David" << endl;

david.afficherEtat();

cout << endl << "Goliath" << endl;

goliath.afficherEtat();

return 0;

}

# Surcharge d’opérateurs

## Préparatifs :

On a deux durées en heure et minutes et on veut les additionner sans forcément faire de classe additionneur, juste en faisant :

Duree resultat, duree1, duree2;

resultat = duree1 + duree2;

On prendra comme exemple cette durée :

##### Duree.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

##### Duree.cpp

#include "Duree.h"

Duree::Duree(*int* heures, *int* minutes, *int* secondes) : m\_heures(heures), m\_minutes(minutes), m\_secondes(secondes)

{

}

##### main.cpp ?

*int* main()

{

Duree duree1(0, 10, 28), duree2(0, 15, 2);

return 0;

}

## Les opérateurs de comparaison :

Pour être capables d'utiliser le symbole « == » entre deux objets, vous devez créer une fonction ayant précisément pour nom operator== et dotée du prototype :

*bool* operator==(Objet const& a, Objet const& b);

Même si l'on parle de classe, ceci n'est pas une méthode. C'est une fonction normale située à l'extérieur de toute classe.

Dès le moment où vous avez créé cette fonction operator==, vous pouvez comparer deux objets de type Duree :

if(duree1 == duree2)

{

std::cout << "Les deux durees sont egales !" << std::endl;

}

### Implémenter l’opérateur :

*bool* operator==(Duree const& a, Duree const& b)

{

//Teste si a.m\_heure == b.m\_heure etc.

if (a.m\_heures == b.m\_heures && a.m\_minutes == b.m\_minutes && a.m\_secondes == b.m\_secondes)

return true;

else

return false;

}

On ne peut pas avoir accès aux variables parce qu’elles sont private donc soit :

* Vous créez des accesseurs comme on l'a vu (ces fameuses méthodes getHeures(), getMinutes(), …). Cela marche bien mais c'est un peu ennuyeux à écrire.
* Vous utilisez le concept d'amitié, que nous verrons dans un prochain chapitre.
* Ou bien vous utilisez la technique que je vais vous montrer.

*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;

}

ou (mieux) en version courte:

*bool* Duree::estEgal(Duree const& b) const

{

return (m\_heures == b.m\_heures && m\_minutes == b.m\_minutes && m\_secondes == b.m\_secondes); //Teste si a.m\_heure == b.m\_heure etc.

}

Et on utilise cette méthode dans l'opérateur de comparaison :

*bool* operator==(Duree const& a, Duree const& b)

{

return a.estEgal(b);

}

Dans le main(), on peut faire un simple test de comparaison pour vérifier que l'on a fait les choses correctement :

*int* main()

{

Duree duree1(0, 10, 28), duree2(0, 10, 28);

if (duree1 == duree2)

cout << "Les durees sont identiques";

else

cout << "Les durees sont differentes";

return 0;

}

### L’opérateur != :

On utilise == définit précédement :

*bool* operator!=(Duree const& a, Duree const& b)

{

if(a == b) //On utilise l'opérateur == qu'on a défini précédemment !

return false; //S'ils sont égaux, alors ils ne sont pas différents

else

return true; //Et s'ils ne sont pas égaux, c'est qu'ils sont différents

}

ou en version courte:

*bool* operator!=(Duree const& a, Duree const& b)

{

return !(a==b); //On utilise l'opérateur == qu'on a défini précédemment !

}

### L’opérateur < :

Voici mon implémentation pour l'opérateur < (« est strictement inférieur à ») :

*bool* operator<(Duree const &a, Duree const& b)

{

return a.estPlusPetitQue(b);

}

Et la méthode estPlusPetitQue() de la classe Duree :

*bool* Duree::estPlusPetitQue(Duree const& b) const

{

if (m\_heures < b.m\_heures) // Si les heures sont différentes

return true;

else if (m\_heures == b.m\_heures && m\_minutes < b.m\_minutes) //Si elles sont égales, on compare les minutes

return true;

else if (m\_heures == b.m\_heures && m\_minutes == b.m\_minutes && m\_secondes < b.m\_secondes) // Et si elles sont aussi égales, on compare les secondes

return true;

else //Si tout est égal, alors l'objet n'est pas plus petit que b

return false;

}

## Opérateur arithmétiques :

### L’addition :

Duree operator+(Duree const& a, Duree const& b)

{

Duree resultat;

resultat = a.calculeAddition(b); //Utilise une méthode de Duree pour effectuer l'addition

return resultat;

}

### += :

*void* Duree::operator+=(const Duree& a)

{

//1 : ajout des secondes

m\_secondes += a.m\_secondes;

//Si le nombre de secondes dépasse 60, on rajoute des minutes

//Et on met un nombre de secondes inférieur à 60

m\_minutes += m\_secondes / 60;

m\_secondes %= 60;

//2 : ajout des minutes

m\_minutes += a.m\_minutes;

//Si le nombre de minutes dépasse 60, on rajoute des heures

//Et on met un nombre de minutes inférieur à 60

m\_heures += m\_minutes / 60;

m\_minutes %= 60;

//3 : ajout des heures

m\_heures += a.m\_heures;

}

## Les opérateurs de flux :

<< utilisé avec cout et >> avec sin sont aussi des opérateur.

cout << "Coucou !";

cin >> variable;

Figurez-vous justement que << et >> sont des opérateurs. Le code ci-dessus revient donc à écrire :

operator<<(cout, "Coucou !");

operator>>(cin, variable);

On veut pouvoir faire :

Duree chrono(0, 2, 30);

cout << chrono;

//Erreur : il n'existe pas de fonction operator<<(cout, Duree &duree)

### Opérateur << :

ostream &operator<<( ostream &flux, Duree const& duree)

{

duree.afficher(flux) ;

return flux;

}

Et rajoutez dans le fichier Duree.h une méthode afficher() à la classe Duree :

*void* afficher(std::ostream &flux) const;

Voici l'implémentation de la méthode dans Duree.cpp :

*void* Duree::afficher(ostream &flux) const

{

flux << m\_heures << "h" << m\_minutes << "m" << m\_secondes << "s";

}

# TP, ZFraction

## Préparatifs :

Correction dans le cours c++ : ZFraction.

# Classes et pointeurs

## Pointeur d’une classe vers une autre classe :

Pour le moment on a :

*class* Personnage

{

public:

//Quelques méthodes…

private:

Arme m\_arme; // L'Arme est "contenue" dans le Personnage

//…

};

Un autre moyen serait d’utiliser un pointeur pour que l’arme ne soit plus DANS le personnage :

*class* Personnage

{

public:

//Quelques méthodes…

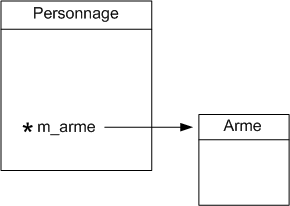
private:

Arme \*m\_arme;

//L'Arme est un pointeur, l'objet n'est plus contenu dans le Personnage

//…

};



## Gestion de l’allocation dynamique :

On part du code au dessus. L’arme est un pointeur.

Notre Arme étant un pointeur, il va falloir le créer par le biais d'une allocation dynamique avec new. Sinon, l'objet ne se créera pas tout seul.

Etant donné qu’il s’agit d’une allocation dynamique, on doit allouer la mémoire pour l’arme dans le constructeur.

Personnage::Personnage() : m\_arme(0), m\_vie(100), m\_mana(100)

{

m\_arme = new Arme();

}

Si vous vous souvenez bien, on avait aussi fait un second constructeur pour ceux qui voulaient que le Personnage commence avec une arme plus puissante dès le début. Il faut là aussi y faire une allocation dynamique :

Personnage::Personnage(string nomArme, *int* degatsArme) : m\_arme(0), m\_vie(100), m\_mana(100)

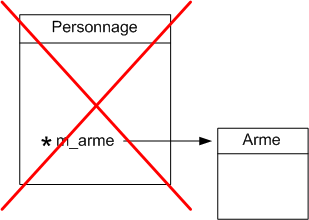
{

m\_arme = new Arme(nomArme, degatsArme);

}

### Désallouer de la mémoire pour l’objet :

Quand on supprime le personnage, l’arme n’est pas supprimée pour autant. C’est ce qu’on appelle une **fuite de mémoire**.



Pour résoudre le problème on doit faire un delete sur l’arme dans le destructeur :

Personnage::~Personnage()

{

delete m\_arme;

}

Comme on a un pointeur, on doit changer les autres méthodes :

*void* Personnage::attaquer(Personnage &cible)

{

cible.recevoirDegats(m\_arme.getDegats());

}

… devient :

*void* Personnage::attaquer(Personnage &cible)

{

cible.recevoirDegats(m\_arme->getDegats());

}

## Le pointeur this :

This est un pointeur vers l’objet actuel, donc \*this retourne l’objet lui-même.

Personnage\* Personnage::getAdresse() const

{

return this;

}

Nous l'avons en fait déjà rencontré une fois, lors de la surcharge de l'opérateur +=. Souvenez-vous, notre opérateur ressemblait à ceci :

Duree& Duree::operator+=(const Duree &duree2)

{

//Des calculs compliqués…

return \*this;

}

## Constructeur de copie :

C’est une surcharge du constructeur.

*int* main()

{

Personnage goliath("Epée aiguisée", 20);

Personnage david(goliath);

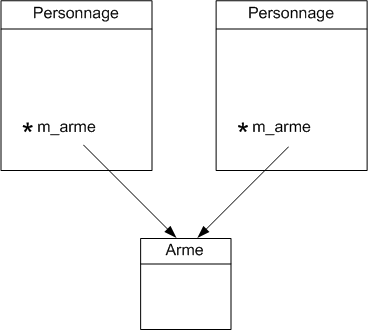
//On crée david à partir de goliath. david sera une « copie » de goliath.

return 0;

}

Mais quand on envoie un objet à une fonction, s’il n’est pas référencé ou pointé, est aussi copié.

Du coup on se retrouve avec un objet et sa copie, ils pointent tous les 2 vers la même arme :



### Création du constructeur de copie :

C’est un constructeur qui prend pour paramètre, une référence constante vers un objet du même type.

*class* Personnage

{

public:

Personnage();

Personnage(Personnage const& personnageACopier);

//Le prototype du constructeur de copie

Personnage(std::string nomArme, *int* degatsArme);

~Personnage();

/\*

… plein d'autres méthodes qui ne nous intéressent pas ici

\*/

private:

*int* m\_vie;

*int* m\_mana;

Arme \*m\_arme;

};

L’implémentation :

Sans les pointeurs :

Personnage::Personnage(Personnage const& personnageACopier)

: m\_vie(personnageACopier.m\_vie), m\_mana(personnageACopier.m\_mana), m\_arme(0)

{

}

Il faut donc appeler le constructeur de copie de Arme, en passant en paramètre l'objet à copier. Vous pourriez penser qu'il faut faire ceci :

m\_arme = new Arme(personnageACopier.m\_arme);

Presque ! Sauf que m\_arme est un pointeur et le prototype du constructeur de copie est :

Arme(Arme const& arme);

… ce qui veut dire qu'il faut envoyer l'objet lui-même et pas son adresse. Vous vous souvenez de la manière d'obtenir l'objet (ou la variable) à partir de son adresse ? On utilise l'étoile \* !  
Cela donne au final :

m\_arme = new Arme(\*(personnageACopier.m\_arme));

Au final :

Personnage::Personnage(Personnage const& personnageACopier)

: m\_vie(personnageACopier.m\_vie), m\_mana(personnageACopier.m\_mana), m\_arme(0)

{

m\_arme = new Arme(\*(personnageACopier.m\_arme));

}

### La surcharge d’affectation :

Operator=.

