Objectif du tutorial

Ce tutorial a pour objectif dans un premier temps de vous permettre d'apprendre à vous servir des outils mis à votre disposition à l'ENSEIRB afin de développer en C++. Nous aborderons l'utilisation des ressources logicielles permettant d'éditer, de compiler et de tester vos programmes écrits dans le langage C++.

Dans un second temps, ce tutorial doit vous permettre de manière simple de mettre en oeuvre les concepts objets que vous avez entrevus en cours.

Ce tutorial va être centré sur 4 développements simples mettant en oeuvre des objets quasi-élémentaires afin de vous permettre de comprendre sereinement le contexte associé à la programmation orientée objet.

Les ressources informatiques

La mise en oeuvre de ce tutorial sera effectuée sur des stations MacOS possédant le compilateur GCC installé. L'environnement de développement Eclipse a été installé sur les stations Linux et les stations SUN pour les personnes souhaitant l'utiliser.

Votre premier programme

Comme dans tout tutorial appliqué à un nouveau langage, nous allons commencer par réaliser le fameux programme "Hello World" avec les nouvelles possibilités offertes par le langage C++ (gestion des flux de données).

Pour cela, nous allons écrire le programme suivant :

```cpp
#include <iostream>

using namespace std;

int main (int argc, char * const argv[ ]) {
    cout << "Hello, World!" << endl;
    return 0;
}
```

Figure 1 : Programme C++ réalisant affichant "Hello World!"

Étape : Ouvrez votre éditeur de texte préféré et écrivez le code source ci-dessus.

Une fois ce programme créé et enregistré sur votre compte, vous allez devoir le compiler afin de transformer ce code source en exécutable (point que vous devez normalement maîtriser...).

Mais avant cela, il vous faut ouvrir un "terminal". Pour en trouver un, il faut aller le chercher dans la catégorie "Applications" de l'explorateur de fichiers, puis d'aller dans "Utilitaires". Après avoir lancé ce programme, vous devriez voir apparaître à l'écran un terminal identique à ce que vous avez déjà rencontré sous Linux / Unix.

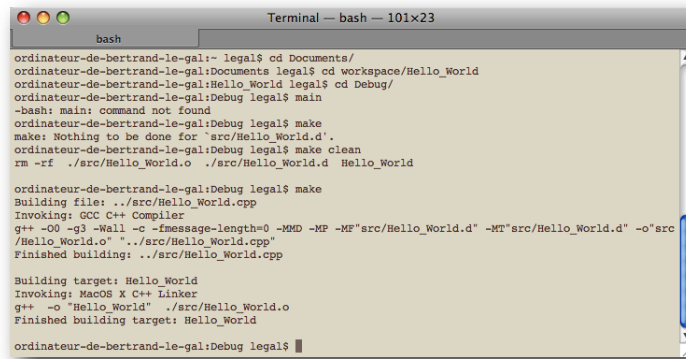

Figure 2 : Exemple de fenêtre terminale présente sous MacOS

Déplacez-vous à l'aide de la commande "cd" dans le répertoire contenant votre projet. Le fonctionnement du terminal est similaire à ce que vous avez appris sur Unix ou Linux. Les commandes sont aussi identiques.

Pour compiler votre programme et générer un exécutable, il vous suffit de taper la commande suivante dans le répertoire ou vous avez enregistré le fichier :

```
g++ main.cpp -o main
```

Figure 3 : Compilation du programme "Hello World!"

Vous remarquerez un point important : nous utilisons **g++** à la place de **gcc**. En effet, les 2 compilateurs réalisent de la compilation de code source, mais **g++** inclut les bibliothèques objets ce que ne réalise pas **gcc**.

Mise en oeuvre de l'environnement Eclipse

Dans un premier temps avant de créer des objets, nous allons prendre en main l'outil Eclipse qui va être l'environnement de conception à utiliser durant les séances de TP / Projet. Dans un même temps, nous allons essayer de vous familiariser avec l'environnement MacOS.

Introduction

L'environnement Eclipse est un IDE (Integrated Development Environment) qui est développé sous forme de logiciel libre par de grandes compagnies afin de fournir un environnement de programmation polyvalent (surport de nombreux langages), performant et multi-plateforme (disponible pour Windows, Linux, MacOS).

L'outil Eclipse est un environnement de développement développé à l'aide du langage Java dont il nécessite les bibliothèques pour fonctionner. Par contre l'outil Eclipse n'embarque aucun compilateur C/C++, il se contente d'utiliser ceux présents sur le système informatique hote.

Plus d'informations sur l'outil Eclipse et son installation peuvent être trouvées dans les pages référencées ci-dessous :

http://www.eclipse.org/

http://fr.wikipedia.org/wiki/Eclipse_(logiciel)

http://wascana.sourceforge.net/ (pour Windows)

Utilisation d'Eclipse

Avant de parler de l'utilisation d'Eclipse, vous allez déjà devoir lancer l'application... Pour trouver l'exécutable, il faut aller chercher dans la catégorie “Applications” de l'explorateur de fichiers, puis d'aller dans le répertoire “eclipse” puis cliquer sur l'application du même nom.

Création d'un projet

Une fois le logiciel Eclipse lancé, vous devez créer un projet afin de pouvoir commencer à développer votre premier exemple. Pour créer un projet (dans le répertoire que vous avez choisi lors du démarrage de l'outil), allez dans le menu “file” puis faites “New” et choisissez “C++ Project” ou *“New=>Project=>C++=>C++ Project”* en fonction de la version de l'outil.

Une fenêtre similaire à celle affichée ci-dessous doit alors apparaître.

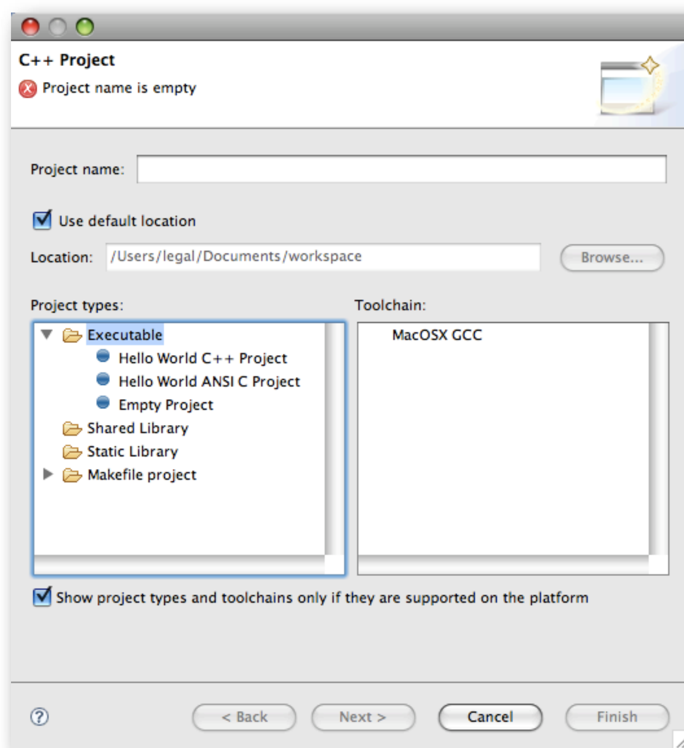

Figure 4 : Fenêtre d'invite permettant de créer un nouveau projet C++

À partir du choix que vous allez réaliser ici, vous allez pouvoir développer différents types de projet. Ce qui vous intéresse aujourd'hui c'est ce créer un exécutable à partir d'un programme écrit en langage C++. Pour cela, choisissez “Hello world C++ Project” et donnez un nom intelligible à votre projet “Hello_World” par exemple.

