INF1600 - H25

Travail pratique 3

Lien entre C++ et assembleur x86.
Appels de fonction et récursivité.

Département de Génie Informatique et Génie Logiciel
Polytechnique Montréal

Table des matières

1 Introduction et sommaire

Ce travail pratique vise à explorer les relations entre le langage C++ et l'assembleur. Vous serez amené à étudier la taille occupée en mémoire par les classes et l'impact de l'alignement sur cette taille. Par ailleurs, vous analyserez le mécanisme permettant à C++ de gérer plusieurs fonctions portant le même nom (name mangling). Vous examinerez également l'effet des appels de fonctions récursives sur le fonctionnement du programme. Enfin, vous devrez mobiliser l'ensemble de vos connaissances en assembleur x86 IA-32 pour déchiffrer un code écrit en C++.

1.1 Remise

Voici les détails concernant la remise de ce travail pratique :

- Méthode : sur Moodle, une **seule remise par équipe**. **Seules des équipes de deux (2) étudiants sont tolérées**, sauf avis contraire.
- Format: un dossier compressé, incluant les sources de vos programmes en C++ et en assembleur
- Fichier attendu, il suffit de faire **`make zip`** pour générer le répertoire suivant qui est à déposer su Moodle.

  INF1600_H25_TP3_REMISE.zip

- Fichiers à remplir :

```
INF1600_TP3
└── /src_TODO
    ├── complex_constructor.s
    ├── complex_modulus.s
    ├── complex_operator_Mul.s
    ├── complex_operator_Plus.s
    └── mandelbrot_set.s
```

- Contrainte : Il est **INTERDIT** d'utiliser la librairie **std ::complex**. Il faut le coder soi-même!

Date de remise : Dimanche 16 mars 23h59

1.2 Barème

Fichier	**Note**
complex_constructor.s	/ 10
complex_operator_Plus.s	/ 10
complex_operator_Mul.s	/ 15
complex_modulus.s	/ 15
mandelbrot_set.s	/ 50
Total	/100

1.3 Prérequis

Ce TP fait le lien entre le C++. On utilise aussi la librairie graphique Raylib. Nous avons donc besoin d'un compilateur C++ 32 bits et des dépendances en lien avec OpenGL et le système de fenêtrage X11. Ces dernièeres doit supporter la compilation en 32 bits qui est utilisée dans le cadre du cours.

1.3.1.1 Installer les dépendances :

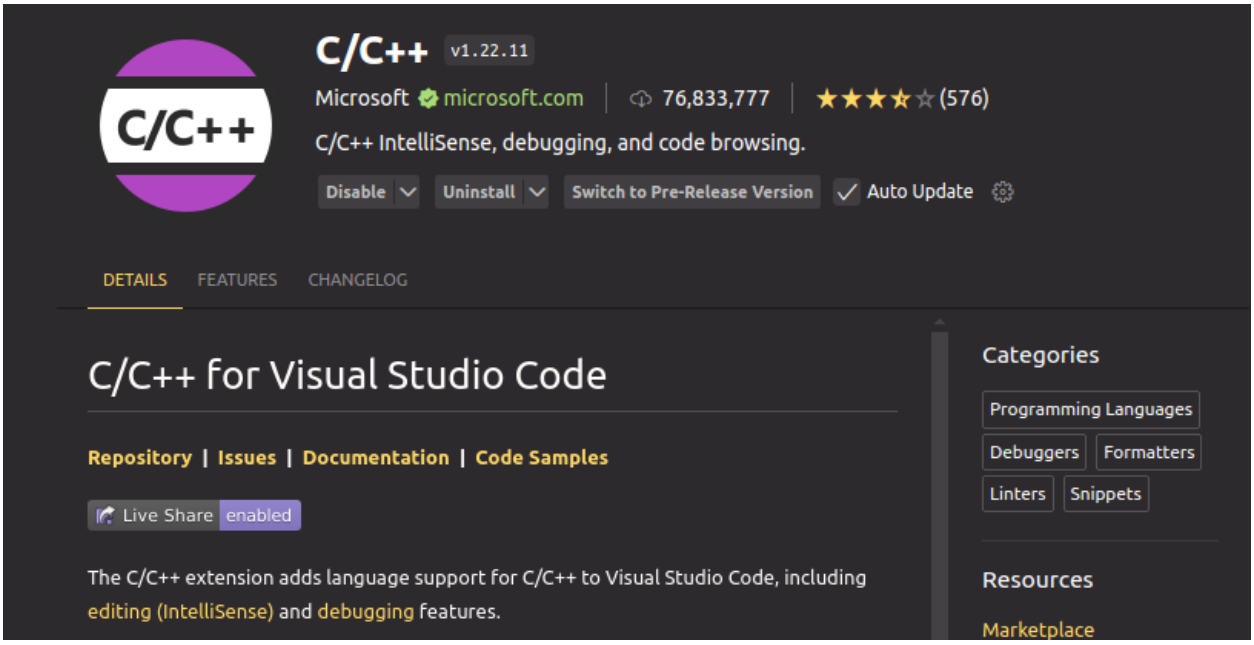
CentOs / Fedora

```
sudo dnf install libstdc++-devel libstdc++-devel.i686
sudo dnf install mesa-libGL.i686 mesa-libGLU.i686 libX11.i686 libXcursor.i686
libXrandr.i686 libXi.i686 libXinerama.i686 libXxf86vm.i686 libX11-devel.i686
libXrandr-devel.i686 libXi-devel.i686 libXinerama-devel.i686 libXcursor-devel.i686
alsa-lib-devel.i686
```