Personnage& Personnage::operator=(Personnage const& personnageACopier)

{

if(this != &personnageACopier)

//On vérifie que l'objet n'est pas le même que celui reçu en argument

{

m\_vie = personnageACopier.m\_vie; //On copie tous les champs

m\_mana = personnageACopier.m\_mana;

delete m\_arme;

m\_arme = new Arme(\*(personnageACopier.m\_arme));

}

return \*this; //On renvoie l'objet lui-même

}

Il faut penser à supprimer l’ancienne arme avant d’ajouter la nouvelle.

Il y a une chose importante à retenir au sujet de cet opérateur : il va toujours de pair avec le constructeur de copie.

Si l'on a besoin d'écrire un constructeur de copie, alors il faut aussi obligatoirement écrire une surcharge de operator=.

C'est une règle très importante à respecter. Vous risquez de graves problèmes de pointeurs si vous ne la respectez pas.

# L’héritage

## Héritage simple :

On part de la classe en simplifié :

#ifndef DEF\_PERSONNAGE

#define DEF\_PERSONNAGE

#include <iostream>

#include <string>

*class* Personnage

{

public:

Personnage();

*void* recevoirDegats(*int* degats);

*void* coupDePoing(Personnage &cible) const;

private:

*int* m\_vie;

std::string m\_nom;

};

#endif

#include "Personnage.h"

using namespace std;

Personnage::Personnage() : m\_vie(100), m\_nom("Jack")

{

}

*void* Personnage::recevoirDegats(*int* degats)

{

m\_vie -= degats;

}

*void* Personnage::coupDePoing(Personnage &cible) const

{

cible.recevoirDegats(10);

}

Pour l’héritage :

#ifndef DEF\_GUERRIER

#define DEF\_GUERRIER

#include <iostream>

#include <string>

#include "Personnage.h"

//Ne pas oublier d'inclure Personnage.h pour pouvoir en hériter !

*class* Guerrier : public Personnage

//Signifie : créer une classe Guerrier qui hérite de la classe Personnage

{

};

#endif

## La dérivation de type :

On peut faire :

Personnage \*monPersonnage(0);

Guerrier \*monGuerrier = new Guerrier();

monPersonnage = monGuerrier; // Mais… mais… Ça marche !?

## Héritage et constructeur :

Quand on crée une classe fille, c’est d’abords le constructeur parent qui est lancé, ensuite celui de la classe fille.

### Appeler le constructeur de la classe mère :

Pour appeler le constructeur de Personnage en premier, il faut y faire appel depuis le constructeur de Magicien. C'est dans un cas comme cela qu'il est indispensable de se servir de la liste d'initialisation (vous savez, tout ce qui suit le symbole deux-points dans l'implémentation).

Magicien::Magicien() : Personnage(), m\_mana(100)

{

}

Le premier élément de la liste d'initialisation indique de faire appel en premier lieu au constructeur de la classe parente Personnage. Puis on réalise les initialisations propres au Magicien (comme l'initialisation du mana à 100).

Il appelle par le constructeur parent sans paramètre.

### Transmission des paramètres :

Si Personnage prend un nom en paramètre :

Magicien::Magicien(string nom) : Personnage(nom), m\_mana(100)

{

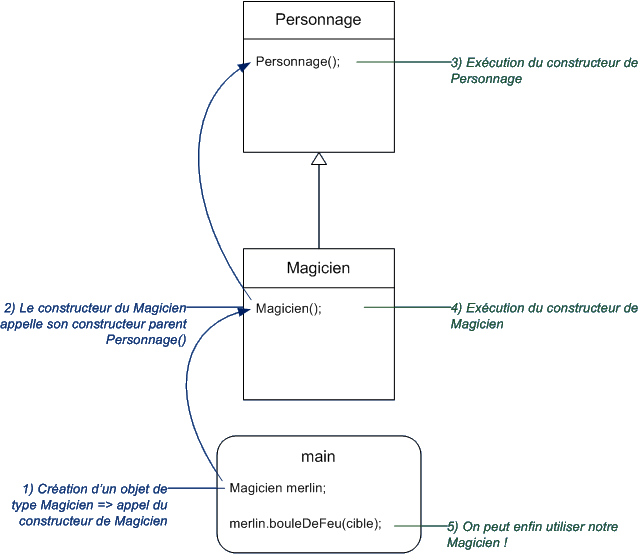
}

Bien entendu, si on veut que cela marche, il faut aussi surcharger le constructeur de Personnage pour qu'il accepte un paramètre string !

Personnage::Personnage(string nom) : m\_vie(100), m\_nom(nom)

{

}



## Portée protected :

Pour le moment on a :

* public : les éléments qui suivent sont accessibles depuis l'extérieur de la classe ;
* private : les éléments qui suivent ne sont pas accessibles depuis l'extérieur de la classe.

Protected : Voici sa signification : les éléments qui suivent protected ne sont pas accessibles depuis l'extérieur de la classe, sauf si c'est une classe fille.

## Le masquage :

Si on a :

*class* Personnage

{

public:

Personnage();

Personnage(std::string nom);

*void* recevoirDegats(*int* degats);

*void* coupDePoing(Personnage& cible) const;

*void* sePresenter() const;

protected:

*int* m\_vie;

std::string m\_nom;

};

Remarquez le const qui indique que le personnage ne sera pas modifié quand il se présentera. Vous en avez maintenant l'habitude, mais j'aime bien vous rafraîchir la mémoire.

Et dans le fichier .cpp :

*void* Personnage::sePresenter() const

{

cout << "Bonjour, je m'appelle " << m\_nom << "." << endl;

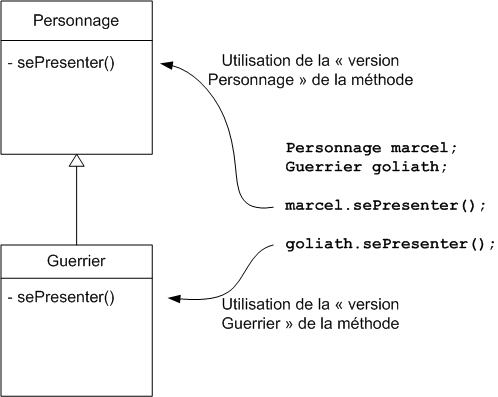
cout << "J'ai encore " << m\_vie << " points de vie." << endl;

}

Donc les classes filles pourront aussi se présenter.

### La masquage :

Si deux classes ont les mêmes méthodes, on executera l’une ou l’aitre en fonction de la nature de la classe. Exemple :



### Le démasquage :

Pour économiser du code, on peut démasquer et utiliser aussi la méthode de la classe mère.

*void* Guerrier::sePresenter() const

{

Personnage::sePresenter();

cout << "Je suis un Guerrier redoutable." << endl;

}

**:: s’appelle : opérateur de résolution de portée.**

# Le polymorphisme

## La résolution des liens :

Exemple d’un garagiste qui sait réparer voitures et motos :

*class* Vehicule

{

public:

*void* affiche() const; //Affiche une description du Vehicule

protected:

*int* m\_prix; //Chaque véhicule a un prix

};

*class* Voiture : public Vehicule //Une Voiture EST UN Vehicule

{

public:

*void* affiche() const;

private:

*int* m\_portes; //Le nombre de portes de la voiture

};

*class* Moto : public Vehicule //Une Moto EST UN Vehicule

{

public:

*void* affiche() const;

private:

*double* m\_vitesse; //La vitesse maximale de la moto

};

Rappel du masquage :

*int* main()

{

Vehicule v;

v.affiche(); //Affiche "Ceci est un vehicule."

Moto m;

m.affiche(); //Affiche "Ceci est une moto."

return 0;

}

### Résolution statique des liens :

*void* presenter(Vehicule v) //Présente le véhicule passé en argument

{

v.affiche();

}

*int* main()

{

Vehicule v;

presenter(v);

Moto m;

presenter(m);

return 0;

}

Le code n’est plus bon, moto est un véhicule maintenant.

### Résolution dynamique des liens :

Il faut deux choses pour la réaliser :

* Utiliser un pointeur ou une référence
* Utiliser une méthode virtuelle.

## Les fonctions virtuelles :

### Déclarer une méthode virtuelle :

*class* Vehicule

{

public:

virtual *void* affiche() const; //Affiche une description du Vehicule

protected:

*int* m\_prix; //Chaque véhicule a un prix

};

*class* Voiture: public Vehicule //Une Voiture EST UN Vehicule

{

public:

virtual *void* affiche() const;

private:

*int* m\_portes; //Le nombre de portes de la voiture

};

*class* Moto : public Vehicule //Une Moto EST UN Vehicule

{

public:

virtual *void* affiche() const;

private:

*double* m\_vitesse; //La vitesse maximale de la moto

};

Virtual n’est pas obligatoire sur les classes filles aec l’héritage, c’est une façon de se rappeler qu’elles le sont.

**Virtual ne se met QUE dans le .h**

### Utiliser une référence :

*void* presenter(Vehicule const& v) //Présente le véhicule passé en argument

{

v.affiche();

}

*int* main() //Rien n'a changé dans le main()

{

Vehicule v;

presenter(v);

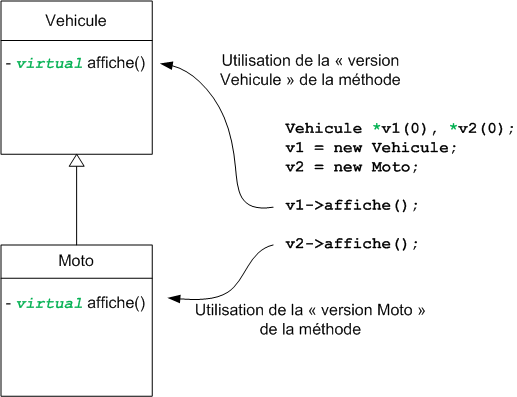
Moto m;

presenter(m);

return 0;

}

On aurait obtenu le même résultat avec des pointeurs :



## Les méthodes spéciales :

Le constructeur et le destructeur n’est pas hérité, tout le reste peut être sujet au polymorphisme.

En revant le destructeur doit toujours être virtuel si on utilise le polymorphisme !

*class* Vehicule

{

public:

Vehicule(*int* prix); //Construit un véhicule d'un certain prix

virtual *void* affiche() const;

virtual ~Vehicule(); //Remarquez le 'virtual' ici

protected:

*int* m\_prix;

};

*class* Voiture: public Vehicule

{

public:

Voiture(*int* prix, *int* portes);

//Construit une voiture dont on fournit le prix et le nombre de portes

virtual *void* affiche() const;

virtual ~Voiture();

private:

*int* m\_portes;

};

*class* Moto : public Vehicule

{

public:

Moto(*int* prix, *double* vitesseMax);

//Construit une moto d'un prix donné et ayant une certaine vitesse maximale

virtual *void* affiche() const;

virtual ~Moto();

private:

*double* m\_vitesse;

};

Il faut bien sûr également compléter le fichier source :

Vehicule::Vehicule(*int* prix)

:m\_prix(prix)

{}

*void* Vehicule::affiche() const

//J'en profite pour modifier un peu les fonctions d'affichage

{

cout << "Ceci est un vehicule coutant " << m\_prix << " euros." << endl;

}

Vehicule::~Vehicule() //Même si le destructeur ne fait rien, on doit le mettre !

{}

Voiture::Voiture(*int* prix, *int* portes)

:Vehicule(prix), m\_portes(portes)

{}

*void* Voiture::affiche() const

{

cout << "Ceci est une voiture avec " << m\_portes << " portes et coutant " << m\_prix << " euros." << endl;

}

Voiture::~Voiture()

{}

Moto::Moto(*int* prix, *double* vitesseMax)

:Vehicule(prix), m\_vitesse(vitesseMax)

{}

*void* Moto::affiche() const

{

cout << "Ceci est une moto allant a " << m\_vitesse << " km/h et coutant " << m\_prix << " euros." << endl;

}

Moto::~Moto()

{}

## Les collections hétérogènes :

Pour gérer notre garage, on utilisera un tableau dynamique :

vector<Voiture> listeVoitures;

vector<Moto> listeMotos;

Mais c’est pas pratique pour l’évolution du code .. ça fairait beaucoup de vector.