Remarque : l'avantage majeur de cette catégorie de projets vis-a-vis de ceux nommés “Makefile project” provient du fait que vous n'avez pas besoin d'écrire les makefiles, l'outil le fait pour vous...

Ensuite, cliquez sur le bouton “Next” afin de passer à l'étape suivante ou vous allez être invité à renseigner quelques champs. Cette partie sert à générer automatiquement les en-têtes des fichiers que vous créerez par la suite.

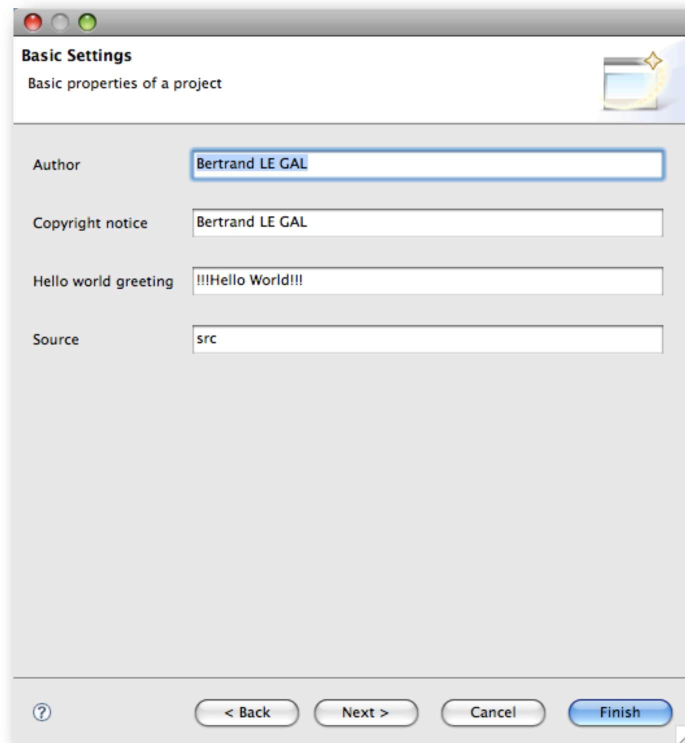

Figure 5 : Fenêtre de mémorisation des informations de l'auteur du projet

Une fois cette opération réalisée vous pouvez soit cliquer sur le bouton "Finish" soit sur le bouton "Next". Le premier terminera automatiquement la création de votre projet et le second vous permettra de configurer les options de compilation, de linkage, etc.

Le plus simple pour vous est pour le moment de faire "Finish".

Fenêtre d'édition et de projet

Après avoir réalisé l'étape de création du projet, vous devriez voir apparaître une fenêtre comparable à celle présentée ci-dessous. Une partie de l'environnement de travail que vous apprendrez à connaître est détaillée.

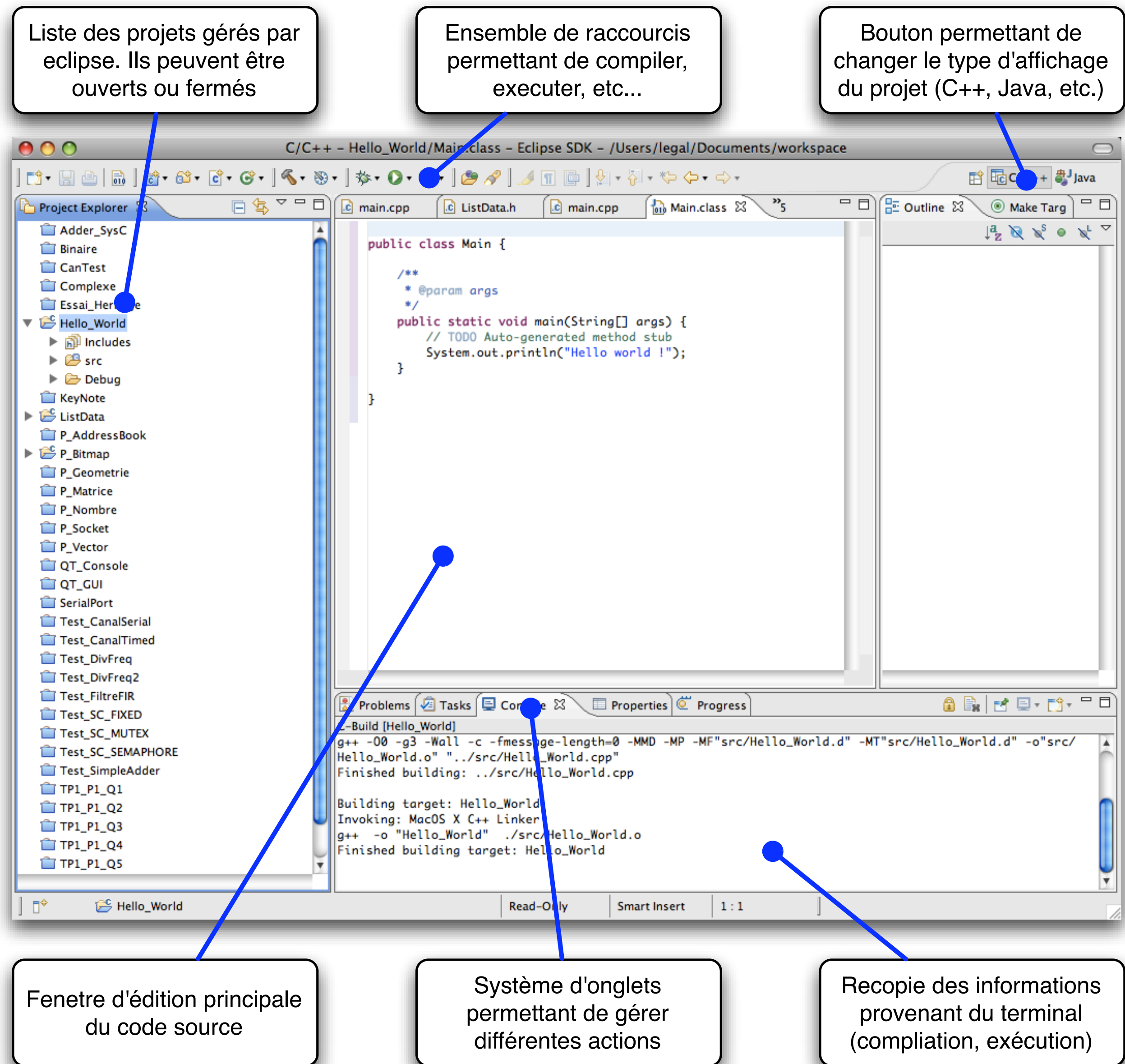

Figure 6 : Interface de l'outil Eclipse dans son mode développement C++

Dans la partie affichant le code source du programme vous devriez voir le code correspondant à la fonction “main” servant à afficher “Hello world” dans le terminal.

