Cette étude nous permet de comprendre comment effectuer une communication entre deux programmes réalisés dans des langages différents. L'objectif à terme est de créer des applications **Android** avec une logique métier exprimée dans le langage **Rust**. Pour que cela se fasse relativement simplement, j'utilise personnellement la librairie **Qt** à l'aide de la technologie **QML** qui permet de fabriquer des applications **multi-plateformes** (aussi bien fonctionnant sur **PC** que sur **Smartphone**).

*Grâce à **QtCreator**, nous pouvons générer ce type d'applications. Normalement, la librairie **Qt** dans son ensemble est construite à partir du langage **C++**, sachant que **QML** est plutôt un langage descriptif lui-même transformé en classes **C++** équivalentes lors de la phase de compilation.*

*Si nous décidons de générer une application **Android** avec ces technologies, le tout est ensuite automatiquement transformé en langage **Java** avec un exécutable de type **APK**, puisque c'est l'ossature même de ce type d'application. **QtCreator** est capable de maîtriser toutes ces phases de développement alors que nous travaillons dans le langage par défaut prévu pour cela, le **C++**.*

***QtCreator** propose également des projets en langage **Python.** Puisque nous disposons de toutes ces possibilités, nous allons voir comment utiliser une librairie fabriquée en **Rust** à travers un script écrit en **Python**. Le choix du **Python** s'explique par le fait que la syntaxe est plus simple et intuitive à écrire, notamment pour la définition des propriétés.*

*Par contre, vu que c'est un langage script, le temps de démarrage et d'exécution est toujours beaucoup plus que l'équivalent en **C++** et surtout, le langage **Python** ne permet absolument pas de faire du développement **Android**. Nous prévoirons alors une communication plus tard avec le langage **C++** qui est le langage natif pour tout développement avec la librairie **Qt**.*

Création d'un librairie Rust avec le module PyO3

Dans toutes les études et les projets que nous avons fait jusqu'à présent, nous avons toujours réalisé des applications définitives, Nous n'avons pas encore créé de librairies qui pouvait être exploitées par d'autres programmes. Nous allons voir ici comment faire sachant que la librairie sera utilisée à posteriori, par un script **Python**.

*Lorsque nous fabriquons des programmes dans des langages différents, il faut qu'ils soient capables de communiquer entre eux (**interopérabilité**), ce qui n'est pas évident à priori. Nous avons besoin d'un transcripteur qui sert d'interface et qui permet de bien adapter les différents types utilisés dans les deux langages.*

*Pour cela, nous avons besoin du module **PyO3** qui réalise cette fonction de transcription grâce à des annotations spécifiques qui permettront ainsi de voir les fonctions et les structures écrites en **Rust** comme si c'était des fonctions et des classes en **Python**. Pour un projet **Rust** dont l'objectif est de créer une librairie, nous devons respecter certains critères.*

*Dans **« src »**, nous devons créer un fichier qui se nomme explicitement **« lib.rs »** et qui comporte tout le source relatif à la bibliothèque. Dans le fichier de description du projet **« Cargo.toml »**, vous indiquez bien qu'il s'agit de construite une librairie. Elle peut être statique **« staticlib »** ou dynamique **« cdylib »** (préférable pour le script **Python**). Enfin, au lieu de proposer un **« run »** classique, nous devons choisir un **« build »** pour la phase de compilation.*

Cargo.toml

```
[package]
name = "rust-pyo3"
version = "0.1.0"
edition = "2021"

[lib]
name = "rust"
crate-type = ["cdylib"]

[dependencies]
pyo3 = { version = "0.16.5", features = ["extension-module"] }
```

*Lorsque nous effectuons la construction avec **« cargo build --release »**, nous obtenons le fichier suivant **« librust.so »**. Il s'agit bien d'une librairie dont le nom est **rust** et qui est une librairie dynamique puisque l'extension est ***.so**. Le déroulement de cette construction respecte le fait que nous avons proposer une zone **[lib]** dans **Cargo.toml** avec les propriétés **name** et **crate-type**.*

Contenu de la librairie

Maintenant que nous savons comment fabriquer une librairie, je vous propose de nous intéresser à la mise en œuvre de fonctions et de structures qui seront exploitées par la suite par le script **Python**. Nous allons pour cela exploiter les annotations propres au module **PyO3** tout en respectant la syntaxe du langage **Rust**.

lib.rs

```
use pyo3::prelude::*;

#[pyfunction]
fn euro_franc(euro: f64) -> f64 {
    euro * 6.55957
}

#[pyfunction]
#[pyo3(name="franceuro")]
fn franc_euro(franc: f64) -> f64 {
    franc / 6.55957
}
```

```rust
#[pyclass]
struct Monnaie {
    #[pyo3(get)]
    taux: f64,
    #[pyo3(set)]
    euro: f64,
    #[pyo3(set)]
    franc: f64
}

#[pymethods]
impl Monnaie {
    #[new]
    fn new() -> Self {
        Monnaie { taux: 6.55957, euro: 0., franc: 0. }
    }
    #[getter]
    #[pyo3(name="franc")]
    fn calcul_franc(&mut self) -> f64 {
        self.franc = self.euro * self.taux;
        self.franc
    }
    #[getter]
    #[pyo3(name="euro")]
    fn calcul_euro(&mut self) -> f64 {
        self.euro = self.franc / self.taux;
        self.euro
    }
    #[pyo3(name="eurofranc")]
    fn euro_franc(&self, euro: f64) -> f64 {
        euro * self.taux
    }
    #[pyo3(name="franceuro")]
    fn franc_euro(&self, franc: f64) -> f64 {
        franc / self.taux
    }
}

#[pyclass(name = "Élève")]
struct Eleve {
    #[pyo3(get)]
    nom: String,
    #[pyo3(get, name="prénom")]
    prenom: String,
    #[pyo3(get, set)]
    notes: Vec<f64>
}

#[pymethods]
impl Eleve {
    #[new]
    fn new(n: String, p: String) -> Self {
        let nom = n.to_uppercase();
        let prenom = p[0..1].to_uppercase() + &p[1..].to_lowercase();
        Eleve {
            nom, prenom,
            notes: vec![]
        }
    }
    #[getter]
    fn moyenne(&self) -> f64 {
        let mut somme = 0.;
        for note in &self.notes {
            somme += note
        }
        somme/self.notes.len() as f64
    }
    #[getter]
    fn description(&self) -> String {
        format!("{} {}, notes : {:?} - moyenne = {}", self.prenom, self.nom, self.notes, self.moyenne())
    }
}

#[pymodule]
fn librust(_py: Python, m: &PyModule) -> PyResult<()> {
    m.add_function(wrap_pyfunction!(euro_franc, m)?)?;
    m.add_function(wrap_pyfunction!(franc_euro, m)?)?;
    m.add_class::<Monnaie>()?;
```

```
    m.add_class::<Eleve>()?;
    Ok(())
}
```

*La première remarque que nous pouvons faire lorsque nous consultons ce code source, c'est que nous retrouvons la syntaxe classique dans la mise en œuvre des fonctions et des structures au sein du langage **Rust**. Toutefois, chacune d'entre-elles est précédée d'un annotation particulière qui permet au script **Python** de s'y retrouver.*

*Ces annotations servent à déclarer les fonctions et les classes qui seront visibles dans le script **Python**. Même si ce n'est pas le cas ici, nous pouvons créer des fonctions et des classes qui sont utiles uniquement pour la librairie sans qu'elles soient accessibles pour le script **Python**. Il suffit de ne pas les déclarer par ces annotations (**fonctions et classes privées**).*

*Pour déclarer une fonction, vous devez proposer l''annotation suivante **#[pyfunction]**, tous les types utilisés par cette fonction seront automatiquement transformés pour être compris par **Python**. Il est possible d'avoir un nom de fonction différent pour chacun des langages, vous rajoutez alors l'annotation **#[pyo3(name= « nompython »)]**.*

*Pour déclarer une classe, vous rajoutez l'annotation **#[pyclass]** au dessous de la composition de votre structure. Les attributs sont ou pas accessibles. Pour qu'ils le soient, vous devez systématiquement proposer une annotation, de lecture avec **#[pyo3(get)]**, d'écriture avec **#[pyo3(set)]** et de lecture/écriture avec **#[pyo3(get, set)]**.*

*Comme pour les fonctions, le nom de la classe **Python** peut être différent du nom de la structure **Rust**, grâce à la spécificité **#[pyclass(name = « nompython »)]**.*

*Lors de la définition des méthodes, afin qu'elles soient toutes utilisables par **Python**, il suffit de faire une déclaration unique au dessus du bloc d'implémentation avec la notation suivante **#[pymethods]**, Ce n'est pas obligatoire, mais nous pouvons rajouter des annotations supplémentaires sur certaine des méthodes de la structure.*

*Nous avons la possibilité de désigner une méthode qui sera le constructeur de la classe grâce à l'annotation **#[new]**. Là aussi, nous pouvons également proposer un changement de nom entre les deux langages.*

***Python** possède une particularité intéressante qui n'existe pas dans **Rust** et qui permet de désigner certaines méthodes de telles sortes qu'elles soient vu comme si c'étaient des attributs grâce à la notion de **getter** et de **setter**. Il existe justement des annotations pour exprimer ce type de concept qui sera extrêmement utile lors de l'exploitation des scripts **QML**.*

*Attention, pour que toutes ces déclarations soient effectives, elles doivent impérativement être intégrées dans un module. Vous devez donc créer un module, sous forme d'une fonction, qui porte exactement le nom que la libraire, ici donc **librust**, puisque c'est le nom du fichier généré.*

*Le module est ainsi identifié par l'annotation **#[pymodule]** qui doit être unique. La fonction qui représente ce module doit posséder deux paramètres respectivement de type **Python** et **PyModule**. Dans ce module, vous intégrez les fonctions et les classes que vous devez diffuser grâce à deux méthodes spécifiques **add_function()** et **add_class()**.*

EXPLOITATION DE LA LIBRAIRIE, SCRIPT PYTHON

Notre librairie étant créée, je vous propose d'écrire un script en **Python** qui va pleinement exploiter les fonctions et les classes que nous venons de décrire. Votre script doit être au même niveau que la librairie. L'idéal peut-être est d'utiliser un lien symbolique sur la librairie afin d'avoir des projets pour ces deux langages dans des dossiers différents.

