TP Qt : Debug

Sommaire

Il est fortement conseillé d'utiliser la documentation Qt de référence en français (http ://qt.developpez.com/doc/) ou en anglais (http ://qt-project.org/doc/).

Débugueur

Présentation

Un débogueur (ou débugueur, de l'anglais *debugger*) est un logiciel qui aide un développeur à analyser les *bugs* d'un programme. Pour cela, il permet **d'exécuter le programme pas-à-pas, d'afficher la valeur des variables à tout moment, de mettre en place des points d'arrêt sur des conditions ou sur des lignes du programme** ...

Le programme à débuguer est exécuté à travers le débogueur et s'exécute normalement. Le débogueur offre alors au programmeur la possibilité d'observer et de contrôler l'exécution du programme.

Lire : fr.wikipedia.org/wiki/Débogueur

GNU Debugger

GNU Debugger, également appelé `gdb`, est le débogueur standard du projet GNU. Il est portable sur de nombreux systèmes type Unix et fonctionne pour plusieurs langages de programmation, comme le C et le C++. Il fut écrit par Richard Stallman en 1988. `gdb` est un logiciel libre, distribué sous la licence GNU GPL.

`gdb` permet de déboguer un programme en cours d'exécution (en le déroulant instruction par instruction ou en examinant et modifiant ses données), mais il permet également un débogage post-mortem en analysant un fichier `core` qui représente le contenu d'un programme terminé anormalement.

L'interface de `gdb` est une **simple ligne de commande**, mais il existe des applications qui lui offrent une interface graphique beaucoup plus conviviale (`ddd`, `nemiver`, ...). `gdb` est souvent invoqué en arrière-plan par les environnements de développement intégré (Eclipse, `geany`, ...).

Important : Les programmes C/C++ doivent être compilés avec l'option -g pour pouvoir être débuggés par `gdb`.

Environnement de Développement Intégré (EDI)

Qt Creator est l'environnement de développement intégré dédié à Qt et facilite la gestion d'un projet Qt. Son éditeur de texte offre les principales fonctions que sont la coloration syntaxique, le complètement, l'indentation, etc... **Qt Creator** intègre aussi en son sein les outils Qt Designer et Qt Assistant.

Il intègre aussi un mode débuggage.

Manipulation

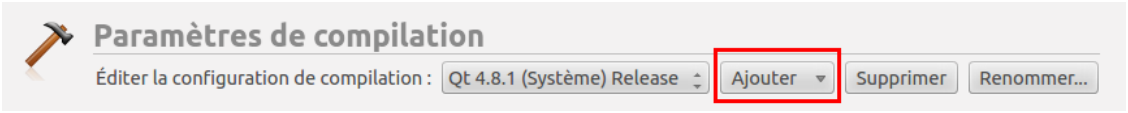
Objectif

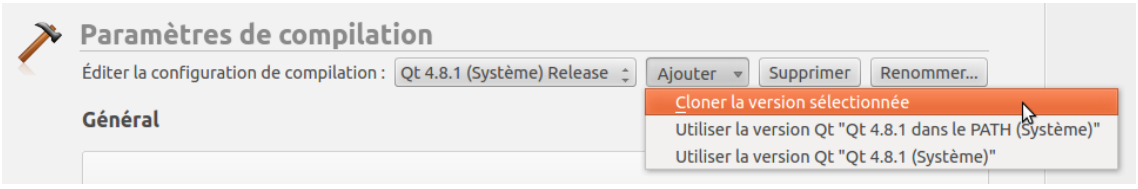
L'objectif de cette manipulation est de mettre en oeuvre de le mode débuggage avec **Qt Creator**.

Étape 1. Lancer **QtCreator**. Ouvrir un projet existant (ou en créer un sinon). Cliquez sur "Projets"

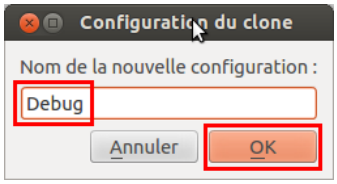
Vérifier si une configuration de compilation pour le mode debug existe. Sinon, passez à l'étape suivante pour en créer une.

Étape 2. On va ajouter une configuration de compilation pour le mode `debug` en cliquant sur "Ajouter" puis en sélectionnant "Cloner la version sélectionnée".