Maintenant que le projet a été créé et que le programme est prêt, nous allons voir comment le compiler et l'exécuter.

Compilation du code source

La phase de compilation d'un programme et des fichiers annexes sous eclipse est très simple ! En fonction des options que vous aurez activées, il se peut que l'environnement de développement essaye de recompiler les fichiers sources à chaque sauvegarde afin de

vous avertir au plus tôt des erreurs que vous avez commises. Ce comportement se valide ou se dévalide à partir du menu “Projet => Build automatically”.

Pour ceux préférant compiler les fichiers sources uniquement lorsqu'ils le souhaitent il faut aller dans le menu “Projet” et faire “Build All” ou plus simplement mémoriser la combinaison de touches “Pomme + B”.

Lorsque vous faites une demande compilation à Eclipse, ce dernier recense tous les fichiers qui composent votre projet et fait appel à “G++” afin que ce dernier puisse créer l'exécutable. Afin de mieux comprendre les erreurs de programmation que vous avez faites, il est possible d'observer les informations renvoyées par le compilateur dans la sous fenêtre en bas de l'écran.

Etape : Compilez le programme qu'Eclipse a créé automatiquement. Dans la fenêtre “terminal” vous devriez voir apparaître un message de type *“Finished building target: Hello_World”* signalant que tout s'est bien passé...

Exécution du code source

Afin de pouvoir lancer votre programme au travers d'Eclipse, vous devez configurer ce dernier afin qu'il soit capable de lancer votre exécutable. Pour réaliser cela, il vous faut cliquer à côté de l'icône verte (sur la petite flèche verticale) afin de voir un menu apparaître.

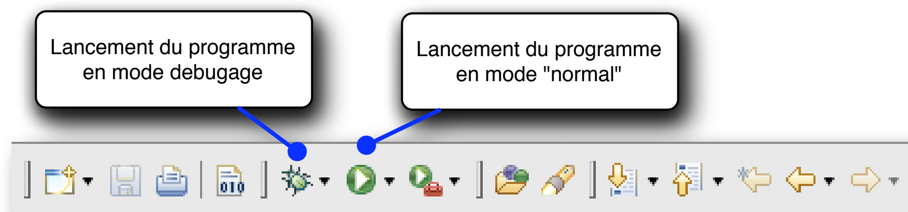

Figure 7 : Barre de menu pour l'exécution des programmes développés

Dans ce menu choisissez “Open Run Dialog”. Une nouvelle boite de dialogue s'affiche vous invitant à créer un profil d'exécution. Double cliquez sur “C/C++ Local application”, un onglet au nom de votre projet vient de se créer. Renseignez les champs et cliquez sur “Run” en bas de la fenêtre.

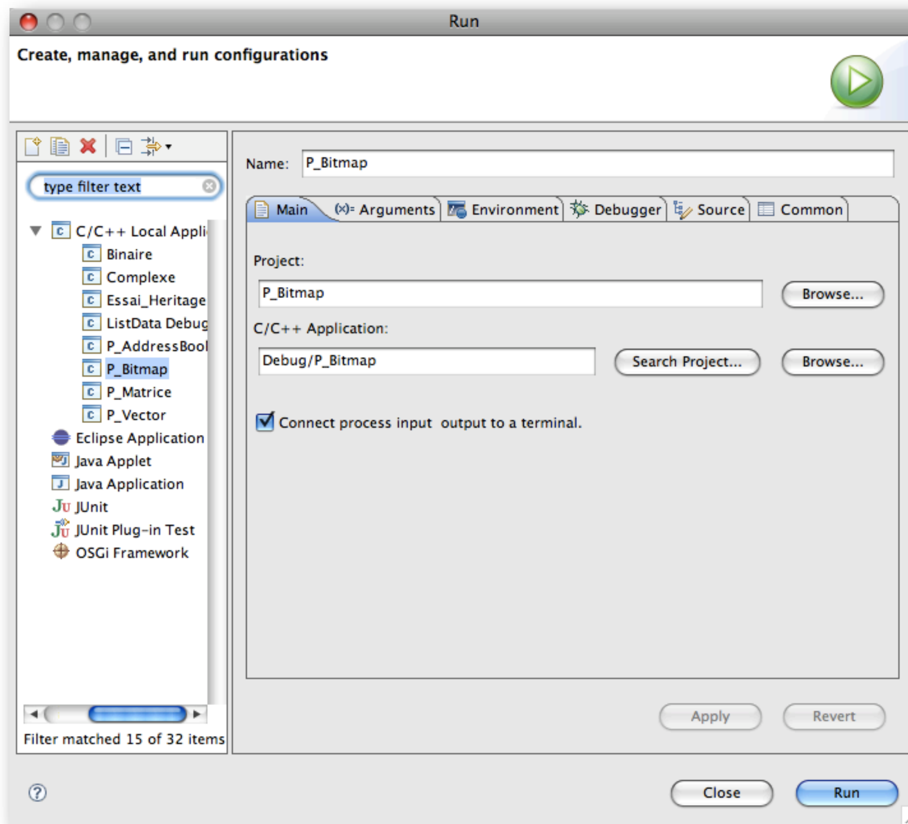

Figure 8 : Fenêtre de création pour les profils d'exécution

Remarque : Maintenant lorsque vous souhaitez relancer l'exécution de cette application, il vous suffit de faire comme pour faire apparaître la boite de dialogue, sauf que vous cliquerez dans le menu sur le profil portant le nom de votre projet.

Compilation & Exécution du programme (terminal)

Comme nous venons de le voir il est possible d'invoquer le programme que vous avez développé au travers de l'interface de l'IDE Eclipse. Cependant, cette manoeuvre n'est pas toujours très pratique lorsque le nombre de données a afficher dans par le programme est important. Nous allons donc aborder dans cette petite partie la manière de compiler et d'exécuter son programme à "l'ancienne".

Déplacez-vous (dans un terminal) à l'aide de la commande "cd" dans le répertoire contenant votre projet. Si vous n'avez pas modifié la configuration par défaut de l'outil, vous devriez pouvoir y accéder en tapant :

cd workspace/Hello_World

puis:

cd Debug

Là, vous êtes dans le répertoire ou l'outil Eclipse construit votre exécutable. Pour le lancer, faites comme sous linux et tapez "./Hello_World". Si vous souhaitez recréer un exé-

cutable manuellement, faites : “make clean” pour effacer les fichiers temporaires puis tapez “make”.