test.py

```
from librust import *

franc = euro_franc(15.24)
euro = franceuro(100.)
print("Franc =", franc, ", Euro =", euro)

monnaie = Monnaie()
monnaie.euro = 15.24
print(monnaie.franc, monnaie.eurofranc(15.25))

élève = Élève("boréale", "aurore")
élève.notes = [12.5, 10., 15.]
print("Moyenne =", élève.moyenne, "de", élève.prénom, élève.nom, "- notes =", élève.notes)
print(élève.description)
```

résultat

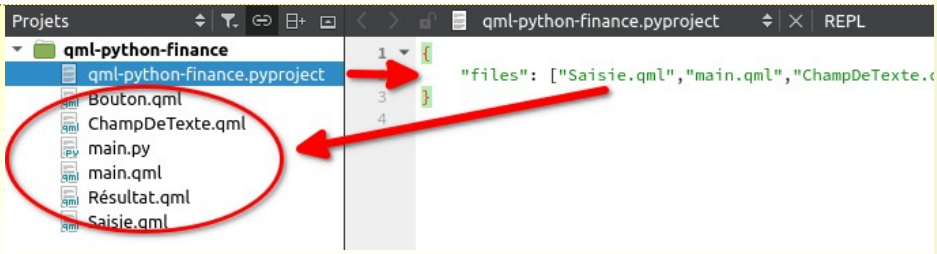
```
Franc = 99.9678468 , Euro = 15.244901723741037
99.9678468 100.03344249999999
Moyenne = 12.5 de Aurore BORÉALE - notes = [12.5, 10.0, 15.0]
Aurore BORÉALE, notes : [12.5, 10.0, 15.0] - moyenne = 12.5

Process finished with exit code 0
```

*La première ligne du script est importante puisqu'elle intègre la bibliothèque précédente grâce à la fois à la bonne désignation du fichier et aussi par la bonne désignation du module décrit dans la bibliothèque. Ensuite, nous retrouvons bien nos fonctions et nos classes comme si elles étaient écrites en **Python**. Elles sont très faciles et intuitives à utiliser. Nous profitons du fait qu'en **Python**, nous pouvons écrire en utilisant l'accentuation française, ce qui pour moi est très agréable.*

CRÉATION D'UN PROJET PYTHON AVEC UNE DESCRIPTION QML À L'AIDE DE QTCREATOR

Nous nous intéressons maintenant à l'élaboration d'un projet multi-plateforme qui va nous permettre de connaître les mensualités sur une étude de financement. Nous utilisons la librairie **Qt**, avec un descriptif **QML** pour **l'IHM,** et le langage **Python** pour la partie contrôleur qui permet d'effectuer les traitements souhaités pour les différents calculs de financement.

*Pour l'instant, dans cette étude, le langage **Rust** n'est pas utilisé. L'objectif ici est de bien maîtriser les concepts nécessaires pour l'élaboration d'un projet multi-plateforme avec la librairie **Qt** avec une écriture la plus simple et la plus intuitive possible. Je ne rentrerais pas dans les détails techniques d'un document **QML** sachant que cette partie est largement traitée sur les cours du langage **C++**.*



*L'objectif d'un développement avec la librairie **Qt** est de séparer l'apparence de l'application (**la vue**) de son traitement de fond (**le contrôleur**) qui ici est réalisé en langage **Python**.*

*Lorsque vous traitez l'aspect visuel sous forme descriptive à l'aide de documents **QML**, vous n'avez pas à vous préoccuper du traitement de fond, et vice-versa. Si l'apparence ne vous plaît plus, il est très facile de la changer en modifiant les différentes descriptions initiales sans se tracasser du traitement de fond associé au **contrôleur**, lui ne bouge pas. C'est le grand intérêt de ce type d'approche, appelé modèle **MVC**.*

*Grâce à **QML**, vous pouvez facilement concevoir de nouveaux composants avec des zones de saisie complètement personnalisés. Dans cette première partie de codage, je propose de fabriquer un bouton personnalisé, un champ de texte et les deux rubriques associées qui représenterons respectivement la saisie et le résultat des calculs, tout ceci grâce à des documents **QML** spécifiques.*

*Ces différents composants personnalisés pourront ensuite être utilisés par le document **QML** principal. Lorsque nous fabriquons de nouveaux composants, nous réalisons en réalité un **héritage** sur d'autres composants déjà préfabriqués, nous rajoutons juste les spécificités souhaitées.*

Bouton.qml

```
import QtQuick
import QtQuick.Controls


Button {
  font {
    bold: true
    italic: true
    pointSize: 16
  }
  anchors {
    left: parent.left
    leftMargin: 20
    right: parent.right
    rightMargin: 20
  }
  background: Rectangle {
    id: fond
    color: "orangered"
    radius: 7
    opacity: 0.4
  }
  onPressed: fond.color = "darkred"
  onReleased: fond.color = "orangered"
}
```

*Voici donc un exemple de la création d'un nouveau composant. Il s'agit d'un composant nommé **Bouton** (puisque le nom du fichier s'appelle « **Bouton.qml** ») qui hérite d'un composant prédéfini **Button** donné avec la librairie **Qt**.*

ChampDeTexte.qml

```
import QtQuick
import QtQuick.Controls
```

```
TextField {
  property string intitulé
  property string symbole: "€"
  property string couleur: "darkred"
  property double valeur: 0.0
  property int décimales: -1
  property alias fond: fond

  font.pointSize: 16
  font.bold: true
  rightPadding: 70
  horizontalAlignment: TextInput.AlignRight
  text: "0.0"

  anchors {
    left: parent.left
    leftMargin: 20
    right: parent.right
    rightMargin: 20
  }

  onDécimalesChanged: text = Number(valeur).toLocaleString(Qt.locale("fr_FR"), 'f', décimales)
  onValeurChanged: text = Number(valeur).toLocaleString(Qt.locale("fr_FR"), 'f', décimales)

  background: Rectangle {
    id: fond
    color: "#BBFFFFFF"
    radius: 7
  }
  Text {
    id: préfixe
    text: intitulé
    color: couleur
    font.pointSize: 14
    font.italic: true
    anchors {
      verticalCenter: parent.verticalCenter
      left: parent.left
      leftMargin: 10
    }
  }
  Text {
    id: suffixe
    text: symbole
    font.pointSize: 16
    color: couleur
    width: 50
    anchors {
      verticalCenter: parent.verticalCenter
      right: parent.right
      rightMargin: 10
    }
  }
}
```

*Ce composant **ChampDeTexte** correspond à une zone de saisie puisqu'il hérite de la classe **TextField**. Au delà de la zone de saisie, ce composant personnalisé propose deux textes supplémentaires qui correspond à un préfixe qui identifie cette zone et un suffixe pour exprimer le type de valeur saisie. Le réglage de ce préfixe et de ce suffixe se fait au travers de propriétés créées de toute pièce afin de personnaliser l'intitulé et le symbole du type de saisie.*

Saisie.qml

```
import QtQuick
import QtQml
import QtQuick.Controls

ChampDeTexte {
  inputMethodHints: Qt.ImhDigitsOnly
  selectedTextColor: "darkred"
  selectionColor: "#20FF0000"
  fond.border.color: "darkred"
  onFocusChanged: {
    fond.color = focus ? "white" : "#BBFFFFFF"
    if (focus) selectAll()
    else {
      valeur = Number.fromLocaleString(Qt.locale("fr_FR"), text)
      deselect()
    }
  }
}
```

*Le composant précédent **ChampDeTexte** sert à créer un nouveau composant propre à la saisie, nommé **Saisie**.*

Résultat.qml

```
import QtQuick
import QtQuick.Controls

ChampDeTexte {
  couleur: "darkgreen"
  readOnly: true
  fond.color: "#BBCCFFCC"
  fond.border.color: "darkgreen"
}
```

*De même **ChampDeTexte** sert à créer un nouveau composant en lecture seule propre à l'affichage des valeurs calculées, nommé **Résultat**. Avant de visualiser le code de la page principale qui utilise ces composants personnalisés, je vous propose de construire le **contrôleur** qui lui aussi sera finalement un nouveau composant utilisable par la page principale. Ce contrôleur est exprimé en langage **Python**. C'est ce composant qui réalise tous les calculs nécessaires à l'application.*

main.py

```
# This Python file uses the following encoding: utf-8
import os
from pathlib import Path
import sys
from PySide6.QtGui import QGuiApplication
from PySide6.QtQml import QQmlApplicationEngine, QmlElement
from PySide6.QtCore import QObject, Slot

QML_IMPORT_NAME = "manu.python.rust"
QML_IMPORT_MAJOR_VERSION = 1
QML_IMPORT_MINOR_VERSION = 0

@QmlElement
class Financement(QObject):
    @Slot(float, int, float)
    def calcul(self, C, années, taux):
        self.C = C
        n = 12
        r = taux / 100
        self.N = N = années * n
        self.m = C * r / n /(1 - pow(1+r/n, -N))

    @Slot(result=int)
    def nombreMensualités(self):
        return self.N

    @Slot(result=float)
    def mensualité(self):
        return self.m

    @Slot(result=float)
    def coûtTotal(self):
        return self.m * self.N

    @Slot(result=float)
    def intérêts(self):
        return self.m * self.N - self.C

if __name__ == "__main__":
    app = QGuiApplication(sys.argv)
    engine = QQmlApplicationEngine()
    engine.load(os.fspath(Path(__file__).resolve().parent / "main.qml"))
    if not engine.rootObjects():
        sys.exit(-1)
    sys.exit(app.exec())
```

*Comme pour le langage **C++**, nous devons créer une classe qui hérite de la classe **QObject** afin de pouvoir bénéficier de la mise en œuvre de nouveaux **slots** et de nouvelles **propriétés** (éventuellement aussi de **signaux**) qui correspond à la philosophie de la librairie **Qt**. Ici, il s'agit du composant nommé **Financement** qui sera directement accessible à la page principale de la **vue**.*

*Ce composant est bien le **contrôleur** principal de l'application. Pour créer un contrôleur à même de maîtriser la communication avec d'autres composants de la librairie **Qt**, nous utilisons une librairie spécifique de **Python** appelée **PySide6**.*