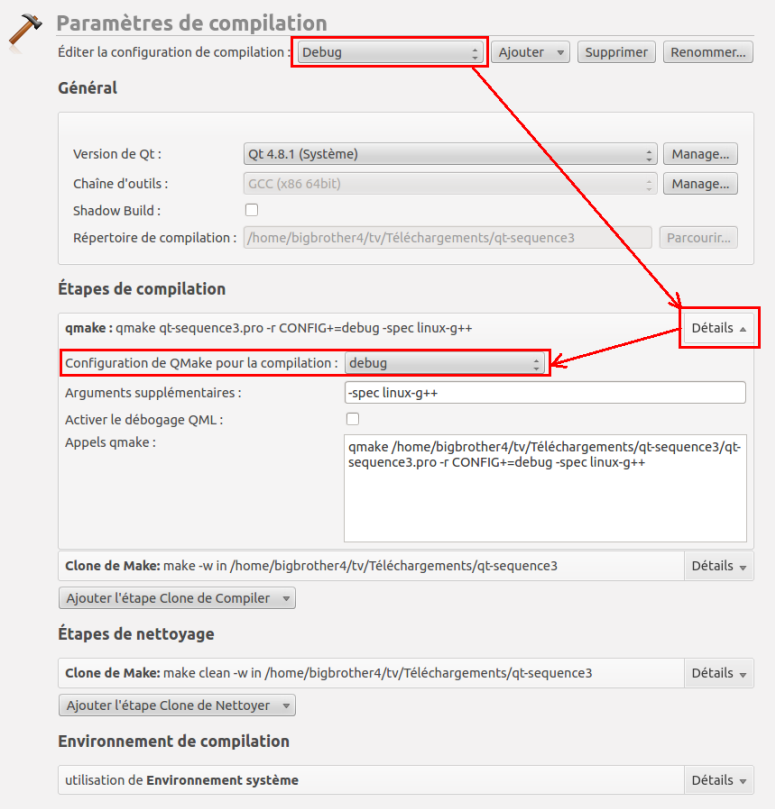




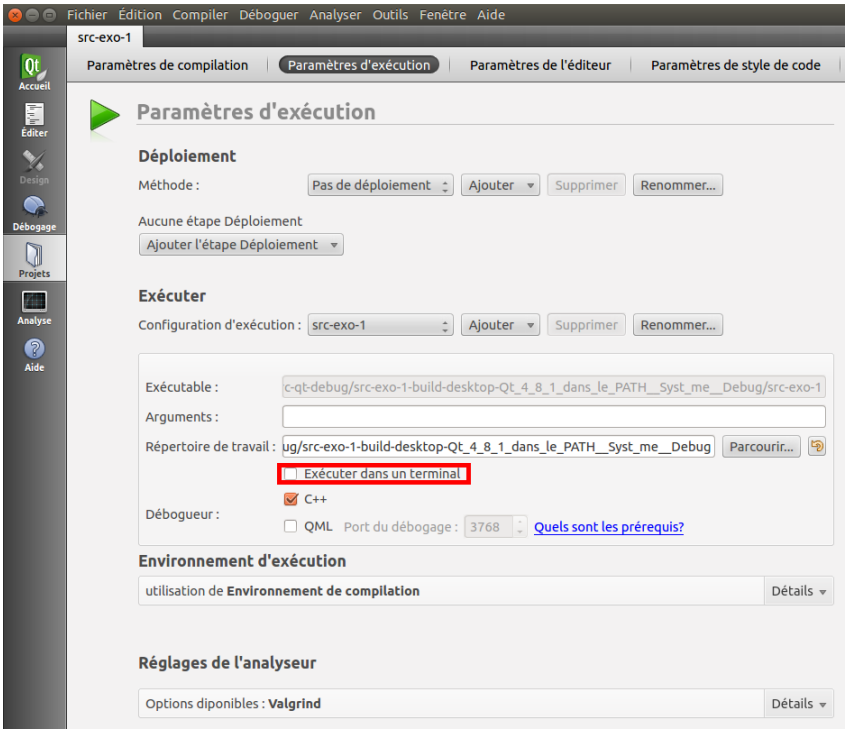
Étape 3. Donner un nom à votre configuration de compilation ...



Étape 4. Paramétrer votre configuration de compilation en choisissant “debug” dans la configuration de **QMake** ...

