

TP 17 Allocation dynamique, Composition et Agrégation

Contexte et objectifs

L'objectif de ce TP est d'implémenter « La chasse aux papillons »© que vous avez conçu en TD.

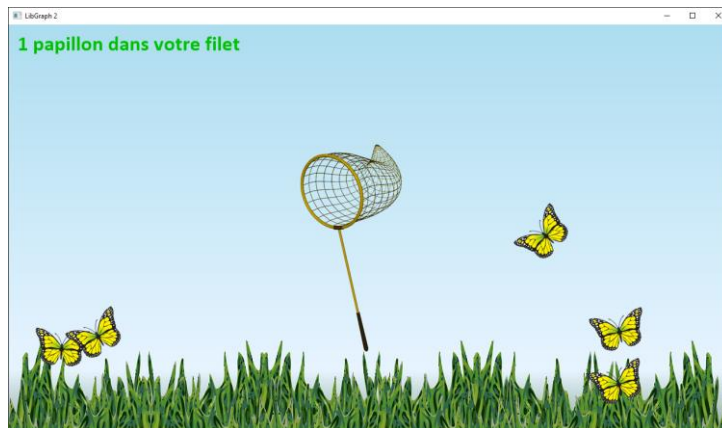


Figure 1 : Capture d'écran de l'application

Exercice 0 Récupération des fichiers et présentation

- Récupérez le fichier **VEtudiant.zip**.
 - Ce fichier contient différents répertoires :
 - **Résultat** : une version compilée de ce que vous devez obtenir.
 - **doc** : une documentation au format html, disponible par le fichier **doc\html\index.html**.
 - **TP 17** : Le répertoire du projet à compléter. Ouvrez la solution **TP 17.sln**.
 - La solution Visual Studio est correctement configurée.
 - Compilez et exécutez, vous devriez obtenir la fenêtre suivante :



Figure 2 : Capture d'écran de la première exécution

- Comme vous pouvez le constater, l'application n'est évidemment pas fonctionnelle puisque la rendre fonctionnelle est l'objet de ce TP !

0.1 Présentation du code existant

0.1.1 Le programme principal `main.cpp` du projet TP17

Le programme principal est déjà fait en grande partie. Vous n'avez le droit de modifier ce fichier qu'aux endroits indiqués par le commentaire `\todo`

Ce fichier fait appel aux classes nommées **CButterfly**, **CSwarm** et **CNet**, que vous aurez à écrire (pas maintenant). Par un procédé magique, le programme compile, même si les classes ne sont pas encore écrites. Cependant, pour que la magie opère, vous devez *scrupuleusement* suivre les indications de réalisation de l'exercice 1.

0.1.2 Fichiers `Butterfly.h`, `Butterfly.cpp`, `Swarm.h`, `Swarm.cpp`, `Net.h` et `Net.cpp`

Ces six fichiers sont le cœur de l'application car ils contiendront les déclarations et définitions des classes principales, que vous aurez à écrire dans l'exercice 1.

0.2 Présentation de la conception du programme

Le programme s'articule autour des classes **CButterfly**, **CSwarm** et **CNet** dont le diagramme UML est donné en figure 3. Une classe utilitaire **CVecteur2D** vous est donnée, elle correspond à la classe **CVecteur2D** déjà vu au semestre 1. Les collections de papillons peuvent être réalisées à l'aide de la classe de la STL **set**, qui fonctionne comme les classes **list** et **vector** que vous connaissez.

