Le ***C++11*** permet maintenant de manipuler des tâches concurrentes. Grâce à cette étude, nous allons voir comment réaliser ces traitements en parallèle. Afin de maîtriser ces différents concepts, il est souhaitable de bien connaître la technologie des foncteurs et des expressions lambdas qui ont été abordés lors de l'étude précédente.

## Les threads (C++11)

Actuellement, toutes machines, qu'elles soient monoprocesseur ou multiprocesseur, permettent d'exécuter plus ou moins simultanément plusieurs programmes (on parle encore de tâches ou de processus). Sur les machines monoprocesseur, la simultanéité, lorsqu'elle se manifeste, n'est en fait qu'une illusion. À un instant donné, un seul programme utilise les ressources de l'unité centrale, mais l'environnement « passe la main » d'un programme à un autre à des intervalles de temps suffisamment courts pour donner l'impression de la simultanéité ; ou encore, l'environnement profite de l'attente d'un programme pour donner la main à un autre.

***C++11 permet d'appliquer cette possibilité de multiprogrammation au sein d'un même programme dont on dit alors qu'il est formé de plusieurs thread indépendants. Le contrôle de l'exécution de ces différents threads (c'est-à-dire la façon dont la main passe de l'un à l'autre) se fera alors au niveau du programme lui-même et ces threads pourront facilement communiquer entre eux et partager des données.***

Vous pouvez réaliser le traitement d'un thread indépendamment avec une fonction classique, un foncteur ou même une expression lambda. Nous pouvons ensuite demander d'exécuter l'une de ces fonctions (foncteurs ou lambdas) sous forme d'un thread, en passant la fonction concernée en argument d'un constructeur d'un objet de type ***thread*** (déclarations dans le fichier en-tête ***<thread>***). Durant cette phase de construction, il est également possible de transmettre des arguments à la fonction représentant le thread, à la suite du nom de la fonction. Enfin, lorsque nous avons besoin de s'assurer qu'un thread a bien fini son exécution, il suffit d'utiliser la méthode ***join()***.

***Je vous propose de réaliser notre première expérience, en ne prévoyant pour l'instant qu'un affichage simultané, d'une suite de nombre entiers, entre la fonction principale qui lance le thread et la fonction qui traite le thread. Après l'affichage de chaque nombre, nous effectuons une pose d'un dixième de seconde que ce soit sur la fonction main() ou sur la fonction du thread.***

**multitache.cpp**

```cpp
#include <iostream>
#include <thread>
#include <vector>
#include <unistd.h>
using namespace std;

void fonction(vector<int> nombres) // traitement du thread
{
  for (int nombre : nombres)  {
    cout << "Dans la fonction --> Nombre = " << nombre << endl;
    usleep(100000);
  }
}


int main()
{
   vector<int> nombres = {12, -9, 56, 18, -89, 5, 7};
 // lance le thread 'fonction' en lui
 // fournissant l'argument nombres
   thread tache(fonction, nombres);
   for (int nombre : nombres)  {
     cout << "Dans main --> Nombre = " << nombre << endl;
     usleep(100000);
   }
   tache.join();
   return 0;
}
```

```
Dans main --> Nombre = Dans la fonction --> Nombre = 1212

Dans la fonction --> Nombre = -9
Dans main --> Nombre = -9
Dans la fonction --> Nombre = 56
Dans main --> Nombre = 56
Dans la fonction --> Nombre = 18
Dans main --> Nombre = 18
Dans main --> Nombre = Dans la fonction --> Nombre = -89-89

Dans la fonction --> Nombre = Dans main --> Nombre = 55

Dans la fonction --> Nombre = Dans main --> Nombre = 77

Appuyez sur <ENTRÉE> pour fermer cette fenêtre...
```

***Notre programme réalise de la programmation concurrente, c'est-à-dire que deux traitements en parallèle s'exécutent simultanément. Sachant que la simultanéité n'est pas réelle, chaque tâche fait une partie de la tâche dans un temps très court (20 ms environ). Vous remarquez que chacune de ces tâches réalise un affichage durant le temps qu'il lui est imparti. Comme ce type de traitement est toujours un peu long, l'affichage prévu dans la fonction principale n'a pas le temps de se terminer que l'affichage du thread est déjà lancé. Chacune des tâches continuent en affichant le premier nombre accolés avec deux retours à la ligne consécutifs, et ainsi de suite.***

Attention, lorsque vous devez réaliser des programmes multi-tâches avec **QtCreator**, vous devez intégrer la librairie qui gère ce genre de processus dans le fichier de projet, comme cela vous est montré ci dessous.