*Pour que le **contrôleur** puisse être importé sur la page principale, nous devons proposer un ensemble de déclarations commençant par **QML_IMPORT_**. Ensuite, la génération du contrôleur se fait à l'aide d'annotations spécifiques, comme **@QmlElement** et **@Slot**.*

*Je rappelle que les **slots** permettent d'appeler automatiquement les méthodes associées lorsqu'un signal particulier est sollicité. Nous pouvons également appeler ces méthodes explicitement depuis la **vue**. Ceci ne peut se faire que si la méthode est bien définie comme un **slot**.*

*Lors de la définition d'un **slot**, l'annotation doit être placée juste au-dessus de la méthode concernée. Dans l'annotation, vous précisez le type des arguments souhaités ainsi que le type de retour, si la méthode en propose un.*

*Enfin, précisons que « **main.py** » est le programme principal de l'application. Son premier objectif est d'activer la vue principale nommée « **main.qml** ». Une fois que la vue principale est active, elle sollicite l'ensemble des composants personnalisés créés précédemment, **contrôleur** compris. C'est ce que nous vérifions maintenant.*

main.qml

```
import QtQuick
import QtQuick.Window
import QtQuick.Layouts
import QtQuick.Controls

import manu.python.rust 1.0

Window {
  visible: true
  width: 380
  height: 570
  color: "beige"
  title: qsTr("Calcul de mensualités")

  Financement {
    id : finance
  }

  Column {
    anchors.fill: parent
    padding: 20
    spacing: 20

    Saisie {
      id: capital
      intitulé: "Capital"
      décimales: 0
    }
    Saisie {
      id: années
      symbole: "ans"
      intitulé: "Nombre d'annuités"
      décimales: 0
    }
    Saisie {
      id: taux
      symbole: "%"
      intitulé: "Taux d'intérêt"
      décimales: 1
    }
    Bouton {
      text: "Calcul des mensualités"
      onClicked: {
        finance.calcul(capital.valeur, années.valeur, taux.valeur)
        mensualité.valeur = finance.mensualité()
        coûtTotal.valeur = finance.coûtTotal()
        intérêts.valeur = finance.intérêts()
        mois.valeur = finance.nombreMensualités()
      }
    }
    Résultat {
      id: mois
      symbole: "mois"
      intitulé: "Nombre de mensualités"
      décimales: 0
    }
    Résultat {
      id: mensualité
      intitulé: "Mensualité"
      décimales: 2
    }
    Résultat {
      id: coûtTotal
      intitulé: "Coût total"
      décimales: 2
    }
```

```
    Résultat {
      id: intérêts
      intitulé: "Intérêts"
      décimales: 2
    }
  }
}
```

*Afin de bien exploiter certaines **propriétés** et certains **slots** des composants (**classes**) de votre **IHM**, vous devez spécifier un nom **d'objet** pour chacun d'entre eux, identifié par la propriété **id**.*

MAÎTRISER LES PROPRIÉTÉS

En reprenant le projet précédent, je vous propose de voir comment communiquer à l'aide de **propriétés** entre le **contrôleur** et la **vue**. Il faut savoir que lorsque nous travaillons avec la librairie **Qt** il existe trois moyens d'échange entre les différents composants : par des **slots**, par des **signaux** et au moyen des **propriétés**.

*Je rappelle que les propriétés complètes sont composées de trois éléments : un **attribut** de l'objet concerné, une **méthode accesseur** qui permet de lire le contenu de l'attribut équivalent et une **méthode mutateur** qui permet de modifier cet attribut. Ces méthodes sont souvent appelées **getter** et **setter**.*

*Lorsque nous définissons une nouvelle **propriété**, nous ne sommes pas obligé de décrire ces trois éléments, par contre la communication avec l'extérieur se fait toujours au moyen d'une de ces méthodes. C'est le principe de la programmation objet, les **attributs** sont normalement toujours **privés**.*

*Dans le projet précédent, je propose trois **propriétés** d'écriture pour enregistrer les valeurs saisies par l'utilisateur et trois **propriétés** de lecture pour récupérer les résultats du calcul. Enfin, le **slot** ne possède plus aucun paramètre puisque les valeurs saisies sont déjà récupérées par les **propriétés** précitées.*

main.py

```
import os
from pathlib import Path
import sys
from PySide6.QtGui import QGuiApplication
from PySide6.QtQml import QQmlApplicationEngine, QmlElement
from PySide6.QtCore import QObject, Slot, Property

QML_IMPORT_NAME = "manu.python.rust"
QML_IMPORT_MAJOR_VERSION = 1
QML_IMPORT_MINOR_VERSION = 0

@QmlElement
class Financement(QObject):
    def enregistrerCapital(self, C):
        self.C = C
    def enregistrerAnnées(self, années):
        self.années = années
    def enregistrerTaux(self, taux):
        self.r = taux / 100;

    @Slot()
    def calcul(self):
        n = 12
        self.N = N = self.années * n
        r = self.r
        self.m = self.C * r / n /(1 - pow(1+r/n, -N))

    def lireNombreMensualités(self):
        return self.N
    def lireMensualité(self):
        return self.m
    def lireCoûtTotal(self):
        return self.m * self.N
    def lireIntérêts(self):
        return self.m * self.N - self.C

    capital = Property(float, fset=enregistrerCapital)
    ans = Property(int, fset=enregistrerAnnées)
    taux = Property(float, fset=enregistrerTaux)
    nombreMensualités = Property(int, fget=lireNombreMensualités)
    mensualité = Property(float, lireMensualité)
    coûtTotal = Property(float, lireCoûtTotal)
    intérêts = Property(float, lireIntérêts)

if __name__ == "__main__":
    app = QGuiApplication(sys.argv)
    engine = QQmlApplicationEngine()
    engine.load(os.fspath(Path(__file__).resolve().parent / "main.qml"))
    if not engine.rootObjects():
```

```
        sys.exit(-1)
    sys.exit(app.exec())
```

*La définition des propriétés se fait au moyen de la directive **Property()** où vous devez spécifier impérativement le type de la propriété (comme pour les **slots** lorsque nous avons des arguments ou des valeurs de retour) suivi dans l'ordre de la méthode **accesseur** et de la méthode **mutateur**.*

*Si vous n'utilisez pas toutes les capacités de la propriété, vous pouvez qualifier l'argument qui vous intéresse, **« fget »** pour la méthode **accesseur**, **« fset »** pour la méthode **mutateur**.*

main.qml

```
import QtQuick
import QtQuick.Window
import QtQuick.Layouts
import QtQuick.Controls
import manu.python.rust 1.0

Window {
  visible: true
  width: 380
  height: 570
  color: "beige"
  title: qsTr("Calcul de mensualités")

  Financement {
    id : finance
  }

  Column {
    anchors.fill: parent
    padding: 20
    spacing: 20
    Saisie {
      id: capital
      intitulé: "Capital"
      décimales: 0
    }
    Saisie {
      id: années
      symbole: "ans"
      intitulé: "Nombre d'annuités"
      décimales: 0
    }
    Saisie {
      id: taux
      symbole: "%"
      intitulé: "Taux d'intérêt"
      décimales: 1
    }
    Bouton {
      text: "Calcul des mensualités"
      onClicked: {
        finance.capital = capital.valeur
        finance.ans = années.valeur
        finance.taux = taux.valeur
        finance.calcul()
        mensualité.valeur = finance.mensualité
        coûtTotal.valeur = finance.coûtTotal
        intérêts.valeur = finance.intérêts
        mois.valeur = finance.nombreMensualités
      }
    }
    Résultat {
      id: mois
      symbole: "mois"
      intitulé: "Nombre de mensualités"
      décimales: 0
    }
    Résultat {
      id: mensualité
      intitulé: "Mensualité"
      décimales: 2
    }
    Résultat {
      id: coûtTotal
      intitulé: "Coût total"
      décimales: 2
    }
    Résultat {
```

```
        id: intérêts
        intitulé: "Intérêts"
        décimales: 2
      }
    }
}
```

*Côté **vue**, lorsque nous utilisons une propriété, c'est comme si nous travaillions directement avec les attributs de l'objet. D'ailleurs, dans **QML**, la description de l'interface graphique se fait justement en définissant les propriétés des différents composants placés sur la **vue**. Pour les **slots**, il s'agit d'appel de méthodes classiques (méthodes **publiques** pour la **vue**).*

Effectuer la logique métier dans un programme Rust

Nous allons maintenant fusionner nos deux études distinctes pour en former plus qu'une. Le calcul du financement se fait dès lors dans un module **Rust** qui sera intégré au projet précédent. De façon classique, le **contrôleur** sert alors juste d'intermédiaire entre la **vue** et le traitement en coulisse.