Remarque : il est important de noter que le fichier “makefile” que vous utilisez implicitement est compatible sous Linux et Windows que ce soit en mode terminal ou à l’aide de l’outil Eclipse.

Utilisation du debugger dans Eclipse

Maintenant nous allons rapidement étudier les capacités offertes par l’outil Eclipse dans le domaine de la mise au point (debugage). Pour pouvoir mieux comprendre des fonctionnalités offertes par l’outil vous allez modifier le programme affichant le message “Hello world” à l’écran afin de :

- Afficher 10 fois le message “X : Hello world !” avec X représentant la ième itération de la boucle d’affichage.

- Encapsuler l’affichage du message dans une fonction nommée Affichage(int numero);

Ces modifications ont pour objectif de vous faire mettre en oeuvre les options de debuggage (pas à pas, entrée dans une fonction, sortie de fonction et lecture ou modification dynamique de données en mémoire).

Étape : Modifiez dans un premier temps votre code source afin de répondre au besoin exprimé ci-dessus.

Maintenant nous allons utiliser le debugger. Pour lancer l'exécution de votre programme en mode debugage, il vous suffit de cliquer sur l'icône à côté de celle que vous avez utilisée tout à l’heure (ressemblant à un “bug”).

L’environnement passe alors dans un mode d’affichage adapté au debugage. Votre environnement doit à présent ressembler à l’image ci-dessous :

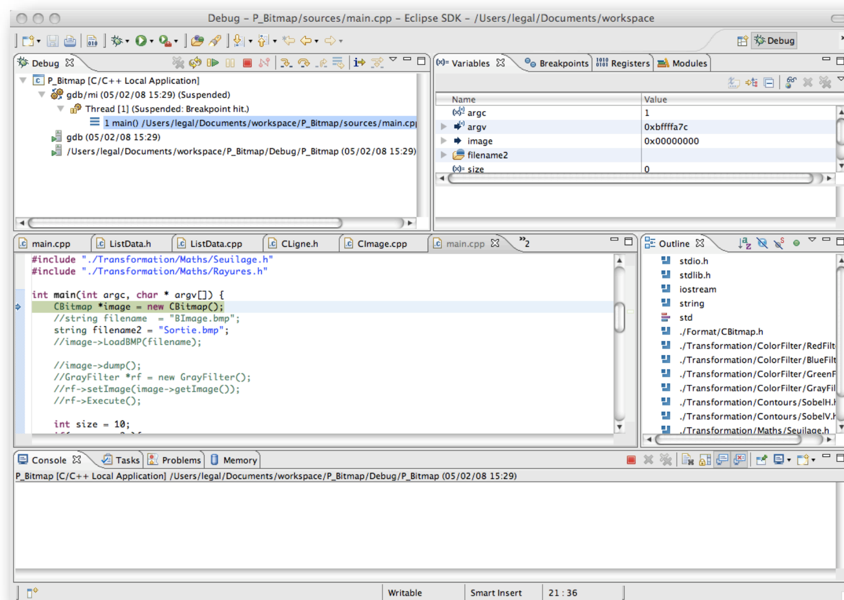

Figure 9 : Interface de l’outil Eclipse en mode debugage

Afin de piloter le debugger dans son exécution maîtrisée de votre programme, vous avez accès aux boutons suivants :

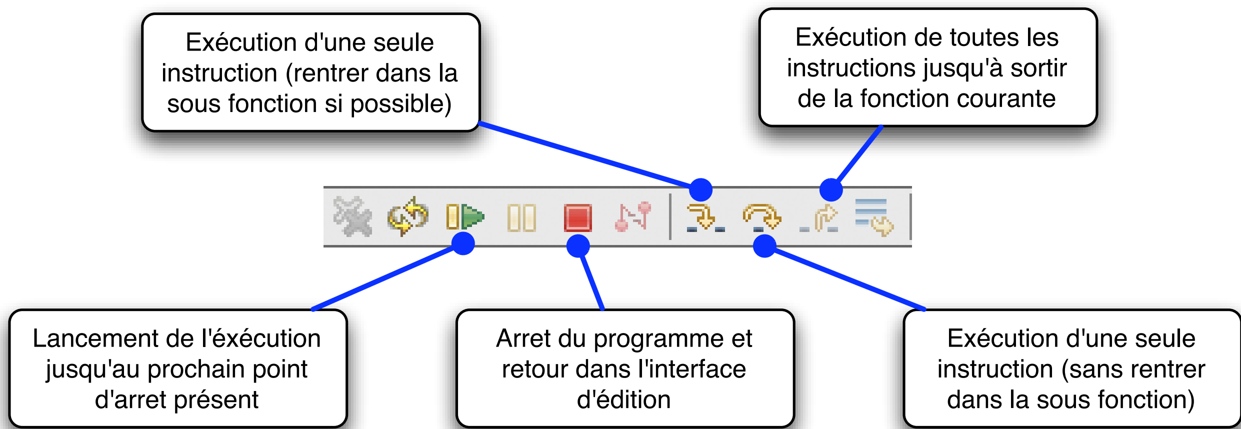

Figure 10 : Barre d'outils permettant de contrôler l'exécution du debugage

Vous retrouverez toutes ces commandes dans le menu "Run". Si vous regardez dans la fenêtre d'édition du code source, vous pourrez trouver des marques dans la marge du code. Ces signes dans la marge peuvent avoir plusieurs significations en fonction de leurs formes comme cela est expliqué dans la figure ci-dessous.

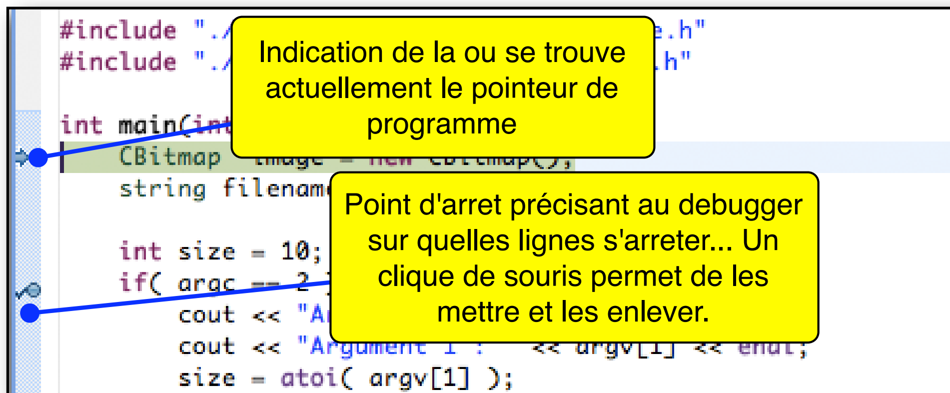

Figure 11 : Informations présentes pour spécifier le fonctionnement du debugger

<u>Étape</u> : À partir des indications qui viennent de vous être données, instrumentez votre code source afin de faire une exécution de ce dernier ligne par ligne et observez l'évolution de la mémoire.