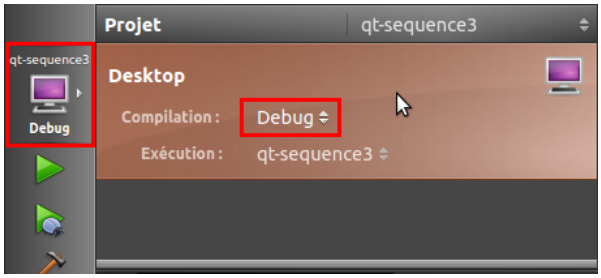


Étape 5. Paramétrer votre configuration d'exécution en *décochant* “Exécuter dans un terminal” ...



Voir l'annexe si vous voulez débuguer dans un terminal, par exemple si vous utilisez des `cout` et des `cin`.

Étape 6. Revenir en mode “Édition” et sélectionner votre configuration de compilation “Debug”...



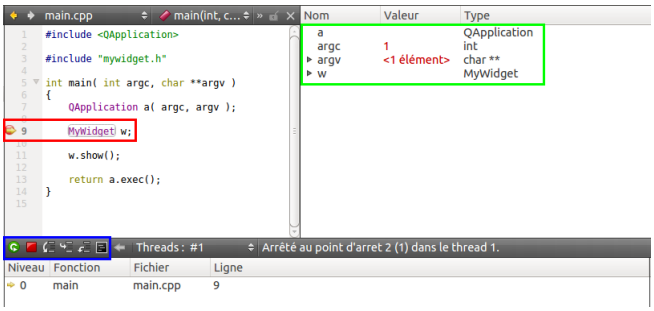
Il vous faudra vraisemblablement recompiler votre application pour prendre en compte l'option de débuggage (-g dans g++ par exemple).

Étape 7. Vous pouvez ajouter des **points d'arrêts** dans votre code source en cliquant sur le numéro de ligne (ici la ligne 9).

```
1  #include <QApplication>
2
3  #include "mywidget.h"
4
5  int main( int argc, char **argv )
6  {
7      QApplication a( argc, argv );
8
9      MyWidget w;
10
11     w.show();
12
13     return a.exec();
14 }
15
```

Étape 8. Puis démarrer l'exécution en mode “déboguage” :

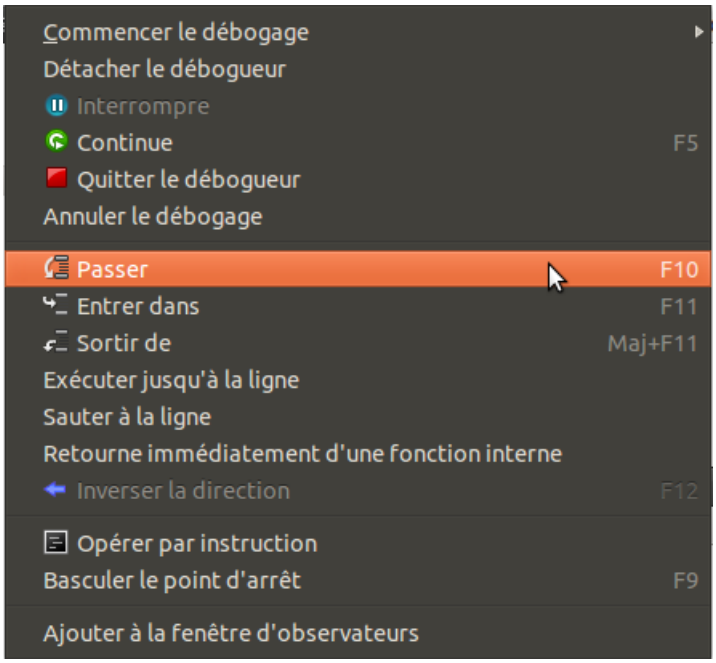
Lance l'exécution en mode `debug` *...*



et le programme s'arrête à la ligne 9

On peut observer le contenu des variables (`a`, `argc`, `argv`, `w`) dans la fenêtre de droite (en vert). On peut par exemple sélectionner le mode d'exécution pas à pas (`F10` ou `F11`), continuer ou arrêter le programme dans la fenêtre du bas (en bleu).