*Le traitement du calcul se fait par l'intermédiaire d'une librairie qui sera intégrée au projet précédent. Nous utilisons pour cela les compétences du module **PyO3**. Nous créons une **structure Finance** qui possède une seule méthode, le **constructeur**. Cette structure comporte juste les attributs nécessaires à la diffusion des résultats. Le constructeur prend les arguments dont il a besoin pour effectuer les calculs, c'est-à-dire le capital, le nombre d'années de remboursement et bien sûr le taux du crédit.*

lib.rs

```
use pyo3::prelude::*;

#[pyclass]
struct Finance {
   #[pyo3(get, name="nombreMensualités")]
   nombre_mensualites: u32,
   #[pyo3(get, name="mensualité")]
   mensualite: f64,
   #[pyo3(get, name="coûtTotal")]
   cout_total: f64,
   #[pyo3(get, name="intérêts")]
   interets: f64
}

#[pymethods]
impl Finance {
   #[new]
   fn new(C: f64, annees: u32, taux: f64) -> Self {
      let n = 12.;
      let r = taux / 100.;
      let N = annees as f64 * n;
      let m = C * r / n / (1. - (1.+r/n).powf(-N));

      Finance {
         nombre_mensualites: N as u32,
         mensualite: m,
         cout_total: m * N,
         interets: m * N - C
      }
   }
}

#[pymodule]
fn librust(_py: Python, m: &PyModule) -> PyResult<()> {
   m.add_class::<Finance>()?;
   Ok(())
}
```

*Cette librairie ne possède que cette structure que nous devons déclarer en trois phases afin qu'elle soit parfaitement opérationnelle dans le **contrôleur** écrit en langage **Python**. D'abords, la définition de la classe, la méthode associée et la déclaration du module qui permet l'accès direct à cette classe.*

main.py

```
import os, sys
from pathlib import Path
from librust import Finance

from PySide6.QtGui import QGuiApplication
from PySide6.QtQml import QQmlApplicationEngine, QmlElement
from PySide6.QtCore import QObject, Slot, Property

QML_IMPORT_NAME = "manu.python.rust"
QML_IMPORT_MAJOR_VERSION = 1
QML_IMPORT_MINOR_VERSION = 0
```

```python
@QmlElement
class Financement(QObject):
    @Slot(float, int, float)
    def calcul(self, C, années, taux):
        self.finance = Finance(C, années, taux)

    nombreMensualités = Property(int, lambda self: self.finance.nombreMensualités)
    mensualité = Property(float, lambda self: self.finance.mensualité)
    coûtTotal = Property(float, lambda self: self.finance.coûtTotal)
    intérêts = Property(float, lambda self: self.finance.intérêts)

if __name__ == "__main__":
    app = QGuiApplication(sys.argv)
    engine = QQmlApplicationEngine()
    engine.load(os.fspath(Path(__file__).resolve().parent / "main.qml"))
    if not engine.rootObjects():
        sys.exit(-1)
    sys.exit(app.exec())
```

*La classe **Financement** est maintenant beaucoup plus réduite, puisque le traitement principal est déporté. Pour la définition des propriétés, j'utilise des **expressions lambda** qui sont sont tout-à-fait adaptés à ce genre de situation, ce qui permet d'avoir une plus grande concision dans le code. Voici ci-dessous le code réduit qui permet de visualiser les changements opérés.*

main.qml

```
...
Window {
  visible: true
  width: 380
  height: 570
  color: "beige"
  title: qsTr("Calcul de mensualités")

  Financement {
    id : finance
  }

  Column {
    anchors.fill: parent
    padding: 20
    spacing: 20

...
    Bouton {
      text: "Calcul des mensualités"
      onClicked: {
        finance.calcul(capital.valeur, années.valeur, taux.valeur)
        mensualité.valeur = finance.mensualité
        coûtTotal.valeur = finance.coûtTotal
        intérêts.valeur = finance.intérêts
        mois.valeur = finance.nombreMensualités
      }
…
```

Création d'une librairie Rust avec le module CXX

Tout ce que nous venons de construire fonctionne très bien pour une application sur poste de travail classique et l'implémentation est relativement simple et concise. Par contre, ce type d'application ne peut pas être déployé sur une plate-forme Android.

*Nous allons reprendre progressivement la même démarche qu'avec le langage **Python**, en mettant en œuvre un développement qui permet de communiquer entre un module **Rust** et un module **C++**. Comme pour les projets précédents, nous devons passer par la construction d'une librairie qui sera ensuite exploité par le langage **C++**.*

*La différence cette fois-ci c'est que nous aurons également la génération du fichier en-tête qui sera nécessaire pour exploiter correctement la librairie. Le module à prendre en compte ici se nomme **cxx** avec la construction automatique par **cxx-build**.*

Cargo.toml

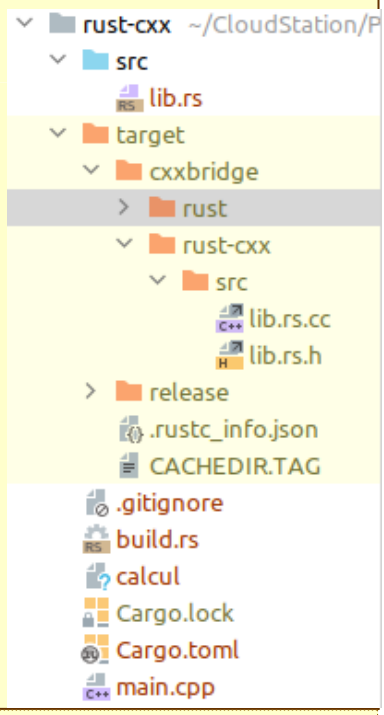
```toml
[package]
name = "rust-cxx"
version = "0.1.0"
edition = "2021"

[lib]
name = "rust"
crate-type = ["staticlib"]
```

```
[dependencies]
cxx = "1.0.73"

[build-dependencies]
cxx-build = "1.0.73"

# g++ -o calcul main.cpp -L target/release -l rust -pthread -l dl
```

*Lorsque nous effectuons la construction avec **« cargo build --release »**, nous obtenons le fichier suivant **« librust.a »**. Il s'agit bien d'une librairie dont le nom est **rust** qui est une librairie statique puisque l'extension est ***.a**. Le déroulement de cette construction respecte le fait que nous avons proposer une zone **[lib]** dans **Cargo.toml** avec les propriétés **name** et **crate-type**.*

lib.rs

```
#[cxx::bridge]
mod ffi {
  extern "Rust" {
    fn euro_franc(euro: f64) -> f64;
    fn franc_euro(franc: f64) -> f64;

    type Monnaie;
    fn creer_monnaie() -> Box<Monnaie>;
    fn calcul_franc(&mut self, euro: f64) -> f64;
    fn calcul_euro(&mut self, franc: f64) -> f64;
    fn description(&self) -> String;
  }
}

fn euro_franc(euro: f64) -> f64 { euro*6.55957 }
fn franc_euro(franc: f64) -> f64 { franc/6.55957 }

struct Monnaie {
  euro: f64,
  franc: f64
}

fn creer_monnaie() -> Box<Monnaie> {  Box::new(Monnaie{euro: 0., franc:0.}) }

impl Monnaie {
  fn calcul_franc(&mut self, euro: f64) -> f64 {
    self.euro = euro;
    self.franc = euro * 6.55957;
    self.franc
  }
  fn calcul_euro(&mut self, franc: f64) -> f64 {
    self.franc = franc;
    self.euro = franc / 6.55957;
    self.euro
  }
  fn description(&self) -> String {  format!("(euro={}, franc={})", self.euro,
self.franc)  }
}
```



*Comme précédemment, nous retrouvons la syntaxe classique pour mettre en œuvre des fonctions et des structures dans le langage **Rust**. Nous pouvons définir des fonctions et des structures avec leurs propres méthodes si nécessaire.*

*Pour que ces fonctions et ces structures puissent être accessibles depuis un programme écrit en langage **C++** dans un fichier annexe, nous devons mettre en œuvre un pont entre les deux modules au travers d'un module nommé **ffi** et d'une annotation adaptée **#[cxx::bridge]**.*

*Ce module **ffi** permet de déclarer les fonctions, les données (**structures**) avec les méthodes associées qui sont exprimées en langage **Rust**. Pour qu'il n'y ait aucune ambiguïté, nous devons déclarer chacun de ces éléments dans une zone externe nommée justement **Rust**.*

*Vous ne devez pas déclarer votre structure avec le mot clé **struct**, mais avec avec le mot clé **type**. Attention, vous ne pouvez utiliser les structures directement mais toujours au travers d'une indirection. La structure ou l'énumération devra être encapsulée dans une boîte (**Box**). La définition des fonctions et des structures avec leurs méthodes doivent par contre se faire à l'extérieur du module **ffi**. La phase de construction permet d'avoir une cohérence entre les deux langages. L'appel des fonctions et des méthodes se fait naturellement côté **C++** sachant que tout ce qui est écrit en **Rust** est adapté en conséquence.*

*Pour que la construction se fasse complètement, nous devons l'initier avec un fichier spécifique qui se nomme justement « **build.rs** » au même niveau que « **Cargo.toml** ».*

build.rs

```
fn main() {
   cxx_build::bridge("src/lib.rs").compile("rust_cpp");
}
```

*La construction complète génère alors la librairie statique ainsi que le fichier en-tête écrit en **C++**. Vous pouvez alors fabriquer votre programme principal en prenant en compte ces deux éléments. Voici un exemple d'utilisation avec un programme écrit en **C++** dans le même répertoire que le projet précédent :*

main.cpp

```
#include <iostream>
#include "target/cxxbridge/rust-cxx/src/lib.rs.h"
using namespace std;

int main() {
   double franc = euro_franc(15.24);
   double euro = franc_euro(100);
   cout << "franc=" << franc << ", euro=" << euro << endl;

   auto monnaie = creer_monnaie();
   cout << "franc=" << monnaie->calcul_franc(15.24) << endl;
   cout << monnaie->description().c_str() << endl;

   return 0;
}
```

*La variable **monnaie** représente le type **Box<Monnaie>** qui, grâce à **auto**, est alors automatiquement déréférencé (**pointeur**). L'utilisation des méthodes se fait en utilisant le séparateur « **->** ».*

*À partir du code source précédent, nous devons compiler le programme principal à l'aide de la commande habituelle **g++**. Dans la phase d'édition de lien, n'oubliez pas d'intégrer la librairie exprimée en **Rust** à l'aide de l'écriture suivante :*

Commande shell et exécution du programme

```
manu@PC Bureau:~/../rust-cxx$ g++ -o calcul main.cpp -L target/release -l rust -pthread -l dl
manu@PC Bureau:~/../rust-cxx$ ./calcul
franc=99.9678, euro=15.2449
franc=99.9678
(euro=15.24, franc=99.9678468)
```

Reprise du projet de financement avec la logique métier écrite en Rust

Nous allons exploiter nos connaissances avec l'interopérabilité entre le langage **Rust** et le langage **C++**. Nous verrons ainsi comment déployer notre librairie **Rust** dans une application **QML** avec l'ossature classique développée avec le langage natif **C++**. Par contre, il s'agira encore une fois d'un développement pour **PC** de Bureau.