### Retour des pointeurs :

*int* main()

{

vector<Vehicule\*> listeVehicules;

return 0;

}

C’est une collection hétérogène puisqu’elle posède des types différents.

*int* main()

{

vector<Vehicule\*> listeVehicules;

listeVehicules.push\_back(new Voiture(15000, 5));

//J'ajoute à ma collection de véhicules une voiture

//Valant 15000 euros et ayant 5 portes

listeVehicules.push\_back(new Voiture(12000, 3)); //…

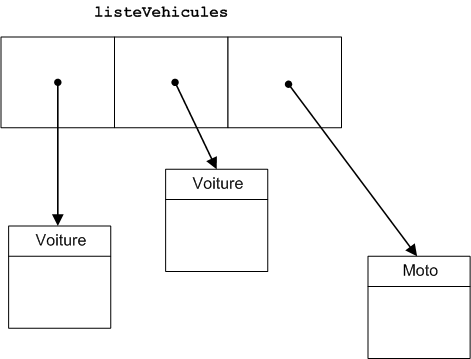
listeVehicules.push\_back(new Moto(2000, 212.5));

//Une moto à 2000 euros allant à 212.5 km/h

//On utilise les voitures et les motos

return 0;

}



Il ne faut pas oublier de vider la mémoire quand on fait une création dynamique avec new :

*int* main()

{

vector<Vehicule\*> listeVehicules;

listeVehicules.push\_back(new Voiture(15000, 5));

listeVehicules.push\_back(new Voiture(12000, 3));

listeVehicules.push\_back(new Moto(2000, 212.5));

//On utilise les voitures et les motos

for(*int* i(0); i<listeVehicules.size(); ++i)

{

delete listeVehicules[i]; //On libère la i-ème case mémoire allouée

listeVehicules[i] = 0; //On met le pointeur à 0 pour éviter les soucis

}

return 0;

}

On peut maintenant les utiliser :

*int* main()

{

vector<Vehicule\*> listeVehicules;

listeVehicules.push\_back(new Voiture(15000, 5));

listeVehicules.push\_back(new Voiture(12000, 3));

listeVehicules.push\_back(new Moto(2000, 212.5));

listeVehicules[0]->affiche();

//On affiche les informations de la première voiture

listeVehicules[2]->affiche();

//Et celles de la moto

for(*int* i(0); i<listeVehicules.size(); ++i)

{

delete listeVehicules[i]; //On libère la i-ème case mémoire allouée

listeVehicules[i] = 0; //On met le pointeur à 0 pour éviter les soucis

}

return 0;

}

## Les fonctions virtuelles pures :

Ajout du nombre de roue des véhicules :

*class* Vehicule

{

public:

Vehicule(*int* prix);

virtual *void* affiche() const;

virtual *int* nbrRoues() const; //Affiche le nombre de roues du véhicule

virtual ~Vehicule();

protected:

*int* m\_prix;

};

*class* Voiture : public Vehicule

{

public:

Voiture(*int* prix, *int* portes);

virtual *void* affiche() const;

virtual *int* nbrRoues() const; //Affiche le nombre de roues de la voiture

virtual ~Voiture();

private:

*int* m\_portes;

};

Que mettre dans nbrRoues de Véhicule ? C’est le rôle des fonctions virtuelles pures : méthodes qui existe dans les classes filles mais pas dans la classe mère.

Dans le .h :

*class* Vehicule

{

public:

Vehicule(*int* prix);

virtual *void* affiche() const;

virtual *int* nbrRoues() const = 0; //Affiche le nombre de roues du véhicule

virtual ~Vehicule();

protected:

*int* m\_prix;

};

### Les classes abstraites :

**Une classe qui possède au moins une méthode virtuelle pure est une classe abstraite.**

Même si on ne peut pas instancier une classe abstraite, le code suivant est autorisé :

*int* main()

{

Vehicule\* ptr(0); //Un pointeur sur un véhicule

Voiture caisse(20000,5);

//On crée une voiture

//Ceci est autorisé puisque toutes les fonctions ont un corps

ptr = &caisse; //On fait pointer le pointeur sur la voiture

cout << ptr->nbrRoues() << endl;

//Dans la classe fille, nbrRoues() existe, ceci est donc autorisé

return 0;

}

On crée un pointeur sur un véhicule vide, puis on redirige le pointeur sur une voiture.

* une méthode virtuelle *peut* être redéfinie dans une classe fille ;
* une méthode virtuelle pure *doit* être redéfinie dans une classe fille.

# Eléments statiques et amitié

## Les méthodes statiques :

Elles appartiennent à la classe mais aps aux objets instancié. Elles sont juste « rangées » dans la classe et n’ont aps accès aux attributs.

Exemple :

*class* MaClasse

{

public:

MaClasse();

static *void* maMethode();

};

Son implémentation dans le .cpp ne possède pas en revanche de mot-clé static :

*void* MaClasse::maMethode() //Ne pas remettre 'static' dans l'implémentation

{

cout << "Bonjour !" << endl;

}

Ensuite, dans le main(), la méthode statique s'appelle comme ceci :

*int* main()

{

MaClasse::maMethode();

return 0;

}

Donc pas besoin de crée d’objet, on utilise directementla méthode avec la résolution de portée ::.

## Les attributs statiques :

Ils peuvent être accessible à l’extérieur.

*class* MaClasse

{

public:

MaClasse();

private:

static *int* monAttribut;

};

On peut cependant les rendre private, c’est une exception.

//Initialiser l'attribut en dehors de toute fonction ou classe (espace global)

*int* MaClasse::monAttribut = 5;

L’utilisation la plus connu est de réaliser un compteur d’instance.

*class* Personnage

{

public:

Personnage(string nom);

//Plein de méthodes…

~Personnage();

static *int* nombreInstances(); //Renvoie le nombre d'objets créés

private:

string m\_nom;

static *int* compteur;

}

Et tout se passe ensuite dans le .cpp correspondant :

*int* Personnage::compteur = 0; //On initialise notre compteur à 0

Personnage::Personnage(string nom)

:m\_nom(nom)

{

++compteur; //Quand on crée un personnage, on ajoute 1 au compteur

}

Personnage::~Personnage()

{

--compteur; //Et on enlève 1 au compteur lors de la destruction

}

*int* Personnage::nombreInstances()

{

return compteur; //On renvoie simplement la valeur du compteur

}

On peut alors à tout instant connaître le nombre de personnages présents dans le jeu en consultant la valeur de l'attribut Personnage::compteur, c'est-à-dire en appelant la méthode statique nombreInstances().

*int* main()

{

//On crée deux personnages

Personnage goliath("Goliath le tenebreux");

Personnage lancelot("Lancelot le preux");

//Et on consulte notre compteur

cout << "Il y a actuellement " << Personnage::nombreInstances() << " personnages en jeu." << endl;

return 0;

}

## L’amitité :

Dans la POO, l’amitié est le fait de donner un accès complet aux éléments d’une classe.

Donc si je déclare une fonction f amie de la classe A, la fonction f pourra modifier les attributs de la classe Amême si les attributs sont privés ou protégés. La fonction f pourra également utiliser les fonctions privées et protégées de la classe A.

Retour sur Duree :

*class* Duree

{

public:

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

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

private:

*int* m\_heures;

*int* m\_minutes;

*int* m\_secondes;

};

//Surcharge de l'opérateur << pour l'écriture dans les flux

//Utilise la méthode affiche() de Duree

ostream &operator<<( ostream &out, Duree const& duree );

Je ne vous ai mis que l'essentiel. Il y avait bien plus d'opérateurs déclarés à la fin du chapitre. Ce qui va nous intéresser, c'est la surcharge de l'opérateur d'injection dans les flux. Voici ce que nous avions écrit :

ostream &operator<<( ostream &out, Duree const& duree )

{

duree.afficher(out) ;

return out;

}

Sauf que afficher ne sert qu’à << et pas au reste. C’est donc une classe qui ne sert à rien pour un utilisateur normal.

Mais en faisant ça :

*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;

};

Il faut maintenant rentre operator<< ami de la classe :

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

On le déclare dans la classe.

*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);

};

Maintenant l’opérateur << a accès à tous les éléments de la classe.

### L’amitié et la responsabilité :

Il faut respecter ces deux aspects de l’amitié :

* une fonction amie ne doit pas, en principe, modifier l'instance de la classe ;
* les fonctions amies ne doivent être utilisées que si vous ne pouvez pas faire autrement.

# Introduction à Qt

## Les fenêtres :

### GUI :

*Graphical user interface.* Prononcé « Goui ».

## Présentation :

Plusieurs modules sont disponibles pour Qt :

* **Module GUI** : c'est toute la partie création de fenêtres. Nous nous concentrerons surtout, dans ce cours, sur le module GUI.
* **Module OpenGL** : Qt peut ouvrir une fenêtre contenant de la 3D gérée par OpenGL.
* **Module de dessin** : pour tous ceux qui voudraient dessiner dans leur fenêtre (en 2D), le module de dessin est très complet !
* **Module réseau** : Qt fournit une batterie d'outils pour accéder au réseau, que ce soit pour créer un logiciel de Chat, un client FTP, un client Bittorent, un lecteur de flux RSS…
* **Module SVG** : Qt permet de créer des images et animations vectorielles, à la manière de Flash.
* **Module de script** : Qt prend en charge le Javascript (ou ECMAScript), que vous pouvez réutiliser dans vos applications pour ajouter des fonctionnalités, par exemple sous forme de plugins.
* **Module XML** : pour ceux qui connaissent le XML, c'est un moyen très pratique d'échanger des données à partir de fichiers structurés à l'aide de balises, comme le XHTML.
* **Module SQL** : permet d'accéder aux bases de données (MySQL, Oracle, PostgreSQL…).

[documentation officielle](https://doc.qt.io/)

# Compiler une première fenêtre

Pour créer un projet : fichie 🡺 nouveau projet 🡺 qt vide.

Le code minimale pour une application C++ avec Qt :

main.cpp :

#include <QApplication>

*int* main(*int* argc, *char* \*argv[])

{

QApplication app(argc, argv);

return app.exec();

}

On crée une application Qt avec QApplication en lui passant les paramètres du programme.

Ensuite on exécute le programme.

### Affichage d’un widget :

Tout est considéré comme widget : bouton, checkbox … La fenêtre elle-même en est une.

Exemple :

#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();

}

Qt ne sert pas qu’à faire des fenêtres, on doit donc lui spécifier que l’on veut afficher une fenêtre.

Ouvrez le fichier .pro de votre projet. Il devrait contenir la liste des fichiers de votre projet :

SOURCES += \

    main.cpp

Ajoutez-y une ligne QT += widgets pour demander à Qt de charger le module QtWidgets :

QT += widgets

SOURCES += \

    main.cpp

## Diffuser le programme :

Pour diffuser le programme il faut les DLL : C:\Qt\5.7\msvc2015\_64\bin

On doit mettre dans le programme les DLL dont on a besoin pour le faire fonctionner.

# Modifier les propriétés d’un widget

## Modifier et personnaliser :

On crée un bouton avec QPushButton.

### Text :

Le texte n’est pas un string mais un Qstring. On le récupère avec Qstring text() const. On le modifie avec void setText().