Étape 9. Vous pouvez contrôler le mode `debug` avec le menu “Déboguer”



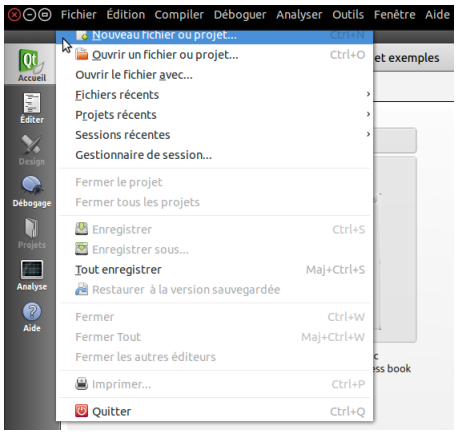
Travail demandé

Mise en situation

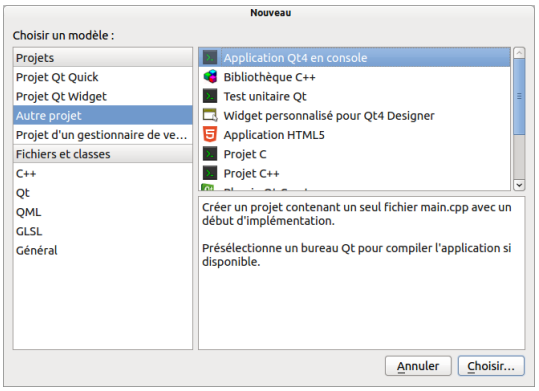
On va créer un **projet Qt** `exo-1` en l'utilisant l'EDI `Qt Creator` :

➠ Démarrer Qt Creator.

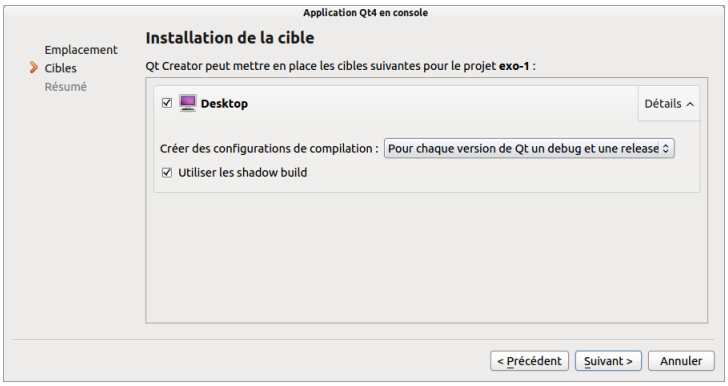
➠ Sélectionner Fichier → Nouveau Projet.



➠ Choisir Autre projet → Application Qt4 en console.

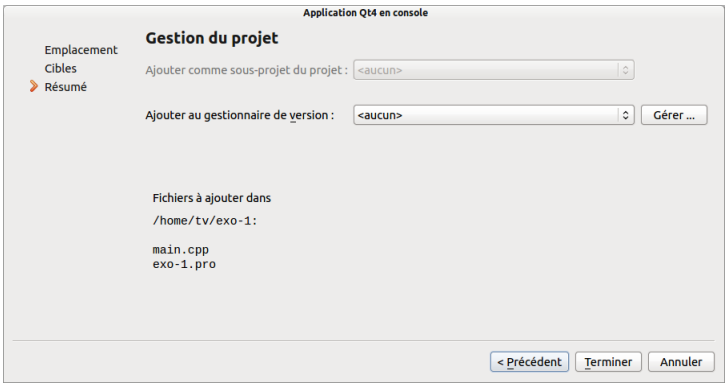


➠ Donner le nom `exo-1` à votre projet (ce sera aussi celui de l'exécutable), choisir son emplacement, puis sélectionner une cible de compilation, choisir la configuration debug et cocher Utiliser les shadow build.

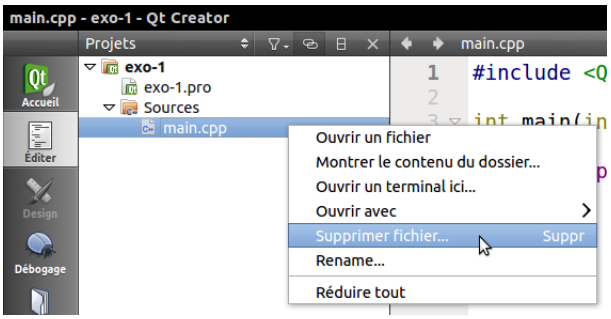


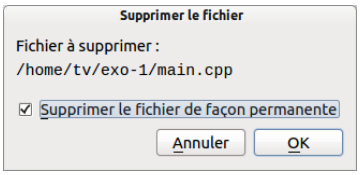
Le *shadow build* permet de séparer les dossiers contenant les codes sources des fichiers générés (exécutable, fichiers objets `.o`, ...) par la fabrication (*build*). Il est important de ne les pas mélanger pour assurer convenablement les tâches courantes (sauvegarde, gestion de versions, portablité multiplateforme, ...). Les fichiers issus de la compilation seront donc stockés dans un répertoire séparé et créé par Qt.