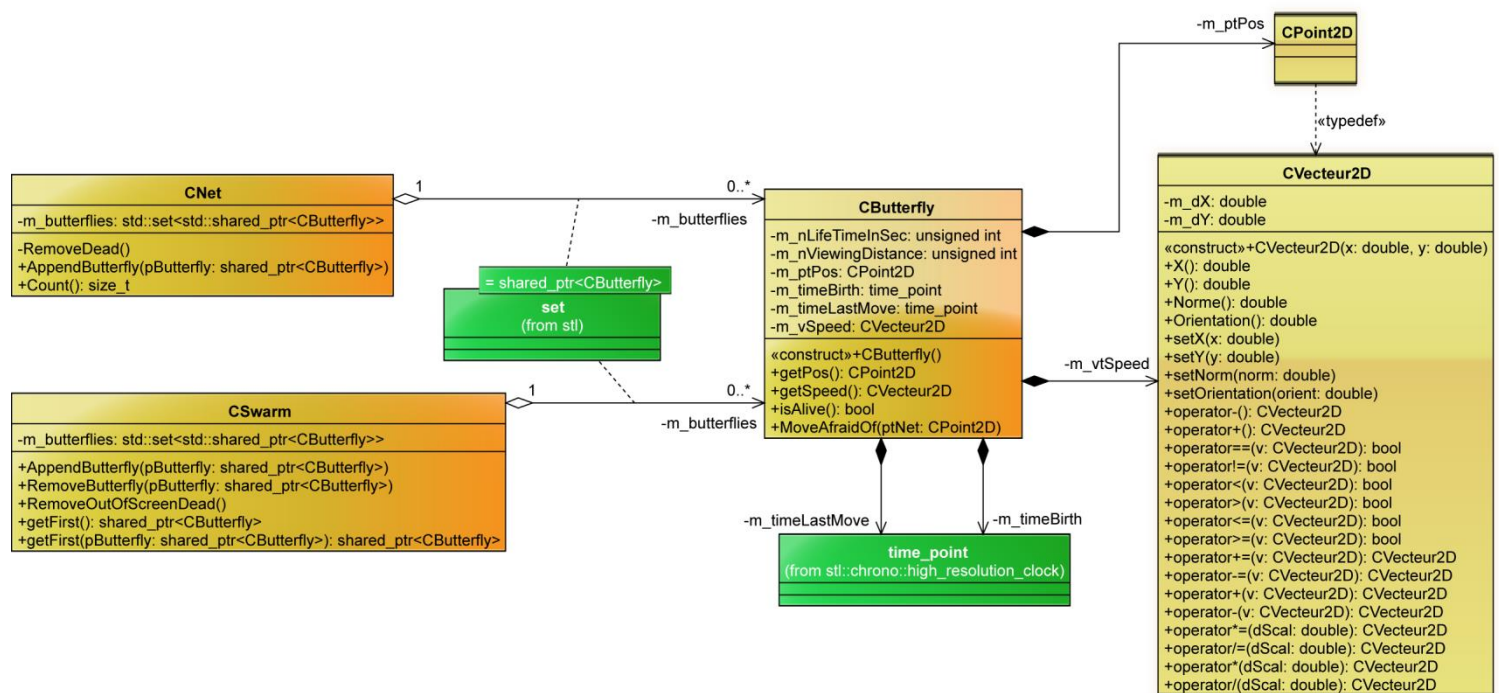


Figure 3 : Diagramme de classes de l'application

Exercice 1 Travail à réaliser

1.1 La classe CButterfly

Pour implémenter cette classe vous aurez besoin de gérer le temps d'exécution et des nombres aléatoires. Lisez les sections suivantes avant tout !

1.1.1 Gestion du temps en C++

Le C++ propose dans la bibliothèque `<chrono>` un ensemble de fonctions et classe de gestion des dates et du temps. Dans le cas de « La chasse aux papillons »©, le temps intervient pour la durée de vie des papillons et pour la gestion de l'animation. La fonction `CButterfly::isAlive` doit mesurer le temps entre le moment de la création de l'objet `CButterfly` et le moment de l'appel de la fonction. Le moment de création doit être stocké dans la donnée membre `CButterfly::m_timeBirth` par l'instruction :

```
m_timeBirth = std::chrono::high_resolution_clock::now();
```

La méthode `std::chrono::high_resolution_clock::now()` précédente retourne un objet de type `std::chrono::high_resolution_clock::time_point` qui correspond, comme son nom de classe l'indique, à un point dans le temps.

Pour mesurer un temps écoulé, il suffit de réaliser une différence entre deux `time_point`, ce qui retourne un objet de type `std::chrono::duration`. On peut choisir d'exprimer la durée dans n'importe quelle unité de temps. Par exemples, `end` et `start` sont tous deux des `time_point` :

```
std::chrono::duration<double, milli> durEnMillisecondes = end - start;
std::chrono::duration<double> durEnSecondes = end - start;
```

Pour retrouver la valeur numérique de la durée, il suffit d'appeler la fonction `count()` de l'objet `duration` :

```
durEnMillisecondes.count(); //Durée en millisecondes
durEnSecondes.count(); //Durée en secondes
```

Enfin, s'il s'agit de comparer des durées, des classes pratiques existent permettant d'exprimer simplement une durée :

```
std::chrono::nanoseconds;
std::chrono::microseconds;
std::chrono::milliseconds;
std::chrono::seconds;
std::chrono::minutes;
std::chrono::hours;
```

Ainsi, pour savoir si une durée entre deux points de temps est supérieure à 4 minutes 30 secondes, il suffit d'écrire :

```
if (end - start > std::chrono::minutes(4) + std::chrono::seconds(30))
    //oui
else
    //non
```