<u>Étape</u> : Maintenant, prenez le temps d'essayer les différentes commandes présentes (step by step, function in, function out, les break point conditionnels ou non, etc.). Ce point est important car il vous fera gagner / perdre du temps durant votre projet...

Mise en oeuvre des assertions

Lors du cours de C++ dans la partie dédiée au rappels liés au langage C, nous avons vu la notion d'assertions. Ces techniques qui ont été développées permettent un gain de temps appréciable lors de la conception de système logiciels et particulièrement dans le développement objets.

Étape : reprenez les exemples développés dans le cours à propos des assertions et mettez-les en oeuvre afin de bien comprendre et maîtriser cette "technologie".

Remarque : les exemples sont disponibles dans les diapositives 62 à 65.

Développement de votre première classe

Nous allons dans cette partie créer un objet nommé **ListData** qui va nous permettre de mémoriser les valeurs entières fournies par l'utilisateur et sur lesquelles il souhaite effectuer des calculs de somme, moyenne, etc.

Ce premier exercice va vous permettre de vous familiariser avec l'environnement de développement Eclipse tout en créant votre premier objet. De plus dans cette partie vous allez apprendre à vous servir du debugger présent dans l'outil afin de comprendre exactement ce qui se passe en terme d'exécution du code source que vous avez écrit.

Mettez en oeuvre l'exercice que vous avez réalisé en cours (amphi) en écrivant le code C++ correspondant à la classe **ListData** (fichiers CPP et H) ainsi qu'en développant la fonction *main* modélisant un exemple d'utilisation de la classe.

Création d'une nouvelle classe

L'outil Eclipse propose de nombreuses fonctionnalités au développeur afin de leur faire gagner du temps. Avant de créer cette classe, n'oubliez pas de créer un nouveau projet afin de ne pas mélanger vos fichiers sources. Puis pour créer une nouvelle classe, allez dans le menu "**File**", choisissez la ligne "**New**" puis sélectionnez le choix "**Class**". Une fois ces actions effectuées, vous devriez voir apparaître la fenêtre présente ci-dessous.

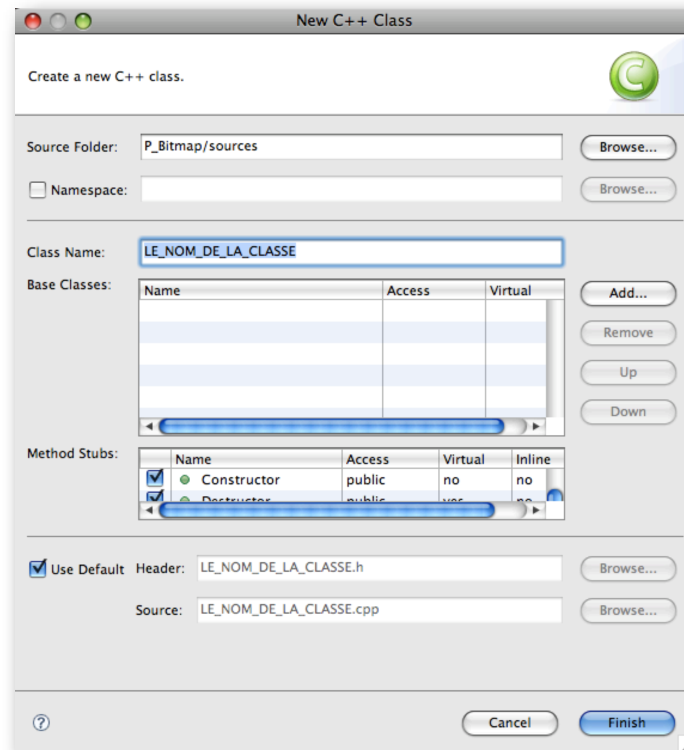

Figure 12 : Fenêtre d'aide à la création d'une nouvelle classe

Dans cette fenêtre vous devez spécifier à l'outil le nom de votre classe afin que ce dernier crée les fichiers CPP et H contenant les bonnes définitions. Une fois le champ "**Class Name**" renseigné (**ListData** dans votre cas), vous avez juste à cliquer sur "**Finish**".

Finalité de l'exercice

L'objectif de cette première partie est de vous permettre de réaliser votre premier objet et de vous en servir afin de comprendre la simplification du code source que permet le langage objets.

Rappel de l'énoncé de la partie traitée en cours

Nous souhaitons développer une classe **ListData** simplifiant les calculs liés aux tableaux en assurant les services suivants :

- Ajout de nouvelles données par l'utilisateur de la classe,
- Lecture du nombre de données fournies par l'utilisateur,
- Calcul de la somme des données fournies par l'utilisateur,
- Calcul de la moyenne des données fournies par l'utilisateur,
- Calcul de la valeur minimum contenue dans les données fournies,
- Calcul de la valeur maximum contenue dans les données fournies,
- Affichage des données rentrées à l'écran.

Suite au développement de la classe et des méthodes permettant de répondre aux besoins énoncés, vous aviez écrit un “programme” permettant de tester cette classe.

Étape : Écrivez la classe étudiée en cours et testez-la afin de mieux comprendre les principes fondamentaux de la programmation objets.

Extension des capacités

Maintenant que vous maîtrisez les bases du langage C++ nous allons modifier les classes utilisées afin de mettre en oeuvre le concept de données statiques. Pour cela vous allez devoir modifier votre classe afin de permettre de :

- Compter le nombre d’objets de type **ListData** crées depuis le lancement du programme main,
- dénombrer le nombre d’objets de type **ListData** actuellement en mémoire (non détruit par la méthode **delete**) à un instant “t”.

Étape : Modifiez votre classe (attributs et méthodes) et adaptez le programme de test afin de voir si vos modifications ont porté leurs fruits.

Développement du second objet

Sujet de l'étude - Nous allons nous intéresser au développement d'une classe permettant de représenter des nombres binaires codés sur N bits (avec N fixé à 32 au début de l'étude). L'objet permettant de modéliser des tels nombres sera ensuite étendu afin de fournir des services permettant d'assurer les opérations usuelles sur les nombres binaires : addition, multiplication, comparaison, etc...

Afin de simplifier l'étude et le développement dans un premier temps, nous allons considérer uniquement des nombres entiers positifs. Cette restriction permettra de simplifier les algorithmes à implémenter dans l'objet afin de permettre une bonne compréhension de la notion d'objet au travers du tutorial.

Nous avons spécifié que le premier objet que nous allons créer va permettre de modéliser sous forme binaire tout entier passé en paramètre. Dans le cas de nombre flottant, nous effectuerons un arrondi au nombre supérieur avant de réaliser la conversion.