➠ Terminer.

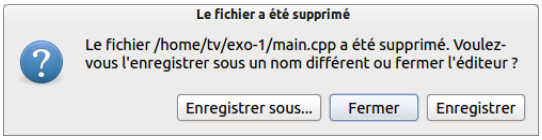


➠ On va **supprimer de façon permanente** le fichier `main.cpp` du projet.



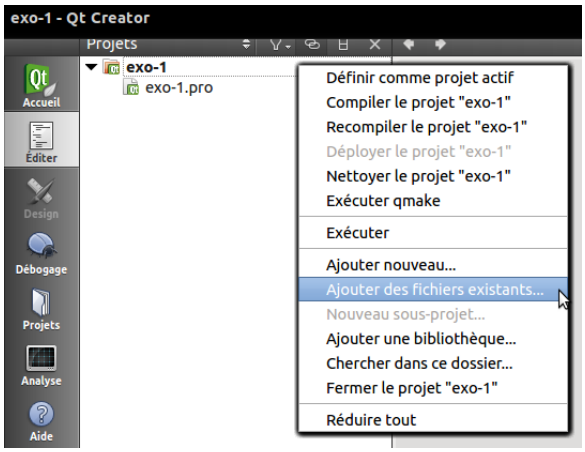


➠ Fermer sans enregistrer.

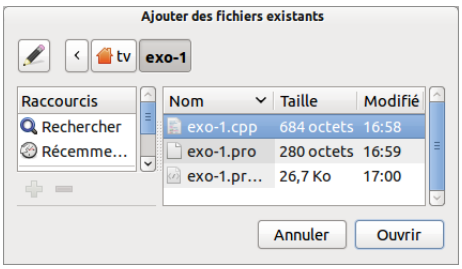


On va maintenant **ajouter le fichier** `exo-1.cpp` fourni avec le TP.

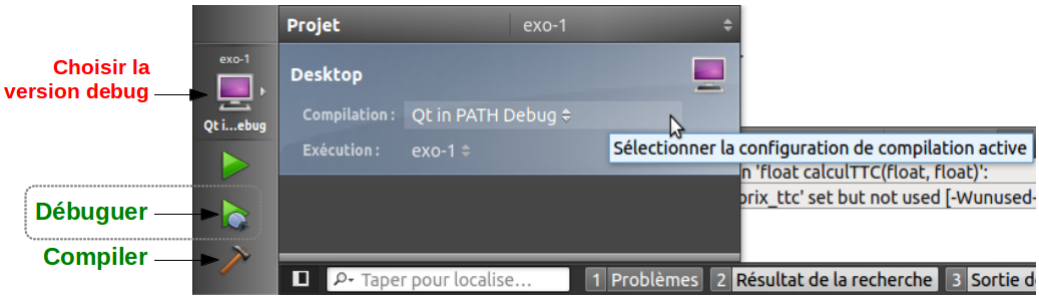
➠ Cliquer avec le bouton droit sur votre projet dans l'explorateur puis sélectionner Ajouter des fichiers existants.



➠ Choisir `exo-1.cpp`.



➠ Sélectionner la version debug.



➠ Commencer le débogage et passer à l'exercice n°1.

Exercice n°1 : prise en main

On désire déboguer le programme `exo-1` qui comporte **trois erreurs** :

```
#include <iostream>

using namespace std;

float calculTTC(float, float);

int main(void)
{
  float prix_ht, prix_ttc, tva;

  cout << "Calcul du prix TTC\n\n";

  cout << "Entrez un prix HT et la TVA : ";
  cin >> prix_ht >> tva;

  prix_ttc = calculTTC(tva, prix_ht);

  cout << "\n" << prix_ht << " euros HT = " << prix_ttc << " euros TTC\n";

  return 0;
}

float calculTTC(float prix_ht, float tva)
{
   float prix_ttc;

   prix_ttc = prix_ht * tva / (100 + prix_ht);
   //prix_ttc = (prix_ht + prix_ht * tva) / 100;

   return prix_ht;
}
```