#include <QApplication>

#include <QPushButton>

*int* main(*int* argc, *char* \*argv[])

{

QApplication app(argc, argv);

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

bouton.setText("Pimp mon bouton !");

bouton.show();

return app.exec();

}

### toolTip() l’infobulle :

setToolTip() :

#include <QApplication>

#include <QPushButton>

*int* main(*int* argc, *char* \*argv[])

{

QApplication app(argc, argv);

QPushButton bouton("Pimp mon bouton !");

bouton.setToolTip("Texte d'aide");

bouton.show();

return app.exec();

}

### Font : la police :

La propriété font est un peu plus complexe car elle contient trois informations :

* le nom de la police de caractères utilisée (Times New Roman, Arial, Comic Sans MS…) ;
* la taille du texte en pixels (12, 16, 18…) ;
* le style du texte (gras, italique…).

La signature de la méthode setFont est :

void setFont ( const QFont & )

Cela veut dire que setFont attend un objet de type QFont !

Rappel :

* **const**: signifie que l'objet passé en paramètre ne sera pas modifié par la fonction ;
* **&** : signifie que la fonction récupère une référence vers l'objet, ce qui lui évite d'avoir à le copier.

QFont attend 4 paramètres :

QFont ( const QString & family, int pointSize = -1, int weight = -1, bool italic = false )

Dans l'ordre, les paramètres signifient :

* family : le nom de la police de caractères à utiliser.
* pointSize : la taille des caractères en pixels.
* weight : le niveau d'épaisseur du trait (gras). Cette valeur peut être comprise entre 0 et 99 (du plus fin au plus gras). Vous pouvez aussi utiliser la constante QFont::Bold qui correspond à une épaisseur de 75.
* italic : un booléen, pour indiquer si le texte doit être affiché en italique ou non.

On va faire quelques tests. Tout d'abord, il faut créer un objet de type QFont :

QFont maPolice("Courier");

J'ai appelé cet objet maPolice.  
Maintenant, je dois envoyer l'objet maPolice de type QFont à la méthode setFont de mon bouton (suivez, suivez !) :

bouton.setFont(maPolice);

En résumé, j'ai donc dû écrire 2 lignes pour changer la police :

QFont maPolice("Courier");

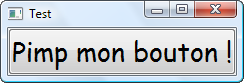
bouton.setFont(maPolice);

Ou plus cours :

bouton.setFont(QFont("Courier"));

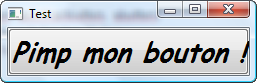
Autres exemples :

bouton.setFont(QFont("Comic Sans MS", 20));



Et voilà le même avec du gras et de l'italique (figure suivante) !

bouton.setFont(QFont("Comic Sans MS", 20, QFont::Bold, true));



### Cursor : curseur de la souris :

On peut en choisir parmi la [liste](https://www.qt-project.org/doc/qt.html#CursorShape-enum).

bouton.setCursor(Qt::PointingHandCursor);



### Icon : l’icône du cursor :

Qt lit les images de type PNG.

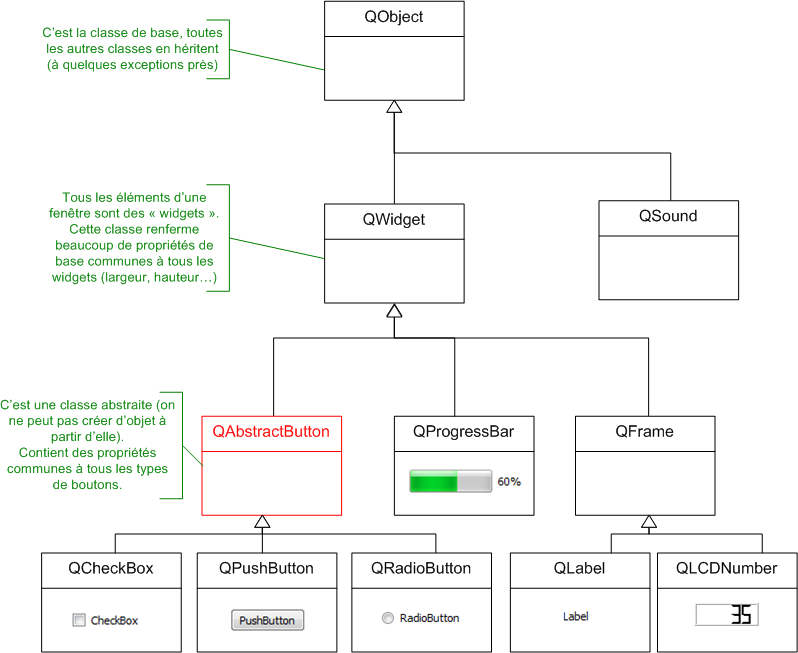
bouton.setIcon(QIcon("smile.png"));

L’image doit être prêt de l’executable et non du dossier de création du projet.

## Qt et l’héritage :

Dans la documentation, les classes de gauches sont parente, celle de droite fille ou sous-fille : [documentation](https://www.qt-project.org/doc/qt-5.1/qtdoc/hierarchy.html) .

### QObject la classe de base :



Comme tout est widget, même les fenêtres il existe une classe de base QWidget pour tous les widgets. Elle contient énormément de propriétés communes à tous les widgets, comme :

* la largeur ;
* la hauteur ;
* la position en abscisse (x) ;
* la position en ordonnée (y) ;
* la police de caractères utilisée (eh oui, la méthode setFont est définie dans QWidget et comme QPushButton en hérite, il possède lui aussi cette méthode) ;
* le curseur de la souris (pareil, rebelote, setCursor est en fait défini dans QWidget et non dans QPushButton car il est aussi susceptible de servir sur tous les autres widgets) ;
* l'infobulle (toolTip)
* etc.

Une classe abstraite sert de classe de base, elle définit des méthodes pour ses sous-classes.

Par exemple, parmi les propriétés communes on trouve :

* text: le texte affiché ;
* icon: l'icône affichée à côté du texte du bouton ;
* shortcut: le raccourci clavier pour activer le bouton ;
* down: indique si le bouton est enfoncé ou non ;
* etc.

Un widget qui n’est pas dans un autre widget est directement considéré comme une fenêtre.

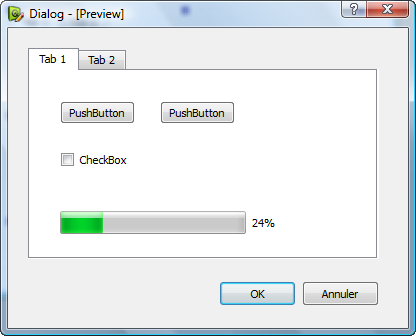
## Un widget peut en contenir un autre :

### Contenant et contenu :

Une fenêtre (QWidget) peut très bien contenir trois boutons :

* QPushButton
* QCheckBox
* QProgressBar

Exemple :



Sur cette capture, la fenêtre contient trois widgets :

* un bouton OK ;
* un bouton Annuler ;
* un conteneur avec des onglets.

Le conteneur avec des onglets est, comme son nom l'indique, un conteneur. Il contient à son tour des widgets :

* deux boutons ;
* une case à cocher (*checkbox*) ;
* une barre de progression.

Les widgets sont donc imbriqués les uns dans les autres suivant cette hiérarchie :

* QWidget (la fenêtre)
  + QPushButton
  + QPushButton
  + QTabWidget (le conteneur à onglets)
    - QPushButton
    - QPushButton
    - QCheckBox
    - QProgressBar

**Ce n’est pas de l’héritage, un widget peut en contenir un autre c’est tout !**

### Créer une fenêtre contenant un bouton :

#include <QApplication>

#include <QPushButton>

*int* main(*int* argc, *char* \*argv[])

{

QApplication app(argc, argv);

// Création d'un widget qui servira de fenêtre

QWidget fenetre;

fenetre.setFixedSize(300, 150);

// Création du bouton, ayant pour parent la "fenêtre"

QPushButton bouton("Pimp mon bouton !", &fenetre);

// Personnalisation du bouton

bouton.setFont(QFont("Comic Sans MS", 14));

bouton.setCursor(Qt::PointingHandCursor);

bouton.setIcon(QIcon("smile.png"));

// Affichage de la fenêtre

fenetre.show();

return app.exec();

}

Tous les widgets ont un constructeur qui permet d’indiquer qui est le parent :

QPushButton bouton("Pimp mon bouton !", &fenetre);

On peut déplacer le bouton avec bouton.move()

bouton.move(60, 50);

À noter aussi : la méthode setGeometry, qui prend 4 paramètres :

bouton.setGeometry(abscisse, ordonnee, largeur, hauteur);

### Tout widget peut en stocker un autre :

Même un bouton dans un bouton.

#include <QApplication>

#include <QPushButton>

*int* main(*int* argc, *char* \*argv[])

{

QApplication app(argc, argv);

QWidget fenetre;

fenetre.setFixedSize(300, 150);

QPushButton bouton("Pimp mon bouton !", &fenetre);

bouton.setFont(QFont("Comic Sans MS", 14));

bouton.setCursor(Qt::PointingHandCursor);

bouton.setIcon(QIcon("smile.png"));

bouton.setGeometry(60, 50, 180, 70);

// Création d'un autre bouton ayant pour parent le premier bouton

QPushButton autreBouton("Autre bouton", &bouton);

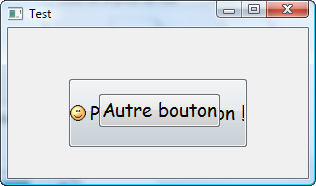
autreBouton.move(30, 15);

fenetre.show();

return app.exec();

}

Résultat : notre bouton est placé à l'intérieur de l'autre bouton (figure suivante) !



On a déjà inclus QPushButton, donc comme il hérite de QWidget il s’est inclus aussi, mais on a utilisé Qicon et d’autre sans les avoir inclus. Normalement ça n’aurait pas dû marcher.

#include <QApplication>

#include <QPushButton>

#include <QWidget>

#include <QFont>

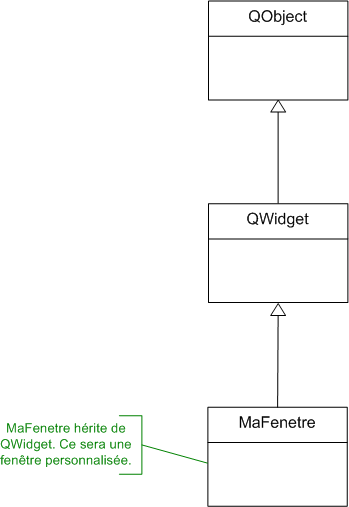
#include <QIcon>

Le plus facile est d’inclure directement QWidget :

#include <QtWidgets>

## Hériter un widget :

On veut créer sa propre fenêtre et hériter de QWidget.



### MaFenetre.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

### MaFenetre.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);

}

### Main.cpp

#include <QApplication>

#include "MaFenetre.h"

*int* main(*int* argc, *char* \*argv[])

{

QApplication app(argc, argv);

MaFenetre fenetre;

fenetre.show();

return app.exec();

}

Même si on a créé dynamiquement un widget, pas besoin de le supprime =r pour autant, Qt s’en occupe à la fermeture de la fenêtre.

# Les signaux et les slots

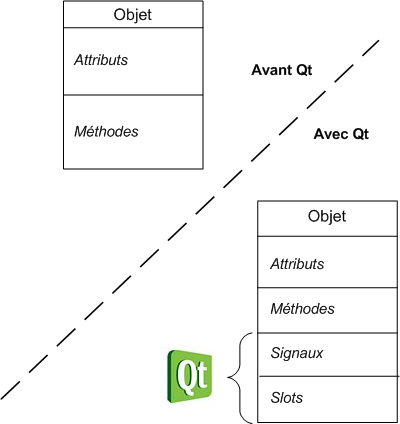
Créé par Qt :

* + **Un signal** : c'est un message envoyé par un widget lorsqu'un évènement se produit.