**multitache.pro**

```
TEMPLATE = app
CONFIG += console c++14
CONFIG -= app_bundle qt
LIBS += -pthread

TARGET = multitaches
SOURCES += main.cpp
fonction
```

Voici ci-dessous le même type de traitement, mais en utilisant cette fois-ci un foncteur. Nous profitons de l'occasion pour réaliser un affichage particulier dans le constructeur du foncteur.

**multitache.cpp**

```
#include ...
using namespace std;

struct Tache // tâche sous forme de foncteur
{
  Tache() { cout << "Lancement de la tâche" << endl; }
  void operator()(vector<int> nombres)
  {
    for (int nombre : nombres)  {
      cout << "Dans le foncteur --> Nombre = "
           << nombre << endl;
      usleep(50000);
    }
  }
};

int main()
{
   vector<int> nombres = {12, -9, 56};
   // lance le thread 'Tache' en lui fournissant l'argument nombres
   thread tache(Tache(), nombres);
   for (int nombre : nombres)  {
     cout << "Dans main --> Nombre = " << nombre << endl;
     usleep(50000);
   }
   tache.join();

   return 0;
}
```

```
Lancement de la tâche
Dans main --> Nombre = 12
Dans le foncteur --> Nombre = 12
Dans main --> Nombre = -9
Dans le foncteur --> Nombre = -9
Dans main --> Nombre = 56
Dans le foncteur --> Nombre = 56
Appuyez sur <ENTRÉE> pour fermer cette fenêtre...
```

**Même type de tâche, mais cette fois-ci à l'aide d'une expression lambda.**

```
#include <iostream>
#include <thread>
#include <vector>
#include <unistd.h>
using namespace std;

int main()
{
   vector<int> nombres = {12, -9, 56, 18};
   // lance le thread sous forme de lambda
   thread tache([](vector<int> tousLesNombres) {
     for (int nombre : tousLesNombres)  {
       cout << "Dans le thread --> " << nombre << endl;
       usleep(50000);
     }
   }, nombres); // argument transmis à la tâche

   for (int nombre : nombres)  {
     cout << "Dans main --> " << nombre << endl;
     usleep(50000);
   }
   tache.join();
   return 0;
}
```

```
Dans main --> 12
Dans le thread --> 12
Dans le thread --> Dans main --> -9-9

Dans main --> Dans le thread --> 56
56
Dans le thread --> Dans main --> 18
18
Appuyez sur <ENTRÉE> pour fermer cette fenêtre...
```

## Partage des ressources – verrou (mutex)

Comme deux threads appartiennent au même programme, ils peuvent donc éventuellement partager les mêmes objets. Cet avantage s'accompagne souvent de contraintes dans la mesure où il peut s'avérer nécessaire que deux threads puissent accéder (presque) en même temps au même objet.

***Quelque soit la solution choisie, nous avons toujours rencontré le même problème lorsque nous affichions la liste des nombres dans le programme précédent. Effectivement, les deux tâches concurrentes tentaient d'utiliser l'écran au même moment ce qui provoquait des aléas d'affichage (recouvrement des informations).***

Pour gérer ces situations, C++11 offre la notion de verrou, avec la classe ***mutex*** (abréviation de ***mutual exclusion***). Cette classe possède deux méthodes importantes : ***lock()*** qui permet de bloquer l'utilisation d'une ressource, ainsi le thread qui effectue ce blocage peut l'utiliser autant de temps qu'il le veut jusqu'à ce qu'il décide de libérer cette ressource au moyen de la méthode ***unlock()*** qui permettra alors aux autres threads de l'utiliser à leurs tours. Pendant qu'une ressource est bloquée, les autres threads qui tentent de l'utiliser sont alors placés en attente jusqu'à que le verrou soit débloqué.