La structure minimum de la classe "*Nombre_Binaire*" doit contenir :

- un tableau de 32 cases afin de mémoriser l'état de chacun des bits composant le nombre,
- un constructeur permettant de créer un objet tout en l'initialisant à 0 ou bien à l'aide de la valeur spécifiée par le concepteur.

La structure de notre objet de base est donc la suivante :

```
#include <stdio.h>
#include <stdlib.h>
#include <iostream>
#include <string>

using namespace std;

#ifndef Nombre_Binaire_H_
#define Nombre_Binaire_H_

class Nombre_Binaire
{
public:
      unsigned char data[32];

      Nombre_Binaire();
      Nombre_Binaire(int valeur);
      ~Nombre_Binaire();
};
#endif /*Nombre_Binaire_H_*/
```

Figure 13 - Déclaration de la classe Nombre_Binaire (Nombre_Binaire.h)

```
#include "Nombre_Binaire.h"

Nombre_Binaire::Nombre_Binaire(){
	cout << "Constructeur Nombre_Binaire()" << endl;
	for(int i=0; i<32; i++){
		data[i] = 0;
	}
}

Nombre_Binaire::Nombre_Binaire(int valeur){
	cout << "Constructeur Nombre_Binaire(int)" << endl;
	for(int i=0; i<32; i++){
		data[i] = valeur & 0x0001);
		valeur = (valeur / 2);
	}
}

Nombre_Binaire::~Nombre_Binaire(){
	cout << "Destructeur ~Nombre_Binaire()" << endl;
}
```

Figure 14 - Implémentation de la classe Nombre_Binaire (Nombre_Binaire.cpp)

Nous pouvons remarquer que des parties du code source sont communes (écriture des données dans le tableau), afin d'améliorer la maintenabilité de la classe, il vaut mieux factoriser les méthodes au comportement similaire. Nous obtenons alors l'implémentation suivante.

```
class Nombre_Binaire
{
public:
    unsigned char data[32];
    Nombre_Binaire();
    Nombre_Binaire(int valeur);
    ~Nombre_Binaire();
    void setValue(int valeur);
};
```

Figure 15 - Nouveau prototype de la classe

```
Nombre_Binaire::Nombre_Binaire(){
    cout << "Constructeur Nombre_Binaire()" << endl;
    setValue( 0 );
}

Nombre_Binaire::Nombre_Binaire(int valeur){
    cout << "Constructeur Nombre_Binaire(int)" << endl;
    setValue( valeur );
}

void Nombre_Binaire::setValue(int valeur){
    cout << "Methode setValue(int)" << endl;
    for(int i=0; i<32; i++){
        data[i] = (valeur & 0x0001);
        valeur = (valeur / 2);
    }
}
```

Figure 16 - Modifications dans l'implémentation des méthodes de la classe

Nous allons maintenant rajouter à cet objet des méthodes afin de pouvoir lire et écrire la valeur des bits contenus dans le tableau. La méthode permettant de lire les données sera nommée get() et celle permettant d'écrire sera nommée set(). Les prototypes des fonctions sont les suivants :

```
int  get(int position);
void set(int position, int valeur);
```

Figure 17 - Déclaration des méthodes get( ) et set( ).

L'implémentation de ces fonctions va se faire en assurant une vérification des accès que va réaliser l'utilisateur de la classe (contrôle de non débordement). Cette approche est un peu plus lente que sans sans contrôles, mais ces derniers permettent de visualiser les défauts de conception et d'utilisation très rapidement.

```
int Nombre_Binaire::get(int position){
    if(position < 32 ){
    return data[position];
    }
    return -1;
}

void Nombre_Binaire::set(int position, int valeur){
    if(position < 32 ){
    data[position] = (valeur > 0);
    }else{
        cout << "Depassement memoire => " << __FILE__;
        cout << ":" << __LINE__ << endl;
    }
}
```

Figure 18 - Implémentation des méthodes get() et set().

De plus ces instructions particulières pourraient appartenir à un bloc masquable à l'aide du pré-processeur afin de les supprimer lors de la génération du programme final.

```
void Nombre_Binaire::setValue(int valeur){
    cout << "Methode setValue(int)" << endl;
    for(int i=0; i<32; i++){
        set(i, valeur & 0x0001);
        valeur = (valeur / 2);
    }
}
```

Figure 19 - Modification de la méthode setValue().

À partir de là, il ne nous reste plus qu'à définir une méthode permettant d'afficher le nombre binaire à l'écran pour que l'on puisse vérifier le fonctionnement de notre classe. Cette méthode sera nommée dans le cadre de ce tutorial *show( )*, son code source vous est donné ci-dessous :

```
void Nombre_Binaire::show(){
    for(int j=0; j<32; j++){
        cout << get(31-j);
    } cout << endl;
}
```

Figure 20 - Implémentation de la méthode show( )

Nous avons maintenant une classe proposant les services minimums que l'on peut attendre d'elle. Nous allons maintenant la mettre en oeuvre afin d'utiliser notre premier objet.

Utilisation du second objet (instanciation)

Maintenant que nous avons décrit notre première classe, nous allons voir comment l'utiliser dans un programme écrit en C++. Pour pouvoir utiliser cette classe, nous allons devoir instancier la classe afin d'obtenir un objet.

Cette instanciation peut être réalisée de 2 manières différentes comme nous l'avons vu en cours : la manière statique (déclaration comme une variable) ou la manière dynamique (avec mise en oeuvre d'un pointeur).

Étape : Commencez par expérimenter la manière statique. Écrivez le programme main permettant de créer 2 objets de type Nombre_Binaire. Valider en même temps le bon fonctionnement de la classe développée. Compilez et exécutez le projet.

Remarque : Lors de l'exécution du programme les différents appels du constructeur de l'objet sont réalisés sans appel à la fonction **new**, il en va de même pour les appels au destructeur quand le programme se termine.

Étape : Continuez votre expérimentation en créant des objets de manière dynamique à l'aide de l'opérateur **new**. Ecrivez le programme main permettant de créer 2 objets de type Nombre_Binaire. Compilez et exécutez le projet.

Remarque : L'ensemble des objets instanciés dynamiquement doit être détruit explicitement, sinon ils provoqueront des fuites mémoire (mémoire allouée et jamais libérée) lors de l'exécution du programme et voir même après son extinction (suivant les OS).

Nous nous intéresserons dans la suite de ce tutorial uniquement aux objets instanciés dynamiquement de manière à ce que vous preniez les bonnes habitudes dès maintenant...

Restriction des fonctionnalités (droits d'accès)

Dans la classe que nous avons définie tout à l'heure, nous n'avons pas considéré la restriction des droits d'accès au tableau contenant notre nombre binaire. Ce point peut poser problème si l'utilisateur réalise des accès en dehors du tableau (l'erreur est humaine...). C'est le cas par exemple du code suivant qui peut être exécuté :

```
int main (int argc, char * const argv[ ]) {
    Nombre_Binaire *nb;
    nb = new Nombre_Binaire(10);
    nb->data[1024] = 1;     /* Erreur dans l'index    */
    nb->data[10]   = 2;    /* Erreur dans la donnée   */
    nb->show();
    delete nb;
    return 0;
}
```

Figure 21 - Problèmes d'accès aux données contenues dans l'objet

Pour éviter ce genre de problèmes, vous allez restreindre le droit des utilisateurs de la classe à accéder au tableau qui est un élément interne. Pour cela, mettez en oeuvre vos connaissances dans le domaine des droits d'accès.