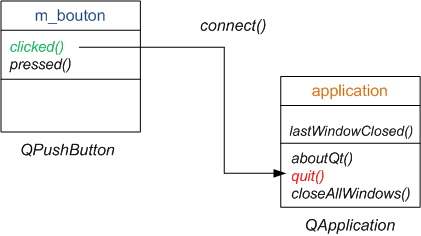
Exemple : on a cliqué sur un bouton.

* + **Un slot** : c'est la fonction qui est appelée lorsqu'un évènement s'est produit. On dit que le signal appelle le slot. Concrètement, un slot est une méthode d'une classe.

Exemple : le slot quit() de la classe QApplication provoque l'arrêt du programme.



## Connexion d’un signal à un slot :



Pour se faire, on utilise une méthode statique de QObject : conect().

### Connect() :

Comme elle est statique, on peut l’appeler sans créer d’objet et on l’appelle avec ::.

Il faut donc écrire :

QObject::connect();

La méthode connect prend 4 arguments :

* un pointeur vers l'objet qui émet le signal ;
* le nom du signal que l'on souhaite « intercepter » ;
* un pointeur vers l'objet qui contient le slot récepteur ;
* le nom du slot qui doit s'exécuter lorsque le signal se produit.

Donc pour quitter l’application :

#include "MaFenetre.h"

MaFenetre::MaFenetre() : QWidget()

{

setFixedSize(300, 150);

m\_bouton = new QPushButton("Quitter", this);

m\_bouton->setFont(QFont("Comic Sans MS", 14));

m\_bouton->move(110, 50);

// Connexion du clic du bouton à la fermeture de l'application

QObject::connect(m\_bouton, SIGNAL(clicked()), qApp, SLOT(quit()));

}

Étudions attentivement cette ligne et plus particulièrement les paramètres envoyés à connect() :

* m\_bouton : c'est un pointeur vers le bouton qui va émettre le signal. Facile.
* SIGNAL(clicked()) : là, c'est assez perturbant comme façon d'envoyer un paramètre. En fait, SIGNAL() est une macro du préprocesseur. Qt transformera cela en un code « acceptable » pour la compilation. Le but de cette technique est de vous faire écrire un code court et compréhensible. Ne cherchez pas à comprendre comment Qt fait pour transformer le code, ce n'est pas notre problème.
* qApp : c'est un pointeur vers l'objet de type QApplication que nous avons créé dans le main. D'où sort ce pointeur ? En fait, Qt crée automatiquement un pointeur appelé qApp vers l'objet de type QApplication que nous avons créé. Ce pointeur est défini dans le header <QApplication> que nous avons inclus dans MaFenetre.h.
* SLOT(quit()) : c'est le slot qui doit être appelé lorsqu'on a cliqué sur le bouton. Là encore, il faut utiliser la macro SLOT() pour que Qt traduise ce code « bizarre » en quelque chose de compilable.

## Des paramètres dans les signaux et slots :

### Dessiner la fenêtre :

On va utiliser deux nouveaux widget :

* QSlider : un curseur qui permet de définir une valeur ;
* QLCDNumber : un widget qui affiche un nombre.

#ifndef DEF\_MAFENETRE

#define DEF\_MAFENETRE

#include <QApplication>

#include <QWidget>

#include <QPushButton>

#include <QLCDNumber>

#include <QSlider>

*class* MaFenetre : public QWidget

{

public:

MaFenetre();

private:

QLCDNumber \*m\_lcd;

QSlider \*m\_slider;

};

#endif

#include "MaFenetre.h"

MaFenetre::MaFenetre() : QWidget()

{

setFixedSize(200, 100);

m\_lcd = new QLCDNumber(this);

m\_lcd->setSegmentStyle(QLCDNumber::Flat);

m\_lcd->move(50, 20);

m\_slider = new QSlider(Qt::Horizontal, this);

m\_slider->setGeometry(10, 60, 150, 20);

}

La connexion :

QObject::connect(m\_slider, SIGNAL(valueChanged(*int*)), m\_lcd, SLOT(display(*int*))) ;

## Créer ses propres slots et widgets :

Pour créer son propre signal ou slot il doit dériver directement de QObject. C’est le cas ici de QWidget qui hérite de QObject.

### Object :

On veut que notre QSlider contrôle lalargeur de la fenêtre.

On doit définir une marco Q\_OBJECT dans le header, au début de la déclaration.

*class* MaFenetre : public QWidget

{

Q\_OBJECT

public:

MaFenetre();

private:

QSlider \*m\_slider;

};

Cette macro prépare le compilateur à accepter un nouveau mot clé slot.

On doit executer qmake après avoir déclaré Q\_OBJECT.

L'implémentation est simple :

*void* MaFenetre::changerLargeur(*int* largeur)

{

setFixedSize(largeur, 100);

}

Pour la connexion :

MaFenetre::MaFenetre() : QWidget()

{

setFixedSize(200, 100);

m\_slider = new QSlider(Qt::Horizontal, this);

m\_slider->setRange(200, 600);

m\_slider->setGeometry(10, 60, 150, 20);

QObject::connect(m\_slider, SIGNAL(valueChanged(*int*)), this, SLOT(changerLargeur(*int*)));

}

### Créer son propre signal :

Si on a atteint le maximum d’agrandissement (600 ici), on veut émettre un signal agrandissementMax.

*class* MaFenetre : public QWidget

{

Q\_OBJECT

public:

MaFenetre();

public slots:

*void* changerLargeur(*int* largeur);

signals:

*void* agrandissementMax();

private:

QSlider \*m\_slider;

};

*void* MaFenetre::changerLargeur(*int* largeur)

{

setFixedSize(largeur, height());

if (largeur == 600)

{

emit agrandissementMax();

}

}

Pour déclencher un signal, on utilise emit.

On peut rajouter :

QObject::connect(this, SIGNAL(agrandissementMax()), qApp, SLOT(quit()));

Du coup, si on a atteint la taille max, l’appli se ferme.

# Les boites de dialogues usuelles

## Afficher un message :

Elles sont gérées par QMessageBox.

On va travailler avec ces fichiers :

// MaFenetre.h

#ifndef DEF\_MAFENETRE

#define DEF\_MAFENETRE

#include <QApplication>

#include <QWidget>

#include <QPushButton>

#include <QMessageBox>

*class* MaFenetre : public QWidget

{

Q\_OBJECT

public:

MaFenetre();

public slots:

*void* ouvrirDialogue();

private:

QPushButton \*m\_boutonDialogue;

};

#endif

// MaFenetre.cpp

#include "MaFenetre.h"

MaFenetre::MaFenetre() : QWidget()

{

setFixedSize(230, 120);

m\_boutonDialogue = new QPushButton("Ouvrir la boîte de dialogue", this);

m\_boutonDialogue->move(40, 50);

QObject::connect(m\_boutonDialogue, SIGNAL(clicked()), this, SLOT(ouvrirDialogue()));

}

*void* MaFenetre::ouvrirDialogue()

{

// Vous insérerez ici le code d'ouverture des boîtes de dialogue

}

C'est très simple. Nous avons créé dans la boîte de dialogue un bouton qui appelle le slot personnalisé ouvrirDialogue(). C'est dans ce slot que nous nous chargerons d'ouvrir une boîte de dialogue.

main.cpp n'a pas changé :

// main.cpp

#include <QApplication>

#include "MaFenetre.h"

*int* main(*int* argc, *char* \*argv[])

{

QApplication app(argc, argv);

MaFenetre fenetre;

fenetre.show();

return app.exec();

}

### Ouvrir une boite de dialogue avec une méthode statique :

La classe QMessageBox crée donc des objets QMessageBox mais on utilisera que ses méthodes statiques.

### QMessageBox ::information :

S’ouvre avec une icône information.

StandardButton information ( QWidget \* parent, const QString & title, const QString & text, StandardButtons buttons = Ok, StandardButton defaultButton= NoButton );

Seuls les trois premiers paramètres sont obligatoires, les autres ayant, comme vous le voyez, des valeurs par défaut.  
Ces trois premiers paramètres sont :

* parent : un pointeur vers la fenêtre parente (qui doit être de type QWidget ou hériter de QWidget). Vous pouvez envoyer NULL en paramètre si vous ne voulez pas que votre boîte de dialogue ait une fenêtre parente, mais ce sera plutôt rare.
* title : le titre de la boîte de dialogue (affiché en haut de la fenêtre).
* text : le texte affiché au sein de la boîte de dialogue.

Testons donc un code très simple. Voici le code du slot ouvrirDialogue() :

*void* MaFenetre::ouvrirDialogue()

{

QMessageBox::information(this, "Titre de la fenêtre", "Bonjour et bienvenueà tous les Zéros !");

}

La fenêtre est dite modale, ce qui signifie qu’elle bloque la fenêtre parente.

On peut d’ailleurs personnaliser l’affichage avec de l’HTML :

QMessageBox::information(this, "Titre de la fenêtre", "Bonjour et bienvenue à<strong>tous les Zéros !</strong>");

### QMessageBox ::warning :

Même procédé mais avec un warning.

QMessageBox::warning(this, "Titre de la fenêtre", "Attention, vous êtes peut-être un Zéro !");

### QMessageBox ::critical et :: question :

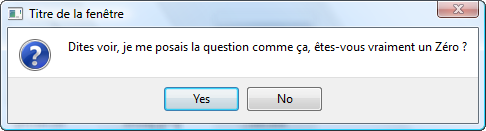
Même procédé.

### Personnaliser les boutons :

On rajoute un quatrième argument avec un flag :

La [liste](https://www.qt-project.org/doc/qmessagebox.html#StandardButton-enum) des flags disponibles est donnée par la documentation.

QMessageBox::question(this, "Titre de la fenêtre", "Dites voir, je me posais la question comme cela, êtes-vous vraiment un Zéro ?", QMessageBox::Yes | QMessageBox::No);



Pour traduire les boutons à partir de locale :

// main.cpp

#include <QApplication>

#include <QTranslator>

#include <QLocale>

#include <QLibraryInfo>

#include "MaFenetre.h"

*int* main(*int* argc, *char* \*argv[])

{

QApplication app(argc, argv);

QString locale = QLocale::system().name().section('\_', 0, 0);

QTranslator translator;

translator.load(QString("qt\_") + locale, QLibraryInfo::location(QLibraryInfo::TranslationsPath));

app.installTranslator(&translator);

MaFenetre fenetre;

fenetre.show();

return app.exec();

}

### Récupérer la valeur de retour :

La valeur retournée avec les méthodes satatiques est un int, donc onpeut récupérer avec un switch ou un if :

*void* MaFenetre::ouvrirDialogue()

{

*int* reponse = QMessageBox::question(this, "Interrogatoire", "Dites voir, je me posais la question comme cela, êtes-vous vraiment un Zéro ?", QMessageBox ::Yes | QMessageBox::No);

if (reponse == QMessageBox::Yes)

{

QMessageBox::information(this, "Interrogatoire", "Alors bienvenue chezles Zéros !");

}

else if (reponse == QMessageBox::No)

{

QMessageBox::critical(this, "Interrogatoire", "Tricheur ! Menteur ! Voleur ! Ingrat ! Lâche ! Traître !\nSors d'ici ou j'appelle la police !");

}

}

## Saisir une information :

Pour saisir ou choisir des informations, on utilise les boites de dialogue de saisies. Elles sont gérées par la classe QInputDialog.

Les boîtes de dialogue « saisir une information » peuvent être de quatre types :

1. saisir un texte : QInputDialog::getText() ;
2. saisir un entier : QInputDialog::getInteger() ;
3. saisir un nombre décimal : QInputDialog::getDouble() ;
4. choisir un élément parmi une liste : QInputDialog::getItem().