*Dans un premier temps, nous nous intéressons uniquement au développement de la librairie **Rust** qui sera ensuite exploitée par l'application finale. Nous reprenons exactement la même pratique que lors du chapitre précédent.*

Cargo.toml

```
[package]
name = "rust-cxx-finance"
version = "0.1.0"
edition = "2021"

[lib]
name = "rust"
crate-type = ["staticlib"]

[dependencies]
cxx = "1.0.73"
[build-dependencies]
cxx-build = "1.0.73"
```

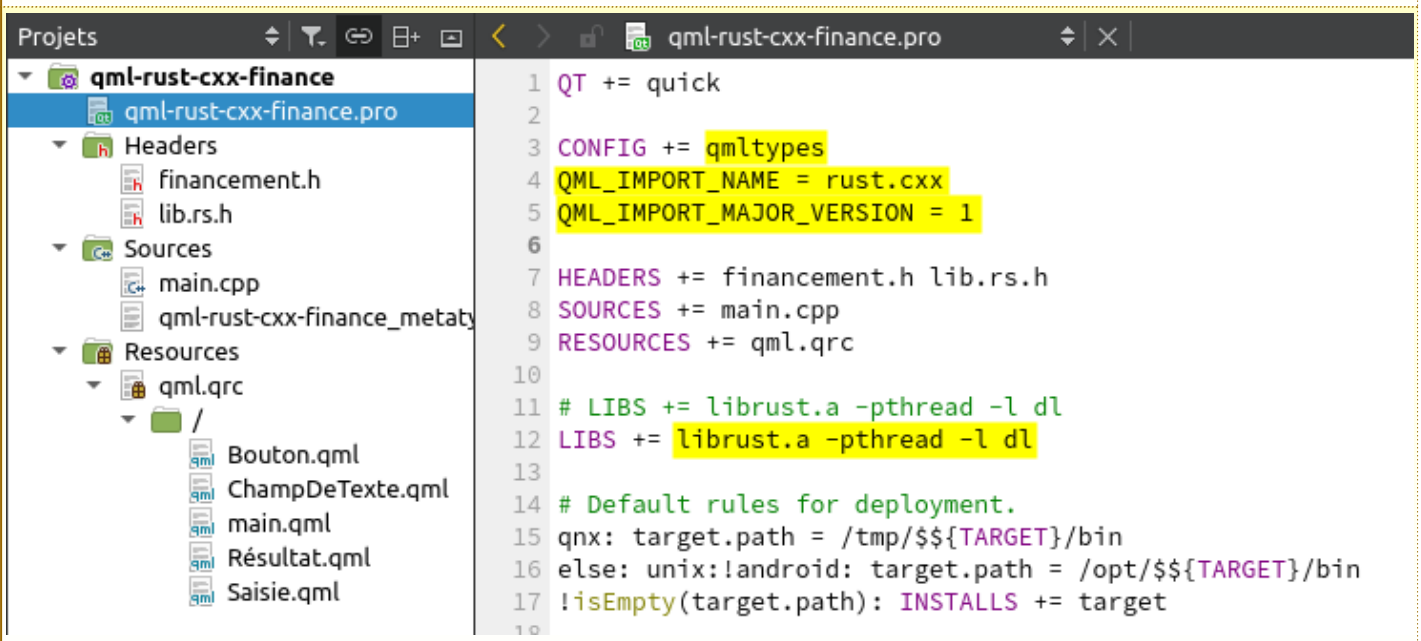
lib.rs

```
#[cxx::bridge]
mod ffi {
   extern "Rust" {
      type Finance;
      fn financement(capital: f64, annees: u32, taux: f64) -> Box<Finance>;
      fn mensualite(&self) -> f64;
      fn nombre_mensualites(&self) -> u32;
      fn cout_total(&self) -> f64;
      fn interets(&self) -> f64;
   }
}
struct Finance {
   attr_mensualite: f64,
   attr_nombre_mensualites: u32,
   attr_cout_total: f64,
   attr_interets: f64
}
```

```rust
fn financement(capital: f64, annees: u32, taux: f64) -> Box<Finance> {
    let mois = 12.;
    let taux = taux / 100.;
    let annuites = annees as f64 * mois;
    let m = capital * taux / mois / (1. - (1.+ taux / mois).powf(-annuites));
    Box::new(Finance {
        attr_nombre_mensualites: annuites as u32,
        attr_mensualite: m,
        attr_cout_total: m * annuites,
        attr_interets: m * annuites - capital
    })
}

impl Finance {
    fn mensualite(&self) -> f64 { self.attr_mensualite }
    fn nombre_mensualites(&self) -> u32 { self.attr_nombre_mensualites }
    fn cout_total(&self) -> f64 { self.attr_cout_total }
    fn interets(&self) -> f64 { self.attr_interets }
}
```

*La librairie est constituée d'une simple structure **Finance**, comme pour l'exemple en **Python**, avec les méthodes de lecture qui nous permettent de récupérer toutes les valeurs utiles après le calcul complet du financement désiré. Le calcul s'effectue avec la fonction **financement()** qui retourne une structure **Finance** avec les bons résultats associés aux paramètres de la fonction.*

*Après cette première phase, nous pouvons passer au développement **QML** avec **QtCreator**. Pour que cela soit plus pratique, je vous invite à placer votre librairie statique « **librust.a** » ainsi que le fichier en-tête « **lib.rs.h** » directement dans le nouveau projet de conception **Quick**.*



*Cette fois-ci, comme tout projet **Quick**, le contrôleur est ici exprimé en langage **C++** dans le fichier nommé « **financement.h** ». Il s'agit de décrire la classe **Financement** (qui hérite de **QObject**) où il sera possible de proposer les **propriétés** et les **slots** nécessaires à la communication des documents **QML**. Grâce, à la description donnée dans le fichier de projet, ce contrôleur pourra être considéré comme un autre objet **QML**, à la condition de le déclarer comme tel, grâce à la macro **QML_ELEMENT**.*

financement.h

```cpp
#ifndef FINANCEMENT_H
#define FINANCEMENT_H

#include <QObject>
#include <qqml.h>
#include "lib.rs.h"

class Financement : public QObject
{
  Q_OBJECT
  Q_PROPERTY(int nombreMensualites MEMBER nombreMensualites)
  Q_PROPERTY(double mensualite MEMBER mensualite)
  Q_PROPERTY(double coutTotal MEMBER coutTotal)
  Q_PROPERTY(double interets MEMBER interets)
  QML_ELEMENT
public:
  explicit Financement(QObject *parent = nullptr) : QObject() {}
```

```
public slots:
  void calcul(double capital, int annees, double taux) {
    auto finance = financement(capital, annees, taux);
    nombreMensualites = finance->nombre_mensualites();
    mensualite = finance->mensualite();
    coutTotal = finance->cout_total();
    interets = finance->interets();
  }
private:
  int nombreMensualites;
  double mensualite, coutTotal, interets;
};
#endif // FINANCEMENT_H
```

main.qml

```
...
import QtGraphicalEffects 1.15
import rust.cxx 1.0

Window {
  visible: true
  width: 380
  height: 570
  title: qsTr("Calcul de mensualités")

  LinearGradient {
    anchors.fill: parent
    gradient: Gradient {
      GradientStop { position: 0.0; color: "bisque" }
      GradientStop { position: 1.0; color: "lightgreen" }
    }
  }

  Financement {
    id : finance
  }

  Column {
    anchors.fill: parent
    padding: 20
    spacing: 20
...
    Bouton {
      text: "Calcul des mensualités"
      onClicked: {
        finance.calcul(capital.valeur, années.valeur, taux.valeur)
        mensualité.valeur = finance.mensualite
        coûtTotal.valeur = finance.coutTotal
        intérêts.valeur = finance.interets
        mois.valeur = finance.nombreMensualites
      }
    }
...
}
```

Développement pour Android

Tous les projets précédents ont permis de développer des applications avec la technologie **QML**, qui fonctionnent parfaitement pour des **PC** de type **Bureau**, mais aucun n'est compatible pour un développement **Android**. Même avec le dernier projet, si nous tentons de compiler la librairie **Rust** pour une cible de type **Android**, une erreur d'édition de lien apparaît systématiquement.

> *La problématique pour un développement **Rust** pour une plate-forme **Android**, c'est que l'ossature doit être à l'image du langage **C** et non du **C++** en passant par la **NDK**. La librairie qui correspond le plus à ce type d'approche en permettant **l'interopérabilité** entre **Rust** et **C++** s'appelle **cbindgen**. Comme pour les autres projets, nous commençons par générer la librairie de financement avec les compétences de **cbindgen**.,*

Cargo.toml

```
[package]
name = "rust-cxx-finance"
version = "0.1.0"
edition = "2021"

[lib]
name = "rust_android"
crate-type = ["staticlib"]

[build-dependencies]
cbindgen = "0.24.3"
```

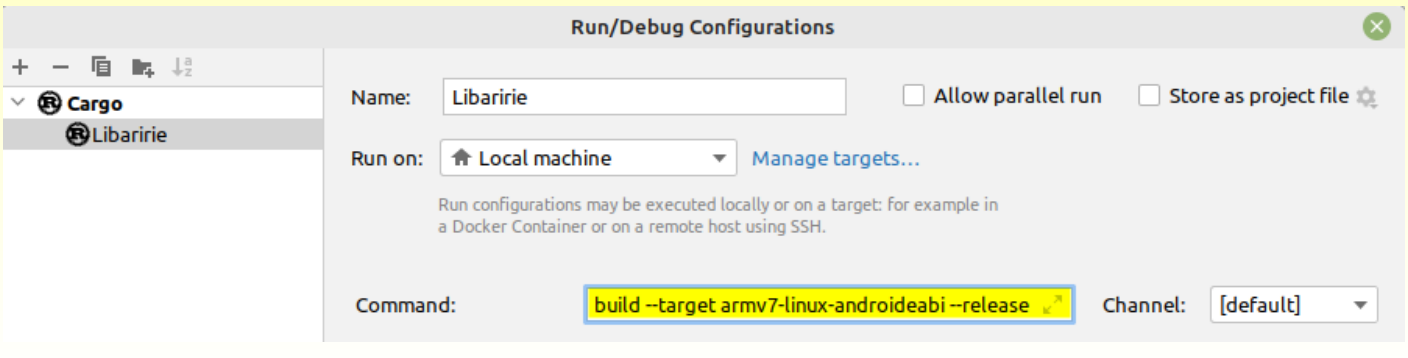
lib.rs

```rust
#[no_mangle]
pub extern fn bienvenue() {
    println!("bienvenue !");
}

#[no_mangle]
pub extern fn addition(left: usize, right: usize) -> usize {
    left + right
}

#[repr(C)]
pub struct Finance {
    mensualite: f64,
    nombre_mensualites: u32,
    cout_total: f64,
    interets: f64
}

#[no_mangle]
pub extern fn financement(capital: f64, annees: u32, taux: f64) -> Finance {
    let mois = 12.;
    let taux = taux / 100.;
    let annuites = annees as f64 * mois;
    let m = capital * taux / mois / (1. - (1.+ taux / mois).powf(-annuites));
    Finance {
        nombre_mensualites: annuites as u32,
        mensualite: m,
        cout_total: m * annuites,
        interets: m * annuites - capital
    }
}

impl Finance {
    #[no_mangle]
    pub extern fn lire_interets(&self) -> f64 {
        self.interets
    }
}
```