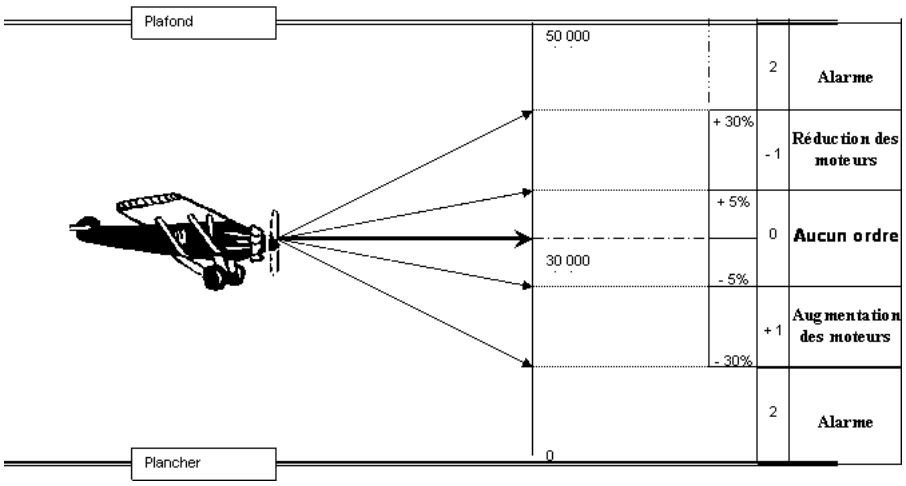
exo-1.cpp

Question 1. En utilisant le débogueur, détecter et corriger les 3 erreurs contenues dans le programme `exo-1.cpp`.

Exercice n°2 : gestion d'un historique d'alarmes

Employé(e) dans une entreprise aéronautique, il est confié à votre service, la lourde responsabilité de valider une partie d'un nouveau système de surveillance destiné à équiper les derniers appareils sortis des ateliers.

Selon l'altitude captée, le système émet une alarme si celle-ci dépasse 39000 pieds ou renvoie des ordres à un autre module chargé de la correction de la vitesse de vol.



Pour des raisons de maintenance du système (analyse de défaillances, ...), il est nécessaire de conserver un **historique** des dépassements horodatés d'altitude.

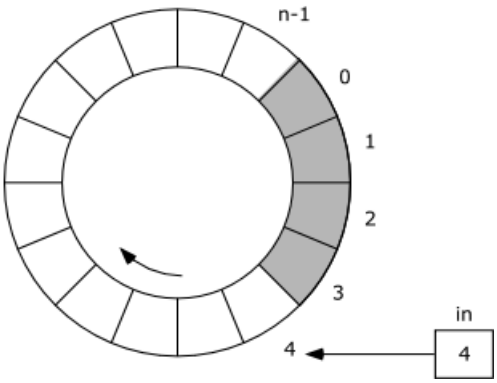
On utilise une structure de données de type `TAlarme` assurant le codage d'une alarme de dépassement d'altitude :

```
typedef long Altitude;

typedef struct
{
   time_t  date;     // timestamp
   Altitude altitude; // la valeur en pieds
} TAlarme;
```

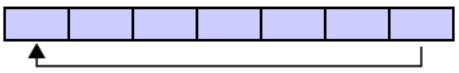
Remarque : le codage de l'horodatage (une date et une heure) est réalisé par la fonction `time()` qui retourne le *timestamp* (de type `time_t`) qui représente le nombre de secondes écoulées depuis le début de l'époque UNIX (1er janvier 1970 00 :00 :00 GMT). Un *timestamp* est très pratique pour réaliser des traitements (calculs) sur des date/heure.

La gestion de l'historique sera assurée par une classe `Historique` possédant les membres privés suivants :
- `unsigned int indice; // index du tableau historique`
- `const unsigned int taille; // le nombre maximum d'alarmes`
- `TAlarme *historique; // le tableau contenant les alarmes`

Le tableau `historique` est en réalité un **tampon circulaire** (*circular buffer*) assurant le stockage des alarmes les plus récentes. Pour réaliser ce tampon (une zone mémoire), on utilise donc un tableau qui permettra de stocker des éléments de type `TAlarme`.



L'utilisation circulaire d'un tampon permet de boucler la fin de la zone vers le début de celle-ci. Cela garantit aussi que l'on accède toujours à l'intérieur de la zone (on évitera ainsi les "erreur de segmentation").