La méthode statique QInputDialog::getText() ouvre une boîte de dialogue qui permet à l'utilisateur de saisir un texte.  
Son prototype est :

QString QInputDialog::getText ( QWidget \* parent, const QString & title, const QString & label, QLineEdit::EchoMode mode = QLineEdit::Normal, const QString & text = QString(), *bool* \* ok = 0, Qt::WindowFlags f = 0 );

Vous pouvez tout d'abord constater que la méthode renvoie un QString, c'est-à-dire une chaîne de caractères de Qt.  
Les paramètres signifient, dans l'ordre :

* parent : pointeur vers la fenêtre parente. Peut être mis à NULL pour ne pas indiquer de fenêtre parente.
* title : titre de la fenêtre, affiché en haut.
* label : texte affiché dans la fenêtre.
* mode : mode d'édition du texte. Permet de dire si on veut que les lettres s'affichent quand on tape, ou si elles doivent être remplacées par des astérisques (pour les mots de passe) ou si aucune lettre ne doit s'afficher. Toutes les options sont dans la documentation. Par défaut, les lettres s'affichent normalement (QLineEdit::Normal).
* text : texte par défaut dans la zone de saisie.
* ok : pointeur vers un booléen pour que Qt puisse vous dire si l'utilisateur a cliqué sur OK ou sur Annuler.
* f : quelques flags (options) permettant d'indiquer si la fenêtre est modale (bloquante) ou pas. Les valeurs possibles sont détaillées par la documentation.

*void* MaFenetre::ouvrirDialogue()

{

QString pseudo = QInputDialog::getText(this, "Pseudo", "Quel est votre pseudo ?");

}

On peut aller plus loin :

*void* MaFenetre::ouvrirDialogue()

{

*bool* ok = false;

QString pseudo = QInputDialog::getText(this, "Pseudo", "Quel est votre pseudo ?", QLineEdit::Normal, QString(), &ok);

if (ok && !pseudo.isEmpty())

{

QMessageBox::information(this, "Pseudo", "Bonjour " + pseudo + ", ça va ?");

}

else

{

QMessageBox::critical(this, "Pseudo", "Vous n'avez pas voulu donner votre nom… snif.");

}

}

Si on veut mettre &ok il faut renseigner tous les autres champs avant.

## Séléctionner une police :

Le prototype :

QFont getFont ( *bool* \* ok, const QFont & initial, QWidget \* parent, const QString & caption )

Exemple :

*void* MaFenetre::ouvrirDialogue()

{

*bool* ok = false;

QFont police = QFontDialog::getFont(&ok, m\_boutonDialogue->font(), this, "Choisissez une police");

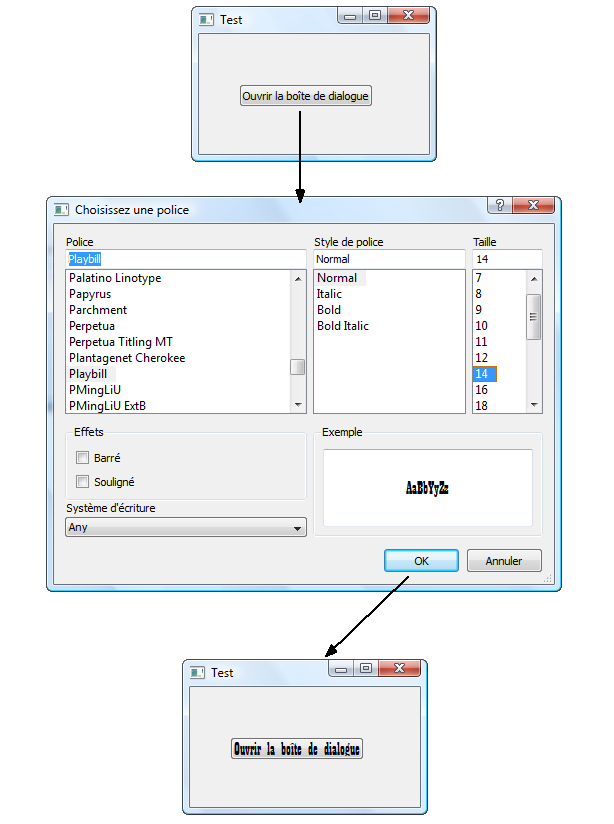
if (ok)

{

m\_boutonDialogue->setFont(police);

}

}



## Sélectionner une couleur :

Le prototype :

QColor QColorDialog::getColor ( const QColor & initial = Qt::white, QWidget \* parent = 0 );

Exemple :

*void* MaFenetre::ouvrirDialogue()

{

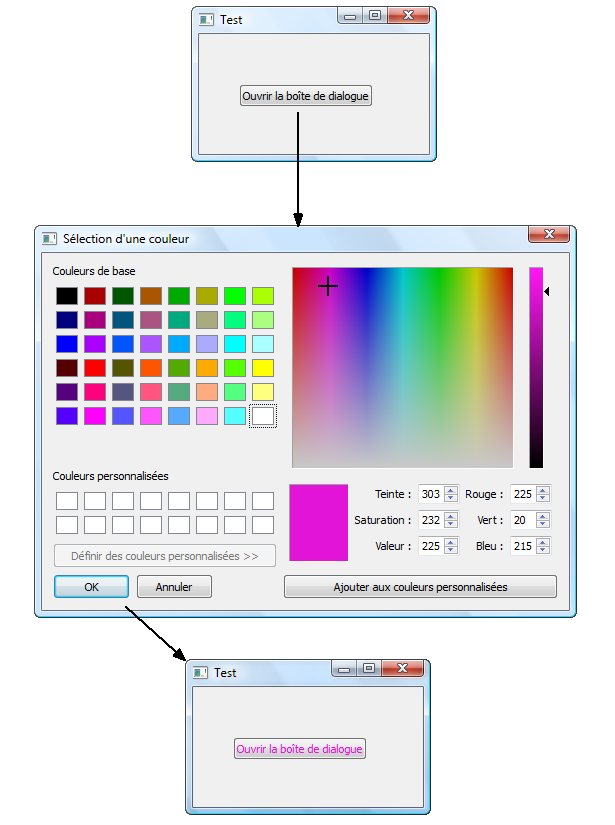
QColor couleur = QColorDialog::getColor(Qt::white, this);

QPalette palette;

palette.setColor(QPalette::ButtonText, couleur);

m\_boutonDialogue->setPalette(palette);

}



## Sélection d’un fichier ou d’un dossier :

### Sélection d’un dossier :

QFileDialog ::getExistingDirectory

QString dossier = QFileDialog::getExistingDirectory(this);

### Sélection d’un fichier :

QFileDialog ::getOpenFileName

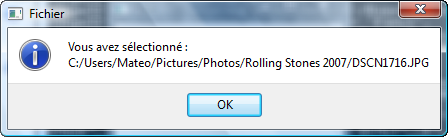
*void* MaFenetre::ouvrirDialogue()

{

QString fichier = QFileDialog::getOpenFileName(this, "Ouvrir un fichier", QString(), "Images (\*.png \*.gif \*.jpg \*.jpeg)");

QMessageBox::information(this, "Fichier", "Vous avez sélectionné :\n" + fichier);

}



Le principe est de donner le chemin d’accès, pas de l’ouvrir, c’est à nous d’ouvrir ensuite.

### Enregistrement d’un fichier :

QFileDialog ::getSaveFileName :

Ouvrir est remplacé par enregistré.

QString fichier = QFileDialog::getSaveFileName(this, "Enregistrer un fichier", QString(), "Images (\*.png \*.gif \*.jpg \*.jpeg)");

# Apprendre à lire la documentation Qt

## Où trouver la doc ?

Il y a en fait deux moyens d'accéder à la documentation :

* si vous avez accès à Internet : vous pouvez aller sur le site de Digia (l'entreprise qui édite Qt) ;
* si vous n'avez pas d'accès à Internet : vous pouvez utiliser le programme Qt Creator qui contient toute la documentation.

L'adresse est simple à retenir : [http://qt-project.org/doc](https://www.qt-project.org/doc).

# Positionner ses widgets avec les layout

## Positionnement absolue et ses défauts :

Pour les exemples, on n’utilisera que une seule fenêtre main.cpp

#include <QApplication>

#include <QPushButton>

*int* main(*int* argc, *char* \*argv[])

{

QApplication app(argc, argv);

QWidget fenetre;

QPushButton bouton("Bonjour", &fenetre);

bouton.move(70, 60);

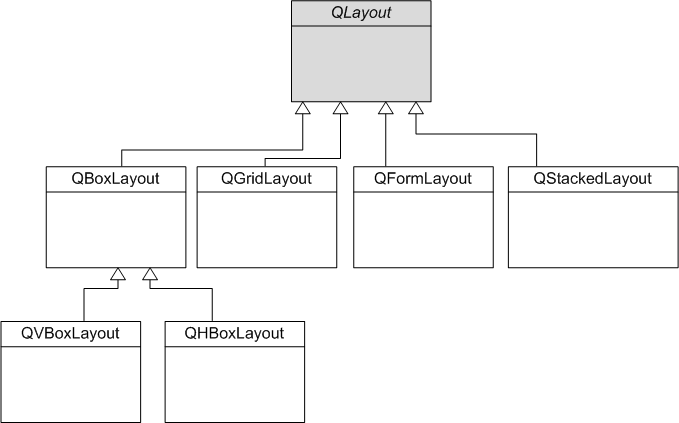
fenetre.show();

return app.exec();

}

Si on réduit la fenêtre, on peut masquer le bouton, idem si quelqu’un travail avec un petit écran.

## Archetecture des classes de layout :



## Layout horizontaux et verticaux :

On travaillera donc avec :

* QHBoxLayout ;
* QVBoxLayout.

L'utilisation d'un layout se fait en 3 temps :

1. **On crée les widgets ;**
2. **On crée le layout et on place les widgets dedans ;**
3. **On dit à la fenêtre d'utiliser le layout qu'on a créé.**

On utilisera des boutons :

QPushButton \*bouton1 = new QPushButton("Bonjour");

QPushButton \*bouton2 = new QPushButton("les");

QPushButton \*bouton3 = new QPushButton("Zéros");

On n’a pas mis de parent aux boutons et c’est fait exprés, on va les mettre dans le layout.

QHBoxLayout \*layout = new QHBoxLayout;

Le constructeur de cette classe est simple, on n'a pas besoin d'indiquer de paramètre.

Maintenant que notre layout est créé, rajoutons nos widgets à l'intérieur :

layout->addWidget(bouton1);

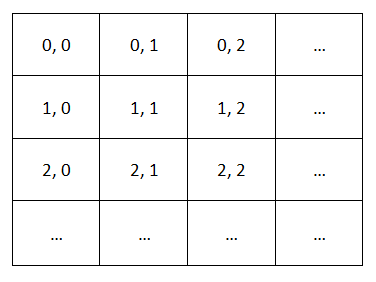
layout->addWidget(bouton2);

layout->addWidget(bouton3);

Maintenant on indique à la fenêtre qu’on va utiliser le layout :

fenetre.setLayout(layout);

## Le layout de grille :



#include <QApplication>

#include <QPushButton>

#include <QGridLayout>

*int* main(*int* argc, *char* \*argv[])

{

QApplication app(argc, argv);

QWidget fenetre;

QPushButton \*bouton1 = new QPushButton("Bonjour");

QPushButton \*bouton2 = new QPushButton("les");

QPushButton \*bouton3 = new QPushButton("Zéros");

QGridLayout \*layout = new QGridLayout;

layout->addWidget(bouton1, 0, 0);

layout->addWidget(bouton2, 0, 1);

layout->addWidget(bouton3, 1, 0);

fenetre.setLayout(layout);

fenetre.show();

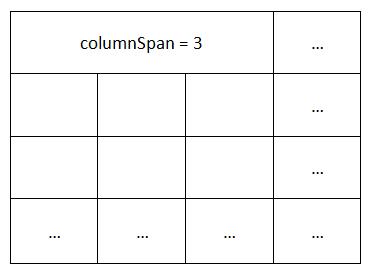
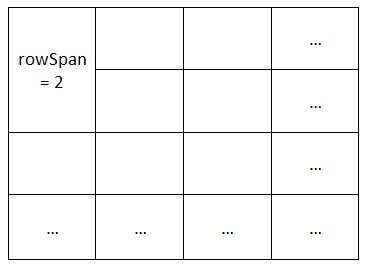
return app.exec();

}

### Un widget qui occupe plusieurs cases :

On va utiliser une version surchargée de addWidget qui orend deux paramètres supplémentaires :

* rowSpan : nombre de lignes qu'occupe le widget (par défaut 1) ;
* columnSpan : nombre de colonnes qu'occupe le widget (par défaut 1).