*Pour que tous les éléments constitutifs soient accessibles pour le code **C++**, il est nécessaire qu'ils soient publics. Les fonctions doivent être nommées externes avec systématiquement le préfixe **#[no_mangle]**. Il est possible d'avoir des structures et des énumérations, il suffit juste de les déclarer avec le préfixe **#[repr(C)]**.*

*Nous pouvons également rajouter des méthodes aux structures, mais elles seront considérées comme de simples fonctions avec en premier paramètre une référence constante à la structure. Comme pour les autres projets nous devons permettre la construction de la librairie avec le fichier en-tête adéquat au travers du fichier « **build.rs** ».*

*Attention, avant de lancer la phase de compilation et d'édition de lien il faut bien choisir sa cible dans votre outil de développement **Rust**, nommé **armv7-linux-androideabi** qui doit être bien entendu installé au préalable grâce à la commande :*
§ rustup target add armv7-linux-androideabi



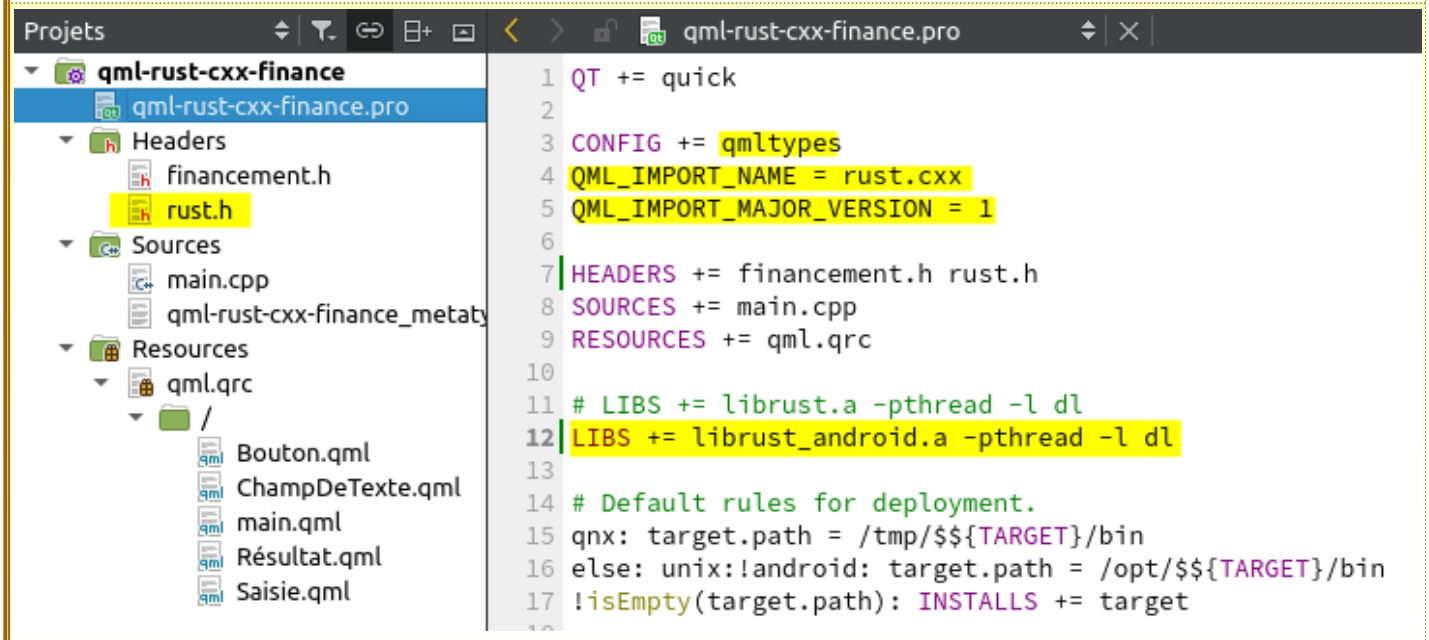
Après la phase de compilation, voici le fichier en-tête dont nous disposons :

rust.h

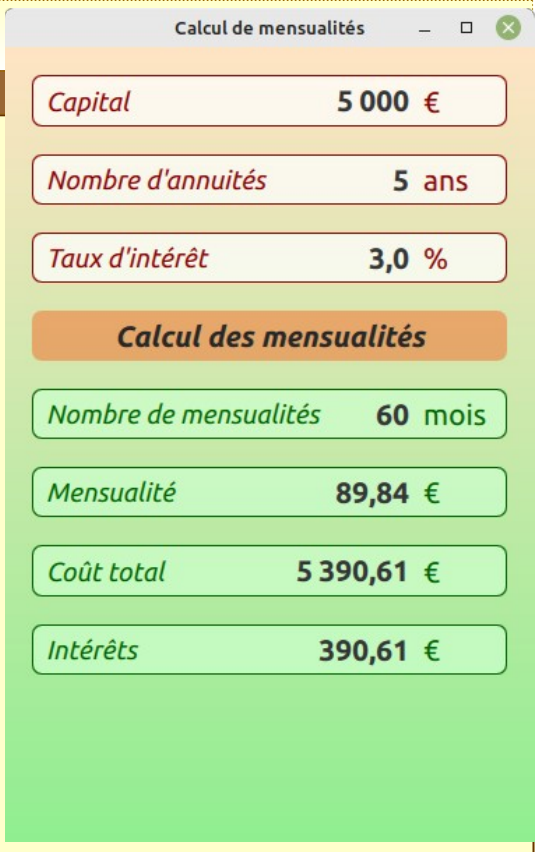
```c
#include <cstdarg>
#include <cstdint>
#include <cstdlib>
#include <ostream>
#include <new>
```

```rust
struct Finance {
  double mensualite;
  uint32_t nombre_mensualites;
  double cout_total;
  double interets;
};

extern "C" {
  void bienvenue();
  uintptr_t addition(uintptr_t left, uintptr_t right);
  Finance financement(double capital, uint32_t annees, double taux);
  double lire_interets(const Finance *self);
} // extern "C"
```

*Comme lors du projet précédent, placer ce fichier en-tête avec la librairie constituée dans le répertoire du projet développé avec **QtCreator** et les documents **QML** déjà conçus. Voici, la nouvelle alternative qui se déploie bien sur un smartphone **Android***



*Nous reprenons le projet précédent où il suffit seulement de modifier le fichier nommé « **financement.h** » en prenant en compte la nouvelle bibliothèque :*



financement.h

```rust
#ifndef FINANCEMENT_H
#define FINANCEMENT_H
#include <QObject>
#include <qqml.h>
#include "rust.h"

class Financement : public QObject
{
  Q_OBJECT
  Q_PROPERTY(int nombreMensualites MEMBER nombreMensualites)
  Q_PROPERTY(double mensualite MEMBER mensualite)
  Q_PROPERTY(double coutTotal MEMBER coutTotal)
  Q_PROPERTY(double interets MEMBER interets)
  QML_ELEMENT
public:
  explicit Financement(QObject *parent = nullptr) : QObject() {}
public slots:
  void calcul(double capital, int annees, double taux) {
    Finance finance = financement(capital, annees, taux);
    nombreMensualites = finance.nombre_mensualites;
    mensualite = finance.mensualite;
    coutTotal = finance.cout_total;
    interets = finance.interets;
  }
private:
  int nombreMensualites;
  double mensualite, coutTotal, interets;
};

#endif // FINANCEMENT_H
```

Communication cryptée entre deux processus distants l'un en Rust l'autre en C++

Dans tous les paragraphes précédents, nous avons réalisé des projets intégrant des modules avec des langages différents dans le même processus, ce qui pose problème lorsque nous devons créer des applications Android. Finalement, l'idéal est de garder le processus **Android** avec la librairie **Qt** et le développement interne en **C++** et d'avoir un autre processus, le service, avec le langage **Rust** qui lui s'occupe de la logique métier.

Comme chaque processus possède son propre langage de programmation, c'est beaucoup plus facile à développer sans contrainte d'intégration entre modules. À priori, cela ne pose aucun problème lorsque les processus sont écrits dans des langages différents. La seule contrainte se situe au niveau de la communication en respectant le protocole entre le service et le client.