Pour gérer ce tampon circulaire, on a besoin d'un index "`indice`" qui :
– permet le comptage des alarmes depuis l'initialisation,
– donne accès en permanence à l'emplacement suivant la dernière alarme enregistrée.

Voici l'ébauche de la classe `Historique` :

```
class Historique
{
   private:
      unsigned int indice;
      const unsigned int taille;
      TAlarme *historique;

   public:
      Historique(int taille=LG_HISTORIQUE);
      ~Historique();

      unsigned int getNbAlarmes() const;
      unsigned int getTaille() const;

      void ajouterAlarme(Altitude altitude);

      void afficherDerniereAlarme() const;
};

Historique::Historique(int taille/*=LG_HISTORIQUE*/) : indice(0), taille(taille)
{
   historique = new TAlarme[taille];
}
```

```
Historique::~Historique()
{
   delete [] historique;
}

unsigned int Historique::getNbAlarmes() const
{
   return indice;
}

unsigned int Historique::getTaille() const
{
   return taille;
}
```

C'est la méthode `ajouterAlarme(long altitude)` qui permettra d'enregistrer une nouvelle **alarme** de dépassement d'altitude dans l'`historique`.

Rappel des besoins :
– connaître le nombre des alarmes depuis l'initialisation,
– avoir accès en permanence à l'emplacement suivant la dernière alarme enregistrée.

Question 2. Les extraits de code ci-dessous proposent des solutions d'enregistrement dans le tableau "`historique`". Pour chaque extrait de code, préciser s'il répond aux besoins, justifiez en cas de réponse négative. **L'utilisation d'un débogueur est conseillée**.

```
// Solution 1 :
void Historique::ajouterAlarmeSolution1(long altitude)
{
   historique[indice].date = time(NULL);
   historique[indice].altitude = altitude;
   indice++;
}

// Solution 2 :
void Historique::ajouterAlarmeSolution2(long altitude)
{
   if (indice >= taille ) indice = 0;
   historique[indice].date = time(NULL);
   historique[indice].altitude = altitude;
   indice++;
}

// Solution 3 :
void Historique::ajouterAlarmeSolution3(long altitude)
{
   historique[indice].date = time(NULL);
   historique[indice].altitude = altitude;
   if (indice++ > taille ) indice = 0;
}

// Solution 4 :
```



```
void Historique::ajouterAlarmeSolution4(long altitude)
{
   historique[indice % LG_HISTORIQUE].date = time(NULL);
   historique[indice % LG_HISTORIQUE].altitude = altitude;
   indice++;
}
```

Les informations maintenues dans l'historique sont susceptibles d'être remontées vers un **système de supervision**. On doit pouvoir disposer de fonctions capables d'interroger son historique.

Question 3. Développer en quelques lignes la fonction `afficherDerniereAlarme()` permettant l'affichage de la dernière alarme horodatée dans l'historique. Pour afficher le `timestamp` de manière humainement lisible, vous pouvez utiliser la fonction `ctime()` ou `strftime()`.

Le programme d'essai :

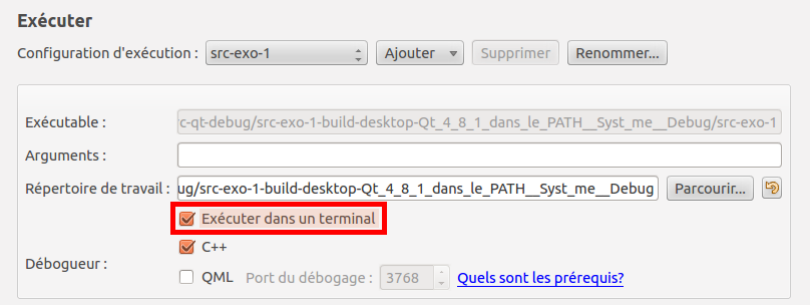
```
#include <QtCore/QCoreApplication>
#include <QTime>

#include <iostream>
#include <ctime>
#include <clocale>
#include <unistd.h>

#include "historique.h"

#define NB_MESURES 200 /* pour les essais : changer cette valeur ! */

using namespace std;

Altitude mesurer(long min, long max)
{
    return static_cast<Altitude>(min + (static_cast<float>(qrand()) / RAND_MAX * (max - min
        + 1)));
}

int main(int argc, char **argv)
{
    QCoreApplication a(argc, argv);

    Historique historique;
    Altitude altitude;
    int i, nb = 0;
    time_t init;
    char horodatage[32];

    qsrand(QTime::currentTime().msec());
    setlocale(LC_TIME, "fr_FR");
    init = time(NULL);
    strftime(horodatage, 32, "%d/%m/%Y à %T", std::localtime(&init));
    cout << "Démarrage du système de journalisation le " << horodatage << "\n\n";

    /* fabrique un historique */
    for(i = 0; i < NB_MESURES; i++)
```

```
    {
        altitude = mesurer(31500, 40000); // simulation
        if(altitude >= SEUIL_ALTITUDE_ALARME)
        {
            historique.ajouterAlarme(altitude);
          //historique.ajouterAlarmeSolution1(altitude);
          //historique.ajouterAlarmeSolution2(altitude);
          //historique.ajouterAlarmeSolution3(altitude);
          //historique.ajouterAlarmeSolution4(altitude);

          cout << "L'information TAlarme { " << altitude << " pieds } a été ajoutée à l'
              historique\n";
          ++nb;
        }
    }

    cout << "\nNombre d'alarmes : " << historique.getNbAlarmes() << " dans l'historique\n";
    cout << "Nombre d'alarmes : " << nb << " au total\n\n";

    historique.afficherDerniereAlarme();

    //historique.afficher(); // affiche l'historique complet

    return 0;
}
```