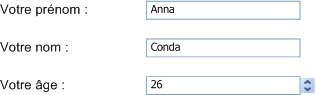


Exemple, bouton sur 2 colonnes :

layout->addWidget(bouton3, 1, 0, 1, 2);

## Le layout de formulaire :

QFormLayout :



Au lieu d’utiliser addWidget on utilisera addRow avec deux paramètre :

* le texte du libellé ;
* un pointeur vers le champ du formulaire.

#include <QApplication>

#include <QLineEdit>

#include <QFormLayout>

*int* main(*int* argc, *char* \*argv[])

{

QApplication app(argc, argv);

QWidget fenetre;

QLineEdit \*nom = new QLineEdit;

QLineEdit \*prenom = new QLineEdit;

QLineEdit \*age = new QLineEdit;

QFormLayout \*layout = new QFormLayout;

layout->addRow("Votre nom", nom);

layout->addRow("Votre prénom", prenom);

layout->addRow("Votre âge", age);

fenetre.setLayout(layout);

fenetre.show();

return app.exec();

}

On peut aussi utiliser des raccourcis claviers, en ajouter & devant la fameuse lettre :

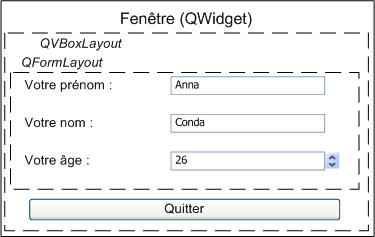
layout->addRow("Votre &nom", nom);

layout->addRow("Votre &prénom", prenom);

layout->addRow("Votre â&ge", age);

## Combiner les layouts :

On va mettre notre formulaire dans un QVBoxLayout puis ajouter un bouton Quitter.



QVBoxLayout contient deux choses, dans l'ordre :

1. Un QFormLayout (qui contient lui-même d'autres widgets) ;
2. Un QPushButton.

### Utilisation de addLayout :

#include <QApplication>

#include <QLineEdit>

#include <QPushButton>

#include <QVBoxLayout>

#include <QFormLayout>

*int* main(*int* argc, *char* \*argv[])

{

QApplication app(argc, argv);

QWidget fenetre;

// Création du layout de formulaire et de ses widgets

QLineEdit \*nom = new QLineEdit;

QLineEdit \*prenom = new QLineEdit;

QLineEdit \*age = new QLineEdit;

QFormLayout \*formLayout = new QFormLayout;

formLayout->addRow("Votre &nom", nom);

formLayout->addRow("Votre &prénom", prenom);

formLayout->addRow("Votre â&ge", age);

// Création du layout principal de la fenêtre (vertical)

QVBoxLayout \*layoutPrincipal = new QVBoxLayout;

layoutPrincipal->addLayout(formLayout); // Ajout du layout de formulaire

QPushButton \*boutonQuitter = new QPushButton("Quitter");

QWidget::connect(boutonQuitter, SIGNAL(clicked()), &app, SLOT(quit()));

layoutPrincipal->addWidget(boutonQuitter); // Ajout du bouton

fenetre.setLayout(layoutPrincipal);

fenetre.show();

return app.exec();

}

# Les principaux widgets

## Les fenêtres :

Il n’existe pas de classe QFenetre puisqu’est considéré comme fenêtre, une classe n’ayant pas de widget parent.

Il y a toutefois 2 classes :

* QMainWindow: c'est un widget spécial qui permet de créer la fenêtre principale de l'application. Une fenêtre principale peut contenir des menus, une barre d'outils, une barre d'état, etc.
* QDialog: c'est une classe de base utilisée par toutes les classes de boîtes de dialogue qu'on a vues il y a quelques chapitres. On peut aussi s'en servir directement pour ouvrir des boîtes de dialogue personnalisées.

### Une fenêtre avec QWidget :

Quelques méthodes utiles :

* cursor: curseur de la souris à afficher lors du survol du widget. La méthode setCursor attend que vous lui envoyiez un objet de type QCursor. Certains curseurs classiques (comme le sablier) sont prédéfinis dans une énumération. La documentation vous propose un lien vers cette énumération.
* enabled: indique si le widget est activé, si on peut le modifier. Un widget désactivé est généralement grisé. Si vous appliquez setEnabled(false) à toute la fenêtre, c'est toute la fenêtre qui devient inutilisable.
* height: hauteur du widget.
* size: dimensions du widget. Vous devrez indiquer la largeur et la hauteur.
* visible: contrôle la visibilité du widget.
* width: largeur.

N'oubliez pas : pour modifier une de ces propriétés, préfixez la méthode par un set. Exemple :

maFenetre.setWidth(200);

### Propriétés utilisables que sur les fenêtres :

Elles commencent toujours par window.

windowFlags: une série d'options contrôlant le comportement de la fenêtre. Il faut consulter l'énumération Qt::WindowType pour connaître les différents types disponibles. Vous pouvez aussi consulter l'exemple Window Flags du programme « Qt Examples and Demos ».

### Une fenêtre avec QDialog :

Les QDialog peuvent être de 2 types :

* **modales**: on ne peut pas accéder aux autres fenêtres de l'application lorsqu'elles sont ouvertes ;
* **non modales** : on peut toujours accéder aux autres fenêtres.

Elles sont modales par défaut.

#include <QApplication>

#include <QtWidgets>

*int* main(*int* argc, *char* \*argv[])

{

QApplication app(argc, argv);

QWidget fenetre;

QPushButton \*bouton = new QPushButton("Ouvrir la fenêtre", &fenetre);

QDialog secondeFenetre (&fenetre);

QVBoxLayout \*layout = new QVBoxLayout;

QLabel \*image = new QLabel(&secondeFenetre);

image->setPixmap(QPixmap("icone.png"));

layout->addWidget(image);

secondeFenetre.setLayout(layout);

QWidget::connect(bouton, SIGNAL(clicked()), &secondeFenetre, SLOT(exec()));

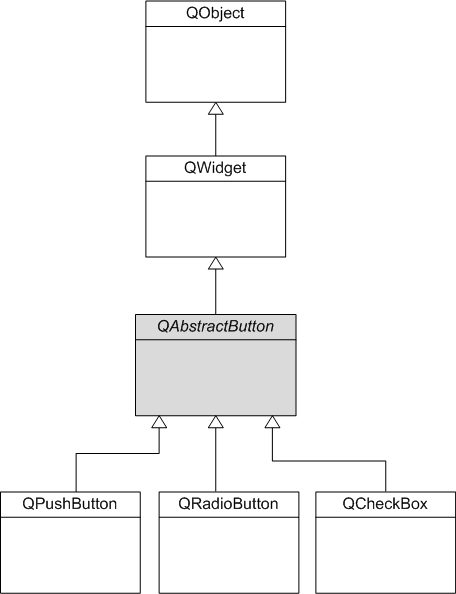
fenetre.show();

return app.exec();

}

## Les boutons :

* QPushButton: un bouton classique, que vous avez déjà largement eu l'occasion de manipuler ;
* QRadioButton: un bouton « radio », pour un choix à faire parmi une liste ;
* QCheckBox: une case à cocher (on considère que c'est un bouton en GUI Design).



On définit le libellé de la case lors de l'appel du constructeur :

#include <QApplication>

#include <QWidget>

#include <QCheckBox>

*int* main(*int* argc, *char* \*argv[])

{

QApplication app(argc, argv);

QWidget fenetre;

QCheckBox \*checkbox = new QCheckBox("J'aime les frites", &fenetre);

fenetre.show();

return app.exec();

}

Le signal renvoyé à la modification : stateChanged(bool).

Pour vérifié si l’était est chocked ou pas : isChecked().

Si on a plusieurs cases à cocher, on peut les regrouper dans un QGroupBox.

### QRadioButton :

#include <QApplication>

#include <QtWidgets>

*int* main(*int* argc, *char* \*argv[])

{

QApplication app(argc, argv);

QWidget fenetre;

QGroupBox \*groupbox = new QGroupBox("Votre plat préféré", &fenetre);

QRadioButton \*steacks = new QRadioButton("Les steacks");

QRadioButton \*hamburgers = new QRadioButton("Les hamburgers");

QRadioButton \*nuggets = new QRadioButton("Les nuggets");

steacks->setChecked(true);

QVBoxLayout \*vbox = new QVBoxLayout;

vbox->addWidget(steacks);

vbox->addWidget(hamburgers);

vbox->addWidget(nuggets);

groupbox->setLayout(vbox);

groupbox->move(5, 5);

fenetre.show();

return app.exec();

}

## Les afficheurs :

Principalement utilisés :

* QLabel: le plus important, un widget permettant d'afficher du texte ou une image ;
* QProgressBar: une barre de progression.

### Qlabel :

Affiche texte ou image :

Rien de plus simple, on utilise setText() :

label->setText("Bonjour les Zéros !");

Mais on peut aussi afficher un texte simple dès l'appel au constructeur, comme ceci :

QLabel \*label = new QLabel("Bonjour les Zéros !", &fenetre);

Pour afficher une image :

QLabel \*label = new QLabel(&fenetre);

label->setPixmap(QPixmap("icone.png"));

### QProgressBar :

Voici quelques propriétés utiles de la barre de progression :

* maximum: la valeur maximale que peut prendre la barre de progression ;
* minimum: la valeur minimale que peut prendre la barre de progression ;
* value: la valeur actuelle de la barre de progression.

On utilisera donc setValue pour changer la valeur de la barre de progression. Par défaut les valeurs sont comprises entre 0 et 100%.

Qt ne peut pas deviner où en sont vos opérations. C'est à vous de calculer leur pourcentage d'avancement. La QProgressBar se contente d'afficher le résultat.

Une QProgressBar envoie un signal valueChanged() lorsque sa valeur a été modifiée.

## Les champs :

* QLineEdit: champ de texte à une seule ligne ;
* QTextEdit: champ de texte à plusieurs lignes pouvant afficher du texte mis en forme ;
* QSpinBox: champ de texte adapté à la saisie de nombre entiers ;
* QDoubleSpinBox: champ de texte adapté à la saisie de nombre décimaux ;
* QSlider: curseur qui permet de sélectionner une valeur ;
* QComboBox: liste déroulante.

Quelques propriétés à connaître :

* text: permet de récupérer/modifier le texte contenu dans le champ.
* alignment: alignement du texte à l'intérieur.
* echoMode: type d'affichage du texte. Il faudra utiliser l'énumération EchoMode pour indiquer le type d'affichage. Par défaut, les lettres saisies sont affichées mais on peut aussi faire en sorte que les lettres soient masquées, pour les mots de passe par exemple.

lineEdit->setEchoMode(QLineEdit::Password);

* inputMask: permet de définir un masque de saisie, pour obliger l'utilisateur à fournir une chaîne répondant à des critères précis (par exemple, un numéro de téléphone ne doit pas contenir de lettres). Vous pouvez aussi jeter un coup d'œil aux *validators*qui sont un autre moyen de valider la saisie de l'utilisateur.
* maxLength: le nombre de caractères maximum qui peuvent être saisis.
* readOnly: le contenu du champ de texte ne peut être modifié. Cette propriété ressemble à enabled (définie dans QWidget) mais, avec readOnly, on peut quand même copier-coller le contenu du QLineEdit tandis qu'avec enabled, le champ est complètement grisé et on ne peut pas récupérer son contenu.