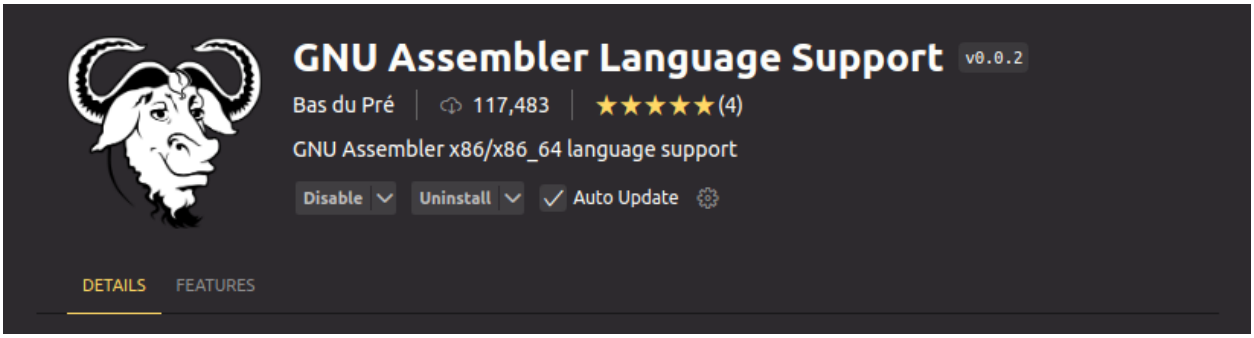
Ubuntu

```
sudo apt update && sudo apt install libgl1-mesa-glx:i386 libglu1-mesa:i386 libx11-6:i386 libxcursor1:i386
libxrandr2:i386 libxi6:i386 libxinerama1:i386 libxxf86vm1:i386 build-essential gcc-multilib g++-multilib
libsdl2-dev libglfw3-dev libglew-dev libopenal-dev libfreetype6-dev
```

1.3.1.2 Installer l'extension C/C++ sur VsCode (normalement déjà fait)



1.3.1.3 Installer l'extension GNU Assembler Language Support sur VsCode (normalement déjà fait)



1.4 Compilation et exécution :

En ligne de commande :

`make` pour compiler

`make run` (ou ./mandelbrot) pour exécuter

Avec les outils VsCode :

`F5` pour compiler et déboguer

`Shift + F5` pour compiler et exécuter sans débogage

2 Visualisation de l'ensemble de Mandelbrot

2.1 L'ensemble de Mandelbrot

Une fractale est un objet géométrique dont la structure se répète à différentes échelles, et ce, de façon infinie. C'est donc un exemple parfait de fonction récursive qui existe naturellement dans le domaine mathématique. Notre objectif : le générer avec du code!

L'ensemble de Mandelbrot (figure 1) est une fractale définie par récurrence comme suit :

$$\begin{cases} z_0 = C \\ z_{n+1} = {z_n}^2 + z_0 \end{cases}$$

où C et z sont des nombres complexe.