*Le code suivant reprend le projet précédent en plaçant un verrou supplémentaire sur chacune des tâches, le thread particulier décrit à l'aide d'une expression lambda, et le thread principal décrit dans la fonction main().*

verrou.cpp

```cpp
#include <iostream>
#include <thread>
#include <mutex>
#include <vector>
#include <unistd.h>
using namespace std;

int main()
{
   vector<int> nombres = {12, -9, 56, 18, -89, 5, 7};
   mutex verrou;

   // lance le thread sous forme de lambda
   thread tache([&verrou](vector<int> tousLesNombres) {
     for (int nombre : tousLesNombres)  {
       verrou.lock();
       cout << "Dans le thread --> " << nombre << endl;
       verrou.unlock();
       usleep(50000);
     }
   }, nombres); // argument transmis à la tâche

   for (int nombre : nombres)  {
     verrou.lock();
     cout << "Dans main() --> " << nombre << endl;
     verrou.unlock();
     usleep(50000);
   }
   tache.join();
   return 0;
}
```

```
Dans main() --> 12
Dans le thread --> 12
Dans main() --> -9
Dans le thread --> -9
Dans main() --> 56
Dans le thread --> 56
Dans main() --> 18
Dans le thread --> 18
Dans main() --> -89
Dans le thread --> -89
Dans main() --> 5
Dans le thread --> 5
Dans main() --> 7
Dans le thread --> 7
Appuyez sur <ENTRÉE> pour fermer cette fenêtre...
```

## Thread en sommeil (C++11)

Nous pouvons mettre un thread « en sommeil » pour une durée déterminée à l'aide de la fonction ***sleep_for()*** intégrée dans l'espace de nom ***this_thread*** (en n'oubliant pas de libérer les éventuels verrous posés par ce thread !). Cet espace dispose également de la fonction ***sleep_until()*** qui permet d'attendre qu'une condition donnée soit réalisée. Enfin, dans l'espace de nom ***chrono***, il existe des intervalles de temps prédéfinis sous forme de classes, comme ***nanoseconds, microseconds, milliseconds, seconds, minutes*** et ***hours*** toutes issues de la classe générique ***duration***. Ces intervalles de temps sont nécessaires pour les deux fonctions ***sleep_for()*** et ***sleep_until()***.

*Voilà comment modifier le code précédent afin de prendre en compte ces nouvelles caractéristiques. Tous ces éléments sont intégrés dans le fichier d'inclusion <thread>.*

sleep.cpp

```cpp
#include <mutex>
#include <vector>
using namespace std;
using namespace std::this_thread;
using namespace std::chrono;

int main()
{
   vector<int> nombres = {12, -9, 56, 18, -89, 5, 7};
   mutex verrou;
   milliseconds attente(50);
   // lance le thread sous forme de lambda
   thread tache([&verrou, &attente](vector<int> tousLesNombres) {
     for (int nombre : tousLesNombres)  {
       verrou.lock();
       cout << "Dans le thread --> " << nombre << endl;
       verrou.unlock();
       sleep_for(attente);
     }
   }, nombres); // argument transmis à la tâche

   for (int nombre : nombres)  {
     verrou.lock();
     cout << "Dans main() --> " << nombre << endl;
     verrou.unlock();
     sleep_for(attente);
   }
   tache.join();
   return 0;
}
```

## Transfert d'information en retour d'un thread

C++11 offre des outils permettant de récupérer le résultat d'un thread, sans avoir à employer explicitement un verrou. Pour cela, nous avons besoin de la classe générique ***future*** qui stocke temporairement le résultat (utilisable dans le futur) lancée par la fonction spécifique ***async()***. Cette fonction exécute une fonction (foncteur ou lambda) proposée en argument avec ses éventuels paramètres, à l'image de ce que nous avons réalisé avec la classe ***thread***. La fonction ***async()*** permet de lancer immédiatement le thread passé en argument ou l'exécution peut être retardée jusqu'à ce que nous ayons besoin du résultat. Ensuite, pour récupérer le résultat, il suffit de faire appel à la méthode ***get()*** de la classe générique ***future***.