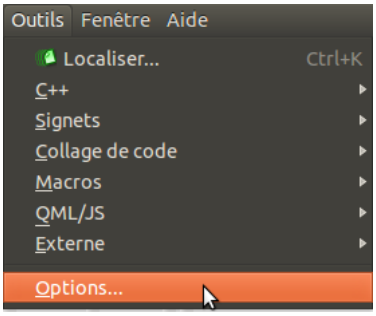
exo-2.cpp

Annexe : débuguer dans un terminal

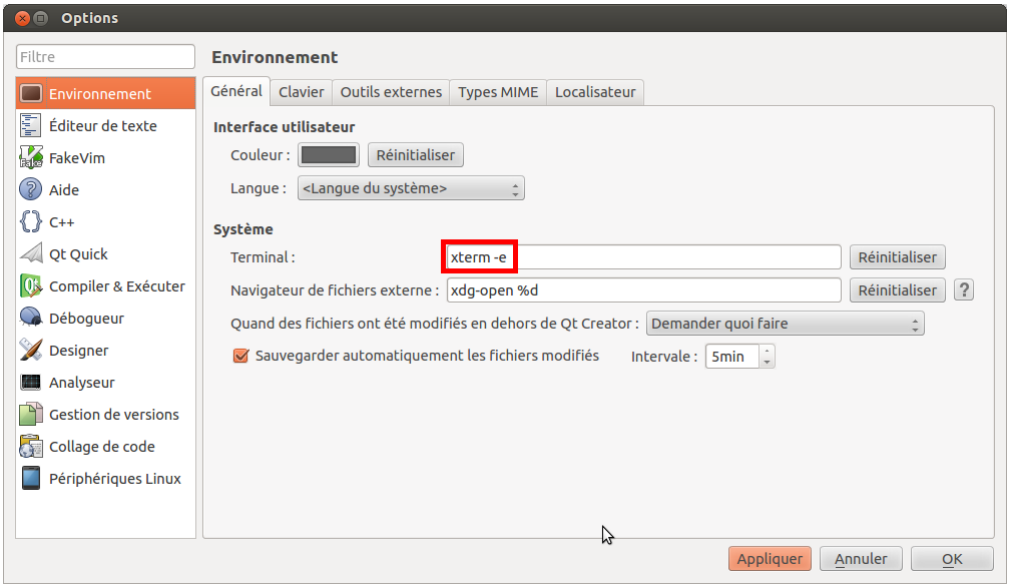
Paramétrer votre configuration d'exécution en *cochant* "Exécuter dans un terminal" ...



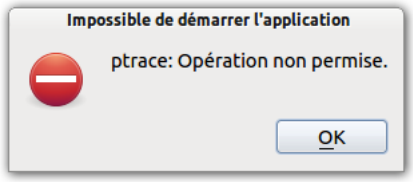
Puis aller dans Outils → Options :



Saisir comme Terminal : `xterm -e`



Si vous rencontrez cette erreur :



Il vous faut exécuter cette commande pour autoriser `ptrace` :

```
$ echo 0 | sudo tee /proc/sys/kernel/yama/ptrace_scope
```