Étape : Essayez maintenant de compiler le programme principal accédant directement au tableau. Que se passe t'il ?

Évolution de la classe (polymorphisme)

Maintenant que nous savons que notre classe est fonctionnellement correcte, nous allons ajouter différents constructeurs pour pouvoir créer un *Nombre_Binaire* à partir de la partie entière d'un nombre de type float / double par exemple en utilisant un arrondi supérieur du nombre passé en paramètre.

La possibilité offerte par le langage C++, afin de nommer des fonctions à l'aide du même nom mais avec des paramètres différents (type, nombre), s'appelle le polymorphisme.

Étape : Que devient la définition de la classe pour répondre à ces besoins ?

Étape : Ajoutez le code source nécessaire pour réaliser ces nouvelles fonctions.

Étape : Testez et validez les nouvelles fonctionnalités en modifiant votre programme main en conséquence.

Remarque : N'oubliez pas que tout bon programmeur C++ est feignant et essaye de ré-utiliser au maximum le code déjà développé... Cela réduit les performances globales du programme mais le rend bien plus évolutif. On ne peut bien évidemment pas gagner sur tous les tableaux.

Extension des capacités (héritage)

Nous avons réalisé un objet très basique permettant de représenter un nombre sous forme binaire. Nous souhaitons maintenant étendre ses capacités en augmentant le nombre des méthodes disponibles pour l'utilisateur de la classe.

Nous sommes dans un tutorial devant vous permettre de comprendre les concepts du monde objets, pour cela, nous allons décider d'utiliser l'héritage pour étendre les capacités de notre classe. Pour récupérer dans notre nouvelle classe tout ce qui a été développé dans *Nombre_Binaire*, nous allons déclarer un nouvel objet nommé *Binaire* comme suit :

```
#include "Nombre_Binaire.h"

#ifndef Binaire_H_
#define Binaire_H_

class Binaire : public Nombre_Binaire
{
public:
    Binaire();
    Binaire(int valeur);
};
#endif /*Binaire_H_*/
```

Figure 22 - Structure de la nouvelle classe nommée Binaire.

```
#include "Binaire.h"

Binaire::Binaire(){
    cout << "Constructeur Binaire()" << endl;
}

Binaire::Binaire(int valeur){
    cout << "Constructeur Binaire(int)" << endl;
    setValue( valeur );
}
```

Figure 23 - Implémentation des 2 constructeurs de la classe Binaire.

Le code source que nous venons d'écrire n'est pas compilable d'après le compilateur... En effet, ce denier nous dis que la méthode *setValue( )* n'est pas accessible car elle est privée. Cela est vrai si vous avez bien travaillé tout à l'heure... En effet si l'on regarde la définition de la classe *Nombre_Binaire*, il faut changer un point précis pour pouvoir utiliser la méthode *setValue( )* dans une classe héritant de la classe *Nombre_Binaire*.

Étape : Modifiez la classe *Nombre_Binaire* afin de résoudre le problème que vous venez de rencontrer. Compiler vos codes sources afin de valider vos modifications.

Afin de tester cette nouvelle classe, nous pouvons remplacer dans le programme Main toutes les références à la classe *Nombre_Binaire* et les remplacer par la classe *Binaire*. La classe *Binaire* héritant de toutes les méthodes et de tous les attributs de la classe *Nombre_Binaire*, cela ne modifie pas le reste du programme Main.

Étape : Lancez le nouveau programme main et observez l'ordre et le nombre des appels aux constructeurs des différentes classes. Que remarquez-vous ?

Nous allons maintenant définir les nouvelles méthodes que nous allons ajouter à la classe *Binaire*. Voici la liste des fonctionnalités à développer dans cette extension de la classe de base :

Ajouter une méthode permettant de réaliser une addition entre 2 nombres. Cette méthode additionnera 2 nombres *Binaire*s. Puis vous étendrez le travail réalisé à l'addition d'un nombre *Binaire* avec un entier. Les prototypes des méthodes seront les suivants :

```
void add(Binaire *nbre);
```

Figure 24 - Prototype de la méthode addition avec un objet Binaire.

```
void add(int nombre);
```

Figure 25 - Prototype de la méthode addition avec un nombre entier.

Étape : Écrivez les codes sources permettant de réaliser ces 2 nouvelles fonctionnalités. Ces 2 méthodes devront pouvoir être utilisées comme cela est démontré dans les exemples ci-dessous.

```
int main (int argc, char * const argv[]) {
    Binaire *n1 = new Binaire(2);
    Binaire *n2 = new Binaire(1);
    n1->add( n2 );
    n1->show();
    delete n1;
    delete n2;
    return 0;
}
```

Figure 26 - Mise en oeuvre et test de la fonctionnalité.

Étape : Compiler vos fichiers sources et essayer vos nouvelles méthodes.

Maintenant que ce nouveau service est validé, nous allons rajouter la possibilité d'additionner un nombre *Binaire* avec un nombre entier.

Pour ceux qui liront cette partie après avoir répondu à la précédente vous trouverez dans la figure suivante un exemple de codage astucieux de la méthode d'addition d'un nombre binaire avec une donnée entière.

```
void Binaire::add(int nombre){
    Binaire *n = new Binaire( nombre );
    add( n );
    delete n;
}
```

Figure 27 - Implémentation de la méthode addition avec un nombre entier.

Comme tout à l'heure vous remarquerez la forte réutilisation de ce qui a déjà été développé, testé et validé. En terme de performances (temps d'exécution), cette solution n'est

pas optimale, mais ce point est secondaire. Dans un développement objet, on cherche d'abord la maintenabilité et la flexibilité des classes avant les performances.

Bien évidement, si les performances étaient problématiques, nous optimiserions le code source de la classe *Binaire*, mais qui dit optimisation dit temps de développement, baisse de la flexibilité et argent investi. Cela sort de loin du cadre de ce cours.

Surcharger les opérateurs par défaut

Nous avons déclaré notre propre méthode pour réaliser une addition entre 2 nombres binaires. C'est déjà pas mal, mais on peut faire mieux... Nous allons surcharger la méthode "+" afin de pouvoir écrire naturellement une addition entre 2 nombres binaires. Pour cela, nous ajoutez les déclarations suivantes :

```
Binaire operator+(Binaire &n);
Binaire operator+=(Binaire &n);
```

Figure 28 - Nouvelles déclarations dans la classe.

Étape : Développez le code source permettant d'implanter les fonctionnalités que nous venons de décrire.