*À titre d'exemple, je vous propose de lancer un thread qui effectue un calcul assez long comme la recherche de la suite des diviseurs d'un nombre entier. Pendant ce temps là, la fonction principale prépare l'affichage et récupère le résultat lorsqu'il est prêt.*

**async_future.cpp**

```
Recherche des diviseurs d'un nombre => n = 24
la suite des diviseurs de 24 est [2 2 2 3]
Appuyez sur <ENTRÉE> pour fermer cette fenêtre...
```

```cpp
#include <future>
#include <vector>
#include <algorithm>
#include <math.h>
using namespace std;

vector<int> recherche(int nombre)
{
  vector<int> diviseurs;
  for (int i=2; i <= sqrt((double)nombre); i++)
    if (nombre%i == 0)
    {
       diviseurs.push_back(i);
       nombre /= i;
       i = 1;
    }
  diviseurs.push_back(nombre);
  return diviseurs;
}

int main()
{
  int nombre;
  cout << "Recherche des diviseurs d'un nombre => n = ";
  cin >> nombre;
  future<vector<int>> resultat = async(launch::async, recherche, nombre);
  cout << "la suite des diviseurs de " << nombre << " est [";
  vector<int> diviseurs = resultat.get();
  for_each(diviseurs.begin(), diviseurs.end(), [](int n) { cout << n << ' '; });
  cout << "\b]" << endl;
  return 0;
}
```

*Le premier argument de la fonction async() est launch::async qui permet de lancer la fonction recherche() dans un thread séparé. Il est possible d'omettre ce premier argument, dans ce cas, l'exécution de la fonction recherche() est retardée, elle n'est exécutée qu'au moment où nous désirons connaître le résultat.*

**Autre exemple avec l'utilisation des lambdas**

```cpp
int main()
{
  int nombre;
  cout << "Recherche des diviseurs d'un nombre => n = ";
  cin >> nombre;
  vector<int> diviseurs;

  future<vector<int>> resultat = async(launch::async, [&diviseurs](int nombre) {
    for (int i=2; i <= sqrt((double)nombre); i++)
      if (nombre%i == 0)
      {
         diviseurs.push_back(i);
         nombre /= i;
         i = 1;
      }
    diviseurs.push_back(nombre);
    return diviseurs;
  }, nombre);

  cout << "la suite des diviseurs de " << nombre << " est [";
  for(int n : resultat.get()) { cout << n << ' '; };
  cout << "\b]" << endl;
  return 0;
}
```

## Envoyer un flot d'informations entre thread

Nous pourrions reprendre le projet précédent de telle sorte que nous ayons cette fois-ci deux threads séparés, un pour la recherche de la suite des diviseurs et un autre pour l'affichage. Au lieu d'attendre la liste complète de l'ensemble des diviseurs, cette fois-ci, dès que nous connaissons un diviseur délivré par le thread de recherche, celui-ci est automatiquement récupéré dans le thread d'affichage pour donner ce résultat immédiatement à l'écran sans attendre les autres calculs. Chaque thread est autonome et va ainsi à sa propre vitesse.

***Pour réaliser ce type de traitement où un ensemble de valeurs peut transiter entre deux threads séparés, il est souvent nécessaire d'utiliser une file d'attente (de type FIFO) implémenté par la classe queue de la STL. Dans beaucoup de cas, Il est judicieux de prévoir une variable booléenne qui permet d'indiquer que l'envoi des valeurs est terminé afin que la tâche qui les récupère soit au courant de l'arrêt.***

**synchonisation.cpp**

```
Recherche des diviseurs d'un nombre => n = 228
la suite des diviseurs de 228 est [2 2 3 19]
Appuyez sur <ENTRÉE> pour fermer cette fenêtre...
```

```cpp
#include <iostream>
#include <thread>
#include <queue>
#include <chrono>
#include <math.h>
using namespace std;

int main()
{
  int nombre;
  cout << "Recherche des diviseurs d'un nombre => n = ";
  cin >> nombre;
  queue<int> diviseurs;
  bool fini = false;

  thread recherche([&](int nombre) {
    for (int i=2; i <= sqrt((double)nombre); i++)
      if (nombre%i == 0)
      {
        diviseurs.push(i);
        nombre /= i;
        i = 1;
//        this_thread::sleep_for(chrono::seconds(1)); // Donne le temps de visualiser la synchronisation entre les threads
      }
    diviseurs.push(nombre);
    fini = true;
  }, nombre);

  thread affichage([&]() {
    cout << "la suite des diviseurs de " << nombre << " est [";
    do {
      while (!diviseurs.empty()) {
        cout << diviseurs.front() << ' ';
        cout.flush();
        diviseurs.pop();
      }
    }
    while (!fini);
    cout << "\b]" << endl;
  });

  recherche.join();
  affichage.join();
}
```