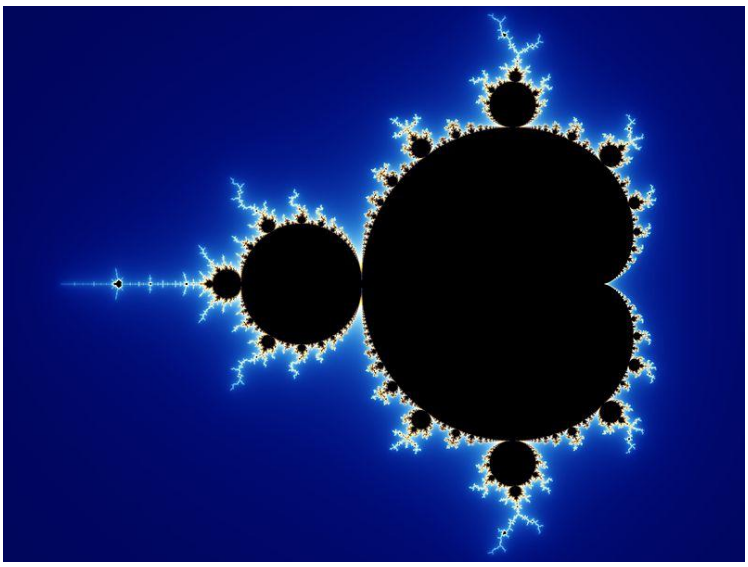


Figure 1 : Visualisation de l'ensemble de Mandelbrot
https://en.wikipedia.org/wiki/Mandelbrot_set

Un nombre complexe est dans l'ensemble lorsque la suite z_n est bornée en module. En d'autres mots, lorsqu'on itère sur cette suite et que le module résultant ne tend pas vers l'infini, le nombre complexe est dans l'ensemble.

$$\begin{gathered} z_0 = C \\ z_1 = {z_0}^2 + C \\ z_2 = {z_1}^2 + C \\ \dots \end{gathered}$$

On peut donc compter le nombre d'itération que ça prend avant que la suite diverge. Pour que ça ne prenne pas un temps infini, on va se limiter à un maximum de 100 itérations. Il suffit donc d'associer une couleur au nombre d'itération et on peut faire apparaître l'image de la figure 1!

2.2 Nombres complexes

Un nombre complexe est défini sous la forme $x + iy$ où i est le nombre imaginaire tel que $i^2 = -1$ (figure 2).

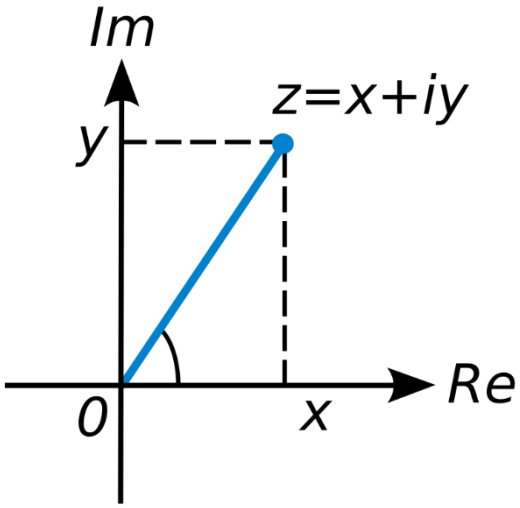


Figure 2 : Représentation graphique des nombres complexes
https://fr.wikipedia.org/wiki/Nombre_complexe

Ce n'est pas le but de ce TP de maîtriser les nombres complexes. Il faut simplement comprendre qu'on peut utiliser un plan cartésien en 2D pour les représenter. En résumé, la partie réelle est associée à l'axe des x, alors que la partie imaginaire est associée à l'axe des y. Il est donc possible de représenter les points de l'ensemble de Mandelbrot sur un graphe.

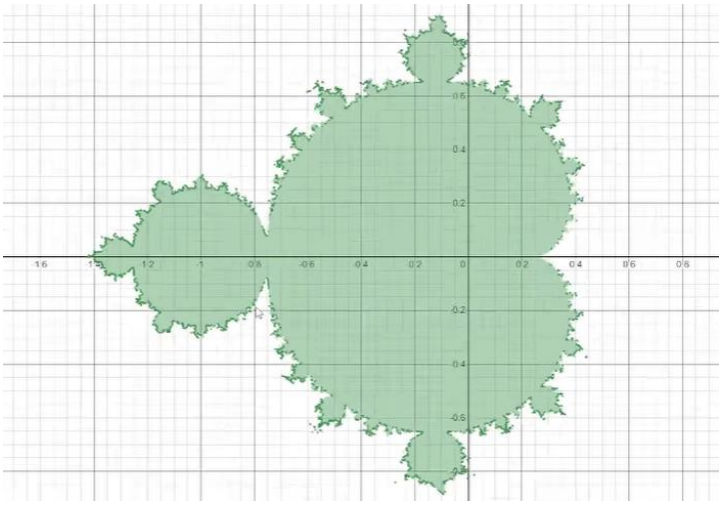


Figure 3 : Ensemble de Mandelbrot sur un graphe
https://math.stackexchange.com/questions/4401337/making-the-mandelbrot-fractal-in-desmos-online-graphing-calculator

2.3 Notion d'infographie

Afin de faire apparaître la fractale, nous allons itérer sur une série de coordonnées. Ces coordonnées proviennent de notre écran. En effet, un tableau 2D de coordonnées peut se traduire en un tableau 2D de pixels. Nous utiliserons la librairie Raylib pour l'affichage.