### QTextEdit :

Comme QLineEdit mais sur plusieurs lignes.

Notez les propriétés plainText et html qui permettent de récupérer et modifier le contenu respectivement sous forme de texte simple et sous forme de texte enrichi en HTML. Tout dépend de l'utilisation que vous en faites : normalement, dans la plupart des cas, vous utiliserez plutôt plainText.

### QSpinBox :

Comme QLineEdit mais avec des flèches pour changer la valeur et n’accepte que les entiers.

QSpinBox hérite de QAbstractSpinBox (tiens, encore une classe abstraite). Vérifiez donc aussi la documentation de QAbstractSpinBox pour connaître toutes les propriétés de la spinbox.

Voici quelques propriétés intéressantes :

* accelerated: permet d'autoriser la spinbox a accélérer la modification du nombre si on appuie longtemps sur le bouton.
* minimum: valeur minimale que peut prendre la spinbox.
* maximum: valeur maximale que peut prendre la spinbox.
* singleStep: pas d'incrémentation (par défaut de 1). Si vous voulez que les boutons fassent varier la spinbox de 100 en 100, c'est cette propriété qu'il faut modifier !
* value: valeur contenue dans la spinbox.
* prefix: texte à afficher avant le nombre.
* suffix: texte à afficher après le nombre.

A noter aussi QDoubleSpinBox pour prendre des nombres à virgule en paramètre.

### QSlider :

Un curseur pour sélectionner une valeur.

Il renvoit un signal valueCHanged(int).

### QCombo : une liste déroulante :

Une QComboBox est une liste déroulante.



On ajoute des valeurs à la liste déroulante avec la méthode addItem :

QComboBox \*liste = new QComboBox(&fenetre);

liste->addItem("Paris");

liste->addItem("Londres");

liste->addItem("Singapour");

liste->addItem("Tokyo");

On dispose de propriétés permettant de contrôler le fonctionnement de la QComboBox :

* + count: nombre d'éléments dans la liste déroulante.
  + currentIndex: numéro d'indice de l'élément actuellement sélectionné. Les indices commencent à 0. Ainsi, si currentIndex renvoie 2, c'est que « Singapour » a été sélectionné dans l'exemple précédent.
  + currentText: texte correspondant à l'élément sélectionné. Si on a sélectionné « Singapour », cette propriété contient donc « Singapour ».
  + editable: indique si le widget autorise l'ajout de valeurs personnalisées ou non. Par défaut, l'ajout de nouvelles valeurs est interdit.

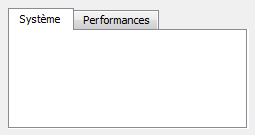
La QComboBox émet des signaux comme currentIndexChanged() qui indique qu'un nouvel élément a été sélectionné et highlighted() qui indique l'élément survolé par la souris (ces signaux peuvent envoyer un int pour donner l'indice de l'élément ou un QString pour le texte).

## Les conteneurs :

Même si tous les widgets peuvent en contenir d’autres, certains ont été conçu specialement pour ça :

* QFrame : un widget pouvant avoir une bordure ;
* QGroupBox : un widget (que nous avons déjà utilisé) adapté à la gestion des groupes de cases à cocher et de boutons radio ;
* QTabWidget : un widget gérant plusieurs pages d'onglets.

Le nouveau, QTabWidget :



**Il ne peut contenir qu’un widget par page.**

1. Créer un QTabWidget.
2. Créer un QWidget pour chacune des pages (chacun des onglets) du QTabWidget, sans leur indiquer de widget parent.
3. Placer des widgets enfants dans chacun de ces QWidget pour peupler le contenu de chaque page. Utiliser un layout pour positionner les widgets de préférence.
4. Appeler plusieurs fois addTab() pour créer les pages d'onglets en indiquant l'adresse du QWidget qui contient la page à chaque fois.

#include <QApplication>

#include <QtWidgets>

*int* main(*int* argc, *char* \*argv[])

{

QApplication app(argc, argv);

QWidget fenetre;

// 1 : Créer le QTabWidget

QTabWidget \*onglets = new QTabWidget(&fenetre);

onglets->setGeometry(30, 20, 240, 160);

// 2 : Créer les pages, en utilisant un widget parent pour contenir chacune des pages

QWidget \*page1 = new QWidget;

QWidget \*page2 = new QWidget;

QLabel \*page3 = new QLabel; // Comme un QLabel est aussi un QWidget (il en hérite), on peut aussi s'en servir de page

// 3 : Créer le contenu des pages de widgets

// Page 1

QLineEdit \*lineEdit = new QLineEdit("Entrez votre nom");

QPushButton \*bouton1 = new QPushButton("Cliquez ici");

QPushButton \*bouton2 = new QPushButton("Ou là…");

QVBoxLayout \*vbox1 = new QVBoxLayout;

vbox1->addWidget(lineEdit);

vbox1->addWidget(bouton1);

vbox1->addWidget(bouton2);

page1->setLayout(vbox1);

// Page 2

QProgressBar \*progress = new QProgressBar;

progress->setValue(50);

QSlider \*slider = new QSlider(Qt::Horizontal);

QPushButton \*bouton3 = new QPushButton("Valider");

QVBoxLayout \*vbox2 = new QVBoxLayout;

vbox2->addWidget(progress);

vbox2->addWidget(slider);

vbox2->addWidget(bouton3);

page2->setLayout(vbox2);

// Page 3 (je ne vais afficher qu'une image ici, pas besoin de layout)

page3->setPixmap(QPixmap("icone.png"));

page3->setAlignment(Qt::AlignCenter);

// 4 : ajouter les onglets au QTabWidget, en indiquant la page qu'ils contiennent

onglets->addTab(page1, "Coordonnées");

onglets->addTab(page2, "Progression");

onglets->addTab(page3, "Image");

fenetre.show();

return app.exec();

}

# TP : générateur de classe C++

Main.cpp

#include <QApplication>

#include "FenPrincipale.h"

*int* main(*int* argc, *char*\* argv[])

{

QApplication app(argc, argv);

FenPrincipale fenetre;

fenetre.show();

return app.exec();

}

FenPrincipale.h :

#ifndef HEADER\_FENPRINCIPALE

#define HEADER\_FENPRINCIPALE

#include <QtWidgets>

*class* FenPrincipale : public QWidget

{

Q\_OBJECT

public:

FenPrincipale();

private slots:

*void* genererCode();

private:

QLineEdit \*nom;

QLineEdit \*classeMere;

QCheckBox \*protections;

QCheckBox \*genererConstructeur;

QCheckBox \*genererDestructeur;

QGroupBox \*groupCommentaires;

QLineEdit \*auteur;

QDateEdit \*date;

QTextEdit \*role;

QPushButton \*generer;

QPushButton \*quitter;

};

#endif

FenPrincipale.cpp :

#include "FenPrincipale.h"

#include "FenCodeGenere.h"

FenPrincipale::FenPrincipale()

{

// Création des layouts et des widgets

// …

// Connexions des signaux et des slots

connect(quitter, SIGNAL(clicked()), qApp, SLOT(quit()));

connect(generer, SIGNAL(clicked()), this, SLOT(genererCode()));

}

*void* FenPrincipale::genererCode()

{

// On vérifie que le nom de la classe n'est pas vide, sinon on arrête

if (nom->text().isEmpty())

{

QMessageBox::critical(this, "Erreur", "Veuillez entrer au moins un nom de classe");

return; // Arrêt de la méthode

}

// Si tout va bien, on génère le code

QString code;

// Génération du code à l'aide des informations de la fenêtre

// …

// On crée puis affiche la fenêtre qui affichera le code généré, qu'on lui envoie en paramètre

FenCodeGenere \*fenetreCode = new FenCodeGenere(code, this);

fenetreCode->exec();

}

FenCodeGenere.h :

#ifndef HEADER\_FENCODEGENERE

#define HEADER\_FENCODEGENERE

#include <QtWidgets>

*class* FenCodeGenere : public QDialog

{

public:

FenCodeGenere(QString &code, QWidget \*parent);

private:

QTextEdit \*codeGenere;

QPushButton \*fermer;

};

#endif

FenCodeGenere.cpp

#include "FenCodeGenere.h"

FenCodeGenere::FenCodeGenere(QString &code, QWidget \*parent = 0) : QDialog(parent)

{

codeGenere = new QTextEdit();

codeGenere->setPlainText(code);

codeGenere->setReadOnly(true);

codeGenere->setFont(QFont("Courier"));

codeGenere->setLineWrapMode(QTextEdit::NoWrap);

fermer = new QPushButton("Fermer");

QVBoxLayout \*layoutPrincipal = new QVBoxLayout;

layoutPrincipal->addWidget(codeGenere);

layoutPrincipal->addWidget(fermer);

resize(350, 450);

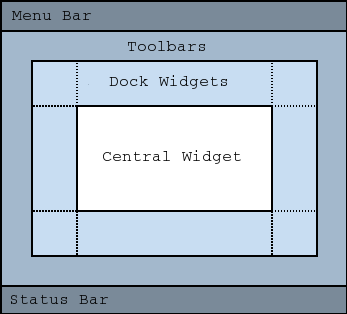
setLayout(layoutPrincipal);

connect(fermer, SIGNAL(clicked()), this, SLOT(accept()));

}

# La fenêtre principale

La fenêtre principale est définie avec la classe QMainWindow qui hérite directement de QWidget.



* **Menu Bar** : c'est la barre de menus. C'est là que vous allez pouvoir créer votre menu Fichier, Édition, Affichage, Aide, etc.
* **Toolbars**: les barres d'outils. Dans un éditeur de texte, on a par exemple des icônes pour créer un nouveau fichier, pour enregistrer, etc.
* **Dock Widgets** : plus complexes et plus rarement utilisés, ces docks sont des conteneurs que l'on place autour de la fenêtre principale. Ils peuvent contenir des outils, par exemple les différents types de pinceaux que l'on peut utiliser quand on fait un logiciel de dessin.
* **Central Widget** : c'est le cœur de la fenêtre, là où il y aura le contenu proprement dit.
* **Status Bar** : c'est la barre d'état. Elle affiche en général l'état du programme (« Prêt/Enregistrement en cours », etc.).

On va créer une classe qui hérite de QMainWindow.

##### main.cpp

#include <QApplication>

#include <QtWidgets>

#include "FenPrincipale.h"

*int* main(*int* argc, *char* \*argv[])

{

QApplication app(argc, argv);

FenPrincipale fenetre;

fenetre.show();

return app.exec();

}

##### FenPrincipale.h

#ifndef HEADER\_FENPRINCIPALE

#define HEADER\_FENPRINCIPALE

#include <QtWidgets>

*class* FenPrincipale : public QMainWindow

{

public:

FenPrincipale();

private:

};

#endif

##### FenPrincipale.cpp

#include "FenPrincipale.h"

FenPrincipale::FenPrincipale()

{

}

## Zone centrale : SDI et MDI :

Comme pour les onglets, elle ne peut contenir qu’un seul widget.

* **Les SDI** (*Single Document Interface*) : elles ne peuvent afficher qu'un document à la fois. C'est le cas du Bloc-Notes par exemple.
* **Les MDI** (*Multiple Document Interface*) : elles peuvent afficher plusieurs documents à la fois. Elles affichent des sous-fenêtres dans la zone centrale. C'est le cas par exemple de Qt Designer (figure suivante).

