SPD : SystÃ¨mes et Programmation DistribuÃ©s

Nicolas Szczepanski

11 février 2019

szczepanski.nicolas@gmail.com

Partie 2 : Programmation Objet C++ et Processus LÃ©gers

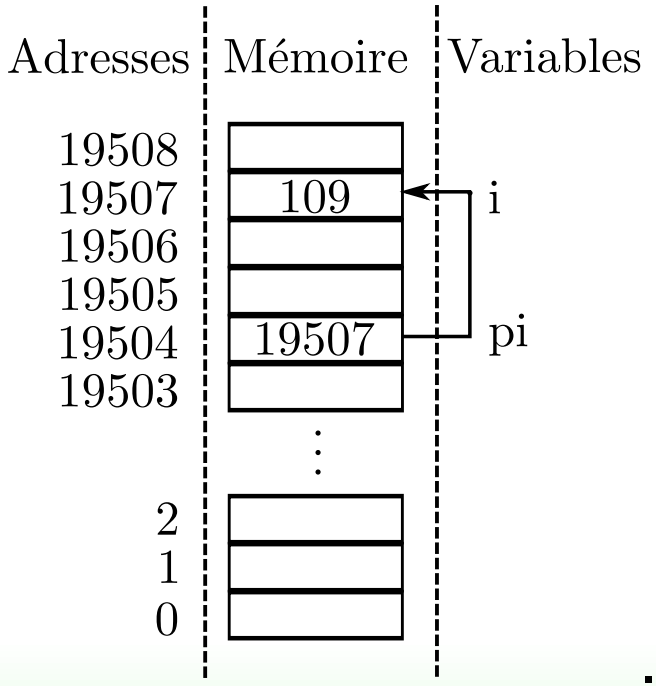
Programmation C++

Les Pointeurs (rappel)

Définition :

Une adresse est une valeur. On peut donc stocker cette valeur dans une variable. Les **pointeurs** sont justement des variables qui contiennent l'adresse d'autres objets, par exemple l'adresse d'une autre variable. On dit que le pointeur fait référence à la variable pointée. Néanmoins, il ne faut pas les confondre avec les références du C++ (voir suite du cours).

```
int i=109;
int* pi=i; //Pointeur sur i
```



Les Pointeurs (rappel)

Définition :

L'opérateur d'**indirection** (&) donne la possibilité d'accéder à l'adresse d'une variable ou d'une méthode ou d'une fonction.

Définition :

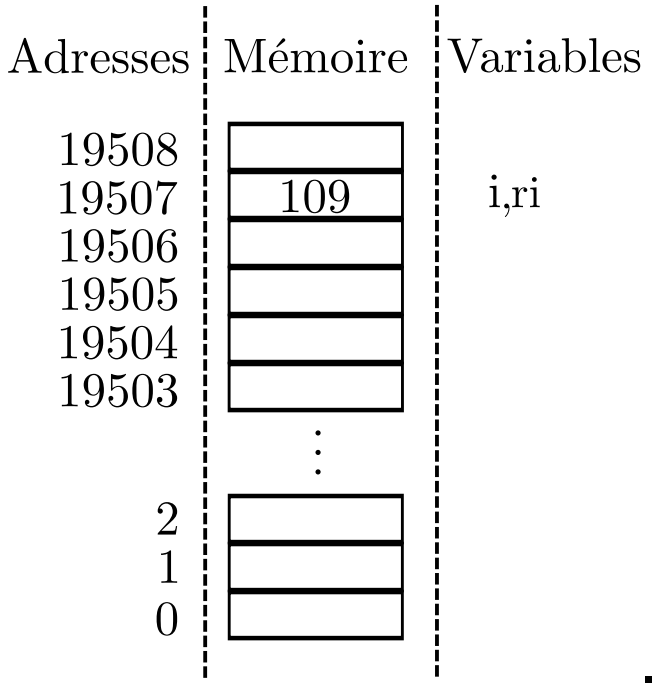
L'opérateur de **déférencement** (*) donne la possibilité d'accéder à l'objet (variable, méthode, ...) référencé par le pointeur afin de le lire ou de le modifier.

Les références

Définition :

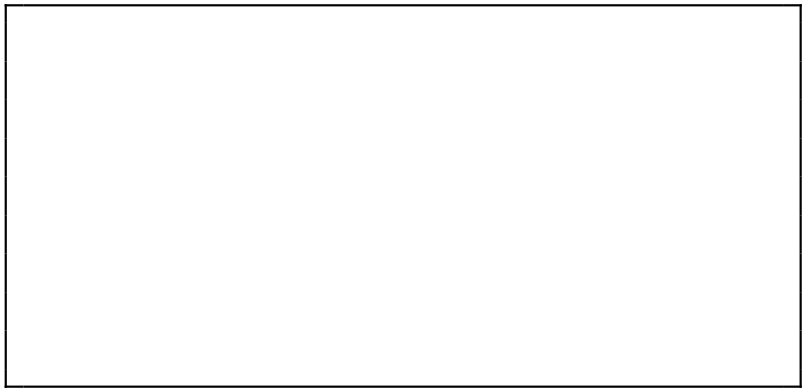
Les **références** sont des synonymes d'identificateurs. Elles permettent de manipuler une variable sous un autre nom que celui sous laquelle cette dernière a été déclarée.