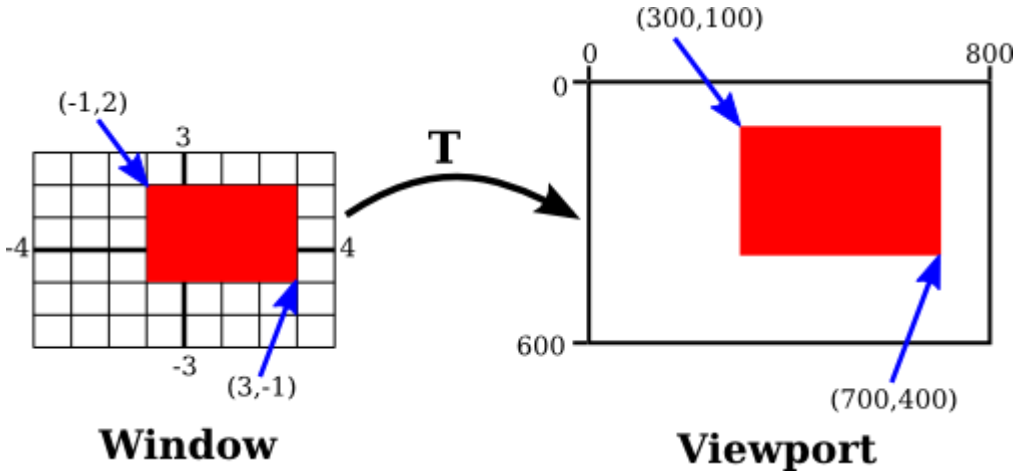


Figure 3 : Tableau de coordonnées en tableau de pixels
https://math.hws.edu/eck/cs424/graphicsbook-1.2/c2/s3.html

3 TLDR

Bref, nous allons itérer sur un tableau 2D de coordonnées *x, y* où chaque coordonnée est un pixel. Ensuite, nous allons transformer ses coordonnées en nombre complexe tel que *x* est la partie réelle et *y* est la partie imaginaire. Nous allons ensuite vérifier si ce nombre complexe vérifie la relation récursive de Mandelbrot. Ceci nous permettra d'afficher la fractale[1]!

Les nombres complexes s'additionnent, se multiplient et ont un module. Bien sûr, ces opérations se calculent différemment que les nombres réels. Afin de faciliter les opérations sur les nombres complexes, nous allons créer notre propre classe **Complex**. Ensuite, nous allons créer notre propre fonction récursive de Mandlebrot.

What's the catch? Le tout doit être en assembleur! Votre objectif est donc de traduire les fonctions et méthodes suivantes en assembleur.

[1] Itérer sur un tableau 2D et appeler une fonction récursive à chaque coordonnée n'est pas une approche optimale et n'est utilisée qu'à des fins académiques.

4 TODO

Nous allons donc nous créer notre propre classe **Complex** et surcharger les opérateurs d'addition et multiplication. Nous allons aussi créer notre méthode qui calcule le module d'un nombre complexe. Enfin, nous allons coder la définition de l'ensemble de Mandelbrot sous la forme d'une fonction récursive.

Attention! Il faut éviter la duplication de code! Rappelez vous qu'on peut faire des appels de fonction C++ en assembleur en utilisant le **mangling** de la fonction.

Certaines instructions de la FPU sont nécessaires. **Lisez bien les commentaires de chaque méthode dans leur fichier respectif** car on vous guide sur leur utilisation.

Selon votre ordinateur, il est très possible que l'affichage soit lent car notre calcul consommera beaucoup de ressources. Aucun point ne sera retiré en raison de l'affichage.