1.1.2 Génération de nombres aléatoires

Dans les premières spécifications du C++, il existait la fonction `rand` retournant un nombre pseudo-aléatoire compris entre 0 et `RAND_MAX`. Cette fonction souffre de nombreux problèmes

statistique (qui sortent du cadre de ce cours) et a été définie comme obsolète dans les spécifications actuelles du C++. Il est recommandé de ne plus l'utiliser ce qui signifie qu'il se pourrait bien qu'elle disparaisse un jour du C++. La génération des nombres aléatoires en C++ se fait maintenant en utilisant différentes classes permettant de contrôler plus efficacement les statistiques du tirage aléatoire. Ces classes sont déclarées dans la bibliothèque `<random>`.



Le principe est d'utiliser un moteur de génération de nombre pseudo aléatoire puis de lui demander un nombre aléatoire suivant la distribution statistique souhaité. Cela peut paraître compliqué, mais les spécifications C++ définissent des classes « par défaut » pour ceux qui veulent aller vite ☺ ! Nous utiliserons donc la classe `std::default_random_engine` comme moteur de génération et la classe `std::uniform_int_distribution` comme distribution. Ainsi pour trouver un nombre entier non signé aléatoirement entre 10 et 20, il suffit de taper :

```
std::default_random_engine gen;
unsigned int val = std::uniform_int_distribution<unsigned int>{10, 20}(gen);
```

Cependant ce code a un inconvénient : le nombre, bien qu'aléatoire, sera toujours le même. La création de l'objet **gen** initialise le moteur de génération. Le terme « pseudo » signifie que l'aléatoire ne l'est pas vraiment. Les algorithmes qui génèrent les nombres pseudo aléatoires sont totalement déterminés, et l'aléatoire n'existe pas dans l'électronique de l'ordinateur (sauf en cas de perturbation électromagnétique extérieure). Pour les mêmes entrées, un algorithme produira toujours les mêmes sorties. Il faut alors que l'initialisation du moteur aléatoire se fasse de façon différente à chaque fois qu'il est initialisé. Le constructeur des moteurs de génération de nombres pseudo aléatoires ont un paramètre nommé « graine » qui détermine un nombre de départ de l'algorithme. Cette graine devrait être différente à chaque fois. Pour cela, nous allons utiliser une machine aléatoire codée dans la classe `std::random_device`. Ainsi, le code parfait devient :

```
std::random_device rd;
std::default_random_engine gen(rd());
unsigned int val = std::uniform_int_distribution<unsigned int>{10, 20}(gen);
```


1.1.3 À vous de jouer

-  Fort de vos nouvelles connaissances et muni de votre sujet de TD, implémentez la classe **CButterfly** dans les fichiers **Butterfly.h** et **Butterfly.cpp** telle que vu en TD et décrite sur le diagramme UML de la figure 3.
-  Pour vérifier que votre classe fonctionne, modifier le fichier **main.cpp** de façon à créer un papillon et à l'afficher. À l'exécution, le papillon doit apparaître à l'écran puis tomber lorsqu'il est mort.

1.2 La classe CSwarm

La relation d'agrégation multiple entre l'essaim et les papillons peut être implémentée en utilisant une collection de type `set`. L'intérêt d'un `set` par rapport à une liste est que l'ordre n'est pas important dans un `set`. Cela permet à l'implémentation d'optimiser les recherches d'éléments dans cet ensemble. Or, de par le principe du jeu, les papillons ne sont pas ordonnés entre eux, et ils seront fréquemment recherchés lorsqu'ils entreront dans le filet.

Le type `set` fonctionne comme le type `list`, à ceci près qu'il n'y a pas de méthodes `push_back` ni `push_front`. Pour insérer un élément, il faut faire appel à la méthode `insert`.

-  Implémentez la classe **CSwarm** dans les fichiers **Swarm.h** et **Swarm.cpp** telle que vu en TD et décrite sur le diagramme UML de la figure 3.

- ➡ Modifier le programme principal pour créer l'essaim et faire en sorte que toutes les 3 secondes un nouveau papillon naisse dans l'essaim. Référez-vous aux commentaires dans le code.

1.3 La classe CNet

- 📄 Implémentez la classe **CNet** dans les fichiers **Net.h** et **Net.cpp** telle que vu en TD et décrite sur le diagramme UML de la figure 3.
- ➡ Modifier le programme principal pour créer le filet et faire en sorte que si un papillon se trouve à moins de 25 pixels du filet, il soit capturé (retiré de l'essaim et ajouté au filet). Référez-vous aux commentaires dans le code.