*Mon objectif est de créer une application cliente **Android** qui permet de gérer tous les comptes à distance qui permettrait de compléter l'application client sur **PC** réalisé dans le langage **Rust**.*

*Comme il s'agit d'une gestion de comptes avec mémorisation des mots de passe, il est absolument nécessaire d'avoir une communication cryptée avec éventuellement un cryptage symétrique, mais aussi un cryptage asymétrique avec échange des clés publiques. Nous avons déjà réalisé ce type de communication dans le projet de gestion de comptes avec les deux processus écrit en **Rust**.*

*Pour **Rust** tout est intégré. Il suffit de trouver une librairie en **C++** qui permette de réaliser une communication réseau avec la possibilité de crypter ses messages. Il existe pour cela la librairie **Chilkat** qui existe également pour différentes plateformes : **PC** sous **linux**, **Windows** et **MacOs.** Il en existe aussi pour **Raspberry** et **Android** entre autre.*

*Pour valider notre sujet d'étude, je vous propose de réaliser un projet rudimentaire qui permet de valider les échanges cryptés entre un processus écrit **Rust** et un autre écrit en **C++** qui utilise cette librairie **Chilkat**. L'échange des clés se fera à l'aide d'un cryptage symétrique, l'échange des messages en format **JSON** seront cryptés à l'aide des clés publiques échangées.*

*Commençons par créer le processus qui implémente le **service** écrit en langage **Rust**. Nous avons besoin des dépendances associées au **cryptage** et au formatage en **JSON**.*

Cargo.toml

```
[package]
name = "rust-base64"
version = "0.1.0"
edition = "2021"

[dependencies]
openssl = {version="0.10.42", features=["vendored"]}
serde = {version="1.0.152", features=["derive"]}
serde_json = "1.0.93"
```

chiffrement.rs

```
use openssl::rsa::{Padding, Rsa};
use openssl::base64::{encode_block, decode_block};
use std::net::TcpStream;
use std::io::{Read, Write};

pub struct Cryptage {
  pub pair: TcpStream,
  cle_privee: Vec<u8>,
  cle_publique_locale: Vec<u8>,
  cle_publique_pair: Vec<u8>,
  bits: usize
}

impl Cryptage {
  pub fn generer(pair: TcpStream, bits: u32) -> Cryptage {
    let rsa = Rsa::generate(bits).unwrap();
    Cryptage {
      pair,
      cle_privee: rsa.private_key_to_pem().unwrap(),
      cle_publique_locale: rsa.public_key_to_pem().unwrap(),
      cle_publique_pair: vec![],
      bits: bits as usize
    }
  }

  pub fn envoyer_cle_publique(&mut self) {
    let codage_cle_publique = encode_block(self.cle_publique_locale.as_slice());
    self.pair.write(codage_cle_publique.as_bytes()).unwrap();
  }

  pub fn recuperer_cle_publique(&mut self) {
    let mut recuperation = vec![0; self.bits];
    let taille = self.pair.read(recuperation.as_mut_slice()).unwrap();
    let trame = String::from_utf8_lossy(&recuperation[..taille]);
    self.cle_publique_pair = decode_block(&trame).unwrap();
```

```
    println!("{}", String::from_utf8_lossy(self.cle_publique_pair.as_slice()));
  }

  pub fn crypter(&mut self, message: String) {
    let rsa =  Rsa::public_key_from_pem(self.cle_publique_pair.as_slice()).unwrap();
    let mut cryptage = vec![0; rsa.size() as usize];
    let _ = rsa.public_encrypt(message.as_bytes(), &mut cryptage, Padding::PKCS1).unwrap();
    self.pair.write(cryptage.as_slice()).unwrap();
  }

  pub fn decrypter(&mut self) -> String {
    let mut trame= vec![0; self.bits];
    let nombre = self.pair.read(trame.as_mut_slice()).unwrap();
    let rsa = Rsa::private_key_from_pem(self.cle_privee.as_slice()).unwrap();
    let mut recuperation = vec![0; self.bits];
    let taille = rsa.private_decrypt(&trame[..nombre], &mut recuperation, Padding::PKCS1).unwrap();
    String::from_utf8_lossy(&recuperation[..taille]).to_string()
  }
}
```

main.rs

```
mod chiffrement;

use std::net::TcpListener;
use chiffrement::Cryptage;
use serde::{Serialize, Deserialize};

#[derive(Debug, Serialize, Deserialize)]
struct Echange {
  nom: String,
  prenom: String,
  age: u32,
  telephones: Vec<String>
}

fn main() {
    let adresse = "0.0.0.0";
    let port = 1234;
    match TcpListener::bind((adresse, port)) {
        Ok(service) => {
            match service.accept() {
                Ok((client, adresse)) => {
                    println!("Nouveau client [adresse : {}]", adresse.ip().to_string());
                    let mut cryptage = Cryptage::generer(client, 1200);
                    cryptage.recuperer_cle_publique();
                    cryptage.envoyer_cle_publique();
                    let decryptage = cryptage.decrypter();
                    let requete : Echange = serde_json::from_str(decryptage.as_str()).unwrap();
                    let reponse = Echange {
                        nom: requete.nom.to_uppercase(),
                        prenom: format!("{}{}", requete.prenom[..1].to_uppercase(), requete.prenom[1..].to_lowercase()),
                        age: requete.age,
                        telephones: requete.telephones.to_vec()
                    };
                    println!("{:?}", &requete);
                    println!("{:?}", &reponse);
                    cryptage.crypter(serde_json::to_string(&reponse).unwrap());
                },
                Err(_) => println!("Un client n'a pas réussi à se connecter")
            }
        },
        Err(_) => println!("Le service ne peut pas s'activer (déjà activé)")
    }
}
```

*Nous avons déjà écrit plusieurs fois ce type code dans les études précédentes. Il est à noter qu'il est très important de bien choisir le nombre de **bits** nécessaires pour la génération des **clés de cryptage** sachant que les messages à coder ne devront pas dépasser cette limite. Ici nous avons choisi **1200** bits. Les messages **JSON** ne devront pas dépasser **150** octets.*

*La deuxième remarque concerne le source **« chiffrement.rs »**. Cette fois-ci, je passe par un **vecteur** et non plus par un **slice** pour représenter les trames de réceptions venant du client. L'intérêt de ce choix est que la taille de la trame devient paramétrable en correspondance avec le nombre de bits choisi pour la génération des clés. Ce qui ne peut se faire avec au travers d'un **slice** puisque la dimension à proposer doit être impérativement une constante.*

Résultat dans la console d'exécution du service

```
Nouveau client [adresse : 192.168.1.23]
-----BEGIN PUBLIC KEY-----
```

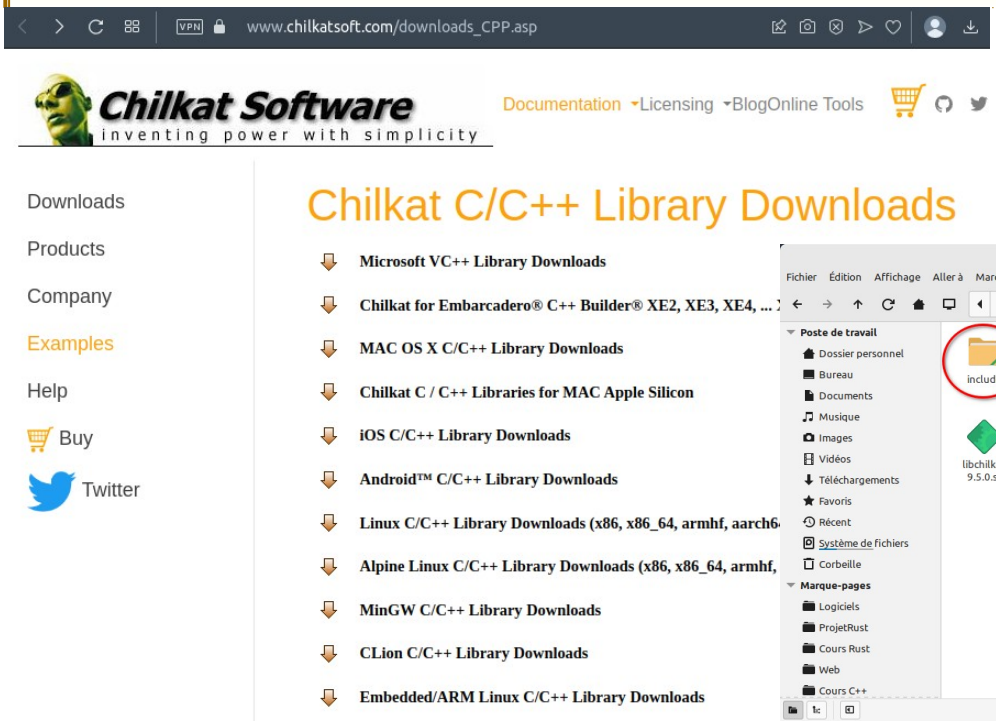
```
MIG1MA0GCSqGSIb3DQEBAQUAA4GjADCBnwKBlwDGs04cddRJVqYfijraPtHvtFV+
DJgzCRt8ZT9xE0gF8m20tw2Hlif19Twqi4Q6+2ExwMFhxBwkf/FdqijclkPxug2Z
EvZQ5U5A/pQvvvVt4yigtckKjNiw5NuTvsX4Hi8eweKLKeldLAT4TSsMNmK7b7x0
PAEGgudmCrNoyxdZQ2hpLZwDo37I74xSaD+BBaFRd86eLMsCAwEAAQ==
-----END PUBLIC KEY-----

Echange { nom: "rémy", prenom: "emmanuel", age: 63, telephones: ["06-89-45-56-22", "04-89-56-32-99"] }
Echange { nom: "RÉMY", prenom: "Emmanuel", age: 63, telephones: ["06-89-45-56-22", "04-89-56-32-99"] }
```

*Je vous propose maintenant de visualiser le code du client exprimé dans le langage **C++** avec l'utilisation de la librairie **Chilkat**. Pour ce projet, nous utilisons le **C++** classique sans la librairie **Qt**. Notre développement se fait à l'aide de deux sources, le premier représente une classe **Cryptage** qui implémente le fonctionnement d'une communication réseau avec le cryptage associé et le formatage des messages en **JSON**. Le deuxième est le source principal décrit dans le fichier classique **« main.cpp »**.*

cpp-openssl.pro

```
TEMPLATE = app
CONFIG += console c++17
CONFIG -= app_bundle qt

HEADERS += cryptage.h
SOURCES += main.cpp
LIBS += libchilkat-9.5.0.a
```

*Pour faire fonctionner ce projet, nous avons besoin de récupérer la bonne librairie **Chilkat** suivant la plateforme qui va contenir l'exécutable. Voici le lien ci-dessous qui vous permet de faire votre choix :*



*Une fois que le bon paquet est récupéré, vous devez le désarchiver, récupérer à la fois la bibliothèque et l'ensemble des fichiers inclus pour les placer dans le projet de développement. Voici ci-dessous la description de la classe **Cryptage** :*