Vous devez compléter les fichiers assembleur (.s) dans le répertoire `INF1600_TP3 > src_TODO`

5 Classe Complex

La définition de la classe **Complex** se trouve dans `INF1600_TP3 > include > complex.hpp`

Pour vous guider, vous trouverez également une implémentation en C++ dans `INF1600_TP3 > src_reference_cpp > complex.cpp`

Voici la déclaration de la classe Complex

```
class Complex {
    float real; // partie réelle     (x)
    float imag; // partie imaginaire (y)

   public:
    Complex(float x, float y);
    float realPart() const; // accesseur de la partie réelle
    float imagPart() const; // accesseur de la partie imaginaire
    float modulus() const;  // retourne le module du nombre complexe
};

Complex operator+(const Complex& c1, const Complex& c2);
Complex operator*(const Complex& c1, const Complex& c2);
```

Voici une description ainsi que l'ordre suggéré de complétion des méthodes.

5.1 Complex()

INF1600_TP3 > src_TODO > complex_constructor.s

Constructeur de la classe Complex. Il initialise les attributs `real` et `imag` par les arguments `x` et `y` fournis en paramètre.

```cpp
Complex::Complex(float x, float y) : real(x), imag(y) {}
```

5.2 realPart()

INF1600_TP3 > src_TODO > complex_realPart.s

Cette méthode vous est déjà fournie. Elle retourne la partie réelle du nombre complexe.

```cpp
float Complex::realPart() const { return real; }
```

5.3 imagPart()

INF1600_TP3 > src_TODO > complex_imagPart.s

Cette méthode vous est déjà fournie. Elle retourne la partie imaginaire du nombre complexe.

```cpp
float Complex::imagPart() const { return imag; }
```

5.4 operator+()

INF1600_TP3 > src_TODO > complex_operator_Plus.s

La surcharge de l'opérateur + pour additionner 2 nombres complexes avec la formule :

$$(a + ib) + (c + id) = (a + c) + i(b + d)$$

L'opérateur plus doit appeler un constructeur pour créer le nombre complexe résultat de l'addition (retour de la fonction).

```
Complex operator+(const Complex& c1, const Complex& c2) {
    return Complex(c1.realPart() + c2.realPart(), c1.imagPart() + c2.imagPart());
}
```

5.5 operator*()

INF1600_TP3 > src_TODO > complex_operator_Plus.s

La surcharge de l'opérateur * pour multiplier 2 nombres complexes avec la formule :

$$(a + ib)(c + id) = (ac - bd) + i(ad + bc)$$

L'opérateur mul doit appeler un constructeur pour créer le nombre complexe résultat de la multiplication (retour de la fonction).

```
Complex operator*(const Complex& c1, const Complex& c2) {
       return Complex(c1.realPart() * c2.realPart() - c1.imagPart() * c2.imagPart(),
                      c1.realPart() * c2.imagPart() + c1.imagPart() * c2.realPart());
      }
```

5.6 modulus()

`INF1600_TP3 > src_TODO > complex_modulus.s`

Retourne le module du nombre complexe avec la formule suivante :

$$\sqrt{x^2 + y^2}$$

```
float Complex::modulus() const {
    return std::sqrt(realPart() * realPart() + imagPart() * imagPart());
}
```

6 Fonction récursive de Mandelbrot en C++

INF1600_TP3 > src_TODO > mandelbrot_set.s

La fonction récursive de Mandelbrot

```
const int maxIterations = 100;
const float escapeRadius = 2.0;

int mandelbrotSet(const Complex& z, const Complex& c, int count) {
    if (z.modulus() >= escapeRadius) return count;  // cas de base 1
    if (count + 1 > maxIterations) return count;    // cas de base 2
    return mandelbrotSet(z * z + c, c, ++count);    // cas recursif
}
```

7 main()

INF1600_TP3 > main.cpp

Fonction main()

Le fichier main.cpp se divise en 4 parties :

Partie 0 : Compilation. Une fonction pour tester la compilation. Si le projet de départ compile, il faut commenter l'appel à drawInit() pour commencer le TP.

Partie 1 : Vide. Un espace vide est laissé pour vous aider à tester vos fonctions avec vos propres appels.

Partie 2 : Tests. La fonction runTests() roule une série de tests pour chaque fonction à coder. Le résultat des tests s'affichera dans le terminal.

Partie 3 : Affichage. La fonction drawMandelbrot() vous permettra de faire afficher la visualation de Mandelbrot que vous avez codé. Il est recommandé de faire passer tous les tests avant de décommenter la fonction. Rappel : l'affichage n'est pas une condition nécessaire à la réussite du TP.

8 Résultat final :

L’ensemble de Mandelbrot généré de façon programmatique!