Exploitez cette nouvelle fonctionnalité et validez vos codes sources à l'aide du programme ci-dessous.

```
int main (int argc, char * const argv[]) {
    Binaire n1(2);
    Binaire n2(1);

    Binaire n3 = n1 + n2;
    n3.show();

    n1 += n2;
    n1.show();

    return 0;
}
```

Figure 29 - Programme permettant de tester les nouvelles possibilités.

Nous avons surchargé l'opérateur "+=", il faut savoir que tous les opérateurs standards sont surchargeables eux aussi.

S'entraîner un peu à la conception objet

Maintenant que vous avons abordé la conception de classes et les principaux apports du langage, il est temps pour vous de développer vous-même de nouvelles fonctionnalités. Vous trouverez ci-dessous un ensemble de propositions vous permettant de compléter votre initiation.

- Implémentation des fonctions logiques : égal, différent, plus grand, plus grand ou égal, plus petit et plus petit ou égal.
- Implémentation des fonctions mathématiques de base, soustraction, multiplication et division entre 2 nombres binaires ou 1 nombre binaire et un entier.
- Surcharges des opérateurs standards permettant d'incrémenter et de décrémenter un nombre binaire, ainsi qu'une surcharge de la méthode permettant d'afficher un nombre Binaire dans le flux de sortie **cout**.
- Ces fonctions seront implémentées pour pouvoir fonctionner entre 2 nombres Binaires ou un nombre *Binaire* et un entier.

Pour des raisons de temps, vous ne développerez pas ces nouvelles fonctionnalités dans le cadre de ce tutorial, mais pourquoi pas sur votre temps "libre".

Optimisation de la classe

La classe *Nombre_Binaire* que nous avons développée tout à l'heure est performante en terme de vitesse d'exécution, mais elle est aussi très gourmande en mémoire...

- Développez une classe nommée *LowMemory_Binaire* qui permet d'optimiser la consommation mémoire, en réalisant des masquages sur les octets du tableau pour obtenir la valeur des bits.
- Une fois que vous aurez développé les nouvelles méthodes *get( )* et *set( )* propres à cette classe vous modifierez l'héritage réalisé par la classe *Binaire* afin de bénéficier de vos nouveaux travaux.

Ce point vous permet de remarquer l'importance de la normalisation des interfaces (méthodes proposées par une classe). Il nous permet ainsi dans notre cas de modifier le coeur d'un objet sans avoir besoin de modifier le reste du programme...

Pour des raisons de temps, vous ne développerez pas ces nouvelles fonctionnalités dans le cadre de ce tutorial, mais pourquoi pas sur votre temps "libre".

Classe abstraite et méthodes virtuelles

Nous allons maintenant au travers d'un autre exemple illustrer un autre point du cours de langages-objets. Le point que nous allons aborder concerne la notion de classes abstraites et de méthodes virtuelles. Pour pouvoir mettre en évidence l'intérêt des méthodes virtuelles et des classes abstraites, nous allons nous baser sur un nouvel exemple que vous allez développer.

Étape : Développez dans un premier temps une classe nommée **Document** qui possède comme attributs : un titre, un auteur et une année (mettez en oeuvre la classe string afin de vous familiariser). Afin de vous simplifier la tâche, toutes ces informations sont obligatoires à la création de l'objet.

Vous mettrez en oeuvre des accesseurs afin permettre une lecture des attributs composants la classe ainsi que le destructeur permettant de libérer la mémoire allouée dynamiquement (objets de type string).

Remarque : Lors de l'appel des constructeurs et les destructeurs de toutes les classes vous afficherez un message expliquant ce qui se passe (c'est essentiel).

Étape : Dans votre classe **Document**, développez une méthode nommée **show( )**. Cette méthode sera en charge d'afficher à l'écran les informations sur le média (un attribut par ligne).

Étape : Mettez au point un petit programme de test permettant de vérifier le fonctionnement de votre classe.

Étape : Développez une classe **CD** qui hérite de **Document** et qui possède en plus un une durée en minutes. N'oubliez pas la méthode **show( )** qui est adapter intelligemment !

Étape : Testez votre classe.

Étape : Développez une classe **Livre** qui hérite de **Document** et qui possède en plus un nombre de pages. N'oubliez pas la méthode **show( )** qui est adapter intelligemment !

Etape : Testez votre classe.

Étape : Maintenant, créez un tableau de **Document** dans votre programme (tableau de pointeurs) et placez dedans des livres et des CDs. Une fois les documents rangés, demandez leur affichage. Que constatez-vous ? Trouvez la solution intelligente à ce problème !

Étape : Maintenant nous souhaitons supprimer la possibilité que possède l'utilisateur de créer des objets de type **Document** (car cela n'a pas de sens physique). Apporter les modifications nécessaires pour répondre à ce besoin. Vérifier que votre technique fonctionne.

Vous venez d'expérimenter les concepts de classes abstraites et de méthodes virtuelles. Ces concepts très puissants permettent de répondre à des besoins réels lors de la création de hiérarchies de classes complexes.

Les templates de classe

Un des derniers aspects majeurs de la programmation-objet appliquée au langage C++ est la notion de template de classe ou dit plus simplement de classe générique (la généricité cible la nature des données manipulées).

L'objectif de la généricité est de permettre la réutilisation de classe C++ et ce peu importe le type des données manipulées. Cette pratique permet par exemple de créer une fois pour toutes une classe FIFO et ce peu importe les données qu'elle devra contenir : nombres entiers, flottants, pointeurs, autres objets, etc.

Dans le cadre de ce dernier exercice, vous allez reprendre la classe **ListData** et la modifier afin de la rendre générique.

Étape : Reprenez votre cours et relisez la partie consacrée aux templates de classe afin de comprendre le concept initial. Puis modifier **ListData** afin de faire en sorte que cette dernière ne soit plus dépendante du type entier.

Étape : Maintenant que vous avez modifié la classe **ListData** écrivez un programme de test permettant de vérifier son utilisation possible avec différents types de données (dans le même programme.

Étape : Une classe appartenant à la STL et qui est assez largement utilisée grâce à sa généricité est la classe **vector**. Reprenez l'exemple fourni en cours et utilisez cette classe afin d'y stocker vos différentes listes. Testez le code que vous avez écrit. Quel est le problème qui apparaît, pourquoi ?

Conclusion finale...

Vous êtes maintenant arrivé à la fin de ce tutorial qui avait pour objectif de vous faire essayer les différents mécanismes liés à la programmation-objet. Cette expérience a dû vous permettre de comprendre l'avantage d'un tel découpage des problèmes et la facilité d'utilisation et de modifications des classes.

Maintenant, il vous reste le plus dur à acquérir, c'est-à-dire l'expérience... Jusqu'à maintenant les prototypes de classe et les fonctionnalités à développées vous ont été dictées. Dans le cadre du projet vous vous rendrez compte que le plus dur n'est pas nécessairement d'écrire le code C/C++ mais de savoir quoi écrire, de déterminer quels objets réaliser et pour faire quoi ?!