Cryptage.h

```
#ifndef CRYPTAGE_H
#define CRYPTAGE_H

#include "include/CkRsa.h"
#include <include/CkPublicKey.h>
#include <include/CkString.h>
#include <include/CkSocket.h>
#include <include/CkByteData.h>
#include <include/CkJsonObject.h>
#include <include/CkJsonArray.h>

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

struct Identite {
  CkString nom, prenom;
  int age;
  vector<CkString> telephones;
```

```
};
class Cryptage
{
  CkRsa rsa;
  CkPublicKey publique;
  CkSocket& connexion;
public:
  Cryptage(CkSocket& pair) : connexion(pair) {}

  void recuperer_cle_publique() {
    CkString recuperer = connexion.receiveString();
    recuperer.base64Decode("utf-8");
    cout << recuperer.getString() << endl;
    publique.LoadFromString(recuperer.getString());
    rsa.ImportPublicKeyObj(publique);
  }

  void envoyer_cle_publique(int bits) {
    rsa.GenerateKey(bits);
    CkString envoyer = rsa.ExportPublicKeyObj()->openSslPem();
    envoyer.base64Encode("utf-8");
    connexion.SendString(envoyer.getString());
  }

  void crypter(CkString message) {
    CkByteData trame;
    rsa.EncryptString(message.getString(), false, trame);
    connexion.SendBytes(trame);
  }

  CkString decrypter() {
    CkByteData trame;
    connexion.ReceiveBytes(trame);
    CkString message;
    rsa.DecryptString(trame, true, message);
    return message;
  }

  CkString creerJson(Identite& identite) {
    CkJsonObject json;
    json.AppendString("nom", identite.nom);
    json.AppendString("prenom", identite.prenom);
    json.AppendInt("age", identite.age);
    json.AppendArray("telephones");
    CkJsonArray* tels = json.ArrayOf("telephones");
    for (CkString& telephone : identite.telephones)  tels->AddStringAt(-1, telephone);
    delete tels;
    json.put_EmitCompact(false);
    json.put_EmitCrLf(false);
    return json.emit();
  }

  Identite lireJson(CkString message) {
    Identite identite;
    CkJsonObject json;
    json.Load(message);
    identite.nom = json.stringOf("nom");
    identite.prenom = json.stringOf("prenom");
    identite.age = json.IntOf("age");
    CkJsonArray* tels = json.ArrayOf("telephones");
    for (int i=0; i<tels->get_Size(); i++) identite.telephones.push_back(tels->stringAt(i));
    return identite;
  }
};

#endif // CRYPTAGE_H
```

main.cpp

```
#include "cryptage.h"

int main()
{
    CkSocket client;
    bool connexion = client.Connect("192.168.1.23", 1234, false, 7000);
    if (connexion) {
      Cryptage reponse(client);
      reponse.envoyer_cle_publique(1200);
      Cryptage requete(client);
      requete.recuperer_cle_publique();
      Identite identite = {"rémy", "emmanuel", 63};
```

```
      identite.telephones.push_back("06-89-45-56-22");
      identite.telephones.push_back("04-89-56-32-99");
      CkString message = requete.creerJson(identite);
      requete.crypter(message);
      cout << message.getString() << endl;
      message = reponse.decrypter().getString();
      cout << message.getString() << endl;
      Identite nouvelle = reponse.lireJson(message);
      cout << nouvelle.nom.getString() << endl;
      cout << nouvelle.prenom.getString() << endl;
      cout << nouvelle.age << endl;
      for (CkString& telephone : nouvelle.telephones)   cout << telephone.getString() << endl;
    }
    else cout << "Impossible de se connecter au service..." << endl;

  return 0;
}
```

*La librairie **Chilkat** possède un nombre considérable de classes préfabriquées pour résoudre tous les types de problème qui peuvent apparaître lors d'une communication réseau. Je vous invite à parcourir le site. Pleins d'exemples d'initiation sont proposés en adéquation avec les différentes solutions que vous souhaitez résoudre.*

*Je ne vais pas tout expliquer, je ne m'intéresse qu'aux classes qui me permettent d'exploiter pleinement mon projet d'étude, les classes représentant la communication réseau, les classes pour le cryptage et les classes pour le formatage **JSON,** sachant que toutes les classes de cette librairie commence systématiquement par le préfixe « **Ck** ».*

*Commençons par la classe **CkString**, qui comme son nom l'indique représente une chaîne de caractères de haut niveau. Elle est capable de représenter n'importe quel type d'encodage de caractères, sachant que c'est « **UTF-8** » qui est utilisé par défaut. Déjà au niveau de cette classe, il est possible de réaliser un cryptage symétrique en **Base64** à l'aide des méthodes respectives **base64Encode()** et **base64Decode()**. Pour retrouver une chaîne **C classique**, il existe également la méthode **getString()**.*

*La classe **CkSocket** représente un point de connexion dans la communication réseau aussi bien côté serveur que côté client. Puisque notre application est une application cliente, c'est dans ce deuxième registre que nous allons l'utiliser. Pour établir la connexion avec le service, vous devez utiliser la méthode **Connect()** avec laquelle vous précisez **@IP**, le numéro de service (numéro de **port**), si vous souhaitez avoir une communication **SSL** et le réglage du **« timeout »** en milliseconde. Cette méthode renvoie une valeur booléenne spécifiant si la connexion a pu avoir lieu ou pas.*

*Pour échanger des informations sur le réseau avec le service connecté, vous pouvez envoyer vos requêtes soit sous forme de texte à l'aide de la méthode **SendString()**, soit sous forme de suite d'octets avec la méthode **SendBytes()**. Dans ce dernier cas, vous devez alors passer par la classe **CkByteData** qui factorise l'ensemble des octets à soumettre. Pour la réponse à la requête, vous avez les méthodes respectives **receivedString()** et **ReceivedBytes()**.*

*Pour le cryptage asymétrique des messages qui transitent sur le réseau, nous avons la classe **CkRsa** qui s'occupe de cela. Vous pouvez aussi manipuler les classes connexes représentant respectivement la clé privée et la clé publique avec **CkPrivateKey** et **CkPublicKey**. Intrinsèquement, la classe **CkRsa** possède ces deux clés afin de permettre le cryptage et le décryptage.*

*Je rappelle que le principe de fonctionnement d'un cryptage asymétrique est que chaque intervenant envoi sa propre **clé publique** à l'ordinateur pair. Ainsi, le serveur et le client s'échangent leurs **clés publiques** afin que l'un et l'autre puissent crypter chacun leur message. Les **clés privées** restent sur chacune des machines afin d'assurer le décryptage du message envoyé.*

*Par exemple, pour envoyer un message au serveur, je le crypte au moyen de la **clé publique** qu'il a envoyé au client, le serveur recevant ce message crypté se sert de **sa clé privée** pour le décrypter et ainsi retrouver le message original. Dans l'autre sens, nous utilisons la même technique, la clé publique du client est sur le serveur, la clé privée est restée chez le client. Dans ce cadre là, nous obtenons deux cryptages différents entre la requête et la réponse, ce qui permet de mieux sécuriser les échanges.*

*C'est la classe **CkRsa** qui réalise le cryptage et le décryptage. Vu l'explication ci-dessus, nous devons créer deux objets de cette classe. Le premier génère les deux clés au moyen de la méthode **GenerateKey()** (en spécifiant le nombre de bits), la clé privée reste chez le client, la clé publique est envoyé au serveur pour que ce dernier envoie les réponses cryptées. Le deuxième objet est généré à partir de la clé publique donnée par le serveur, en se servant de la méthode **LoadFromString()**, afin de pouvoir crypter les requêtes.*

*La classe **CkRsa** possède les méthodes **EncryptBd()**, **EncryptBytes()** et **EncryptString()** pour crypter les messages. Elle possède également les méthodes inverses **DecryptBd()**, **DecryptBytes()** et **DecryptString()** pour décrypter les messages. Puisque nous communiquons avec du texte formaté en **JSON**, ce sont les méthodes **EncryptString()** et **DecryptString()** qui nous intéressent ici.*

Ces méthodes prennent trois paramètres : le message à crypter ou à décrypter, la trame en octets correspondant au cryptage qui est envoyé ou reçu dans le réseau, le dernier paramètre indique si nous passons par la clé privée pour réaliser l'opération souhaité.

*Classiquement, le format **JSON** a besoin de deux classes spécifiques, l'une **CkJsonObject** qui factorise les différents attributs constituant le message, la deuxième **CkJsonArray** qui enregistre les différents valeurs pour un même attribut du format **JSON**.*

*Les méthodes utiles de la classe **CkJsonObject** sont **AppendString()**, **AppendInt()**, **AppendBool()** et **AppendArray()** pour créer le document **JSON**. Même si cela n'est pas utile, vous pouvez décider d'avoir votre document **JSON** décrit sur plusieurs lignes avec des retour à la ligne pour chacun des attributs avec les méthodes **put_EmitCompact()** et **put_EmitCrLf()**. Par défaut, c'est le format compact qui est privilégié.*

*Une fois que tous les attributs sont renseignés nous passons par la méthode **emit()** pour avoir le texte formaté. Pour retrouver l'ensemble des attributs d'un texte formaté en **JSON**, nous passons par la méthode **Load()** et ensuite nous récupérons chaque attribut par les méthodes de lecture **stringOf()**, **IntOf()**, **BoolOf()** et **ArrayOf()**. Voici ci-dessous le résultat des différents échanges avec le serveur distant.*

Résultat dans la console d'exécution du client

```
-----BEGIN PUBLIC KEY-----
MIG1MA0GCSqGSIb3DQEBAQUAA4GjADCBnwKBlwC8T9NP5pF4WsVLt4/OnAxDpjII
5Ts126vrUBkiS5KJG2X2/rRvP5EQ08JagJ8FNe0WtlcraQW9OSGsEgF+dPjv71ig
h4PFfXVtGqwCPltm4WJG3n5YqjxXEda1fb6LZai+gPun2W5ip+FlxuZqIR3Fr2dI
vxqzxPUN6pLn48Ap//4r61xJdsm49XKaPXXCkaKk0ZbLCXsCAwEAAQ==
-----END PUBLIC KEY-----

{
  "nom": "rémy",
  "prenom": "emmanuel",
  "age": 63,
  "telephones": [
    "06-89-45-56-22",
    "04-89-56-32-99"
  ]
}

{"nom":"RÉMY","prenom":"Emmanuel","age":63,"telephones":["06-89-45-56-22","04-89-56-32-99"]}
RÉMY
Emmanuel
63
06-89-45-56-22
04-89-56-32-99
Press <RETURN> to close this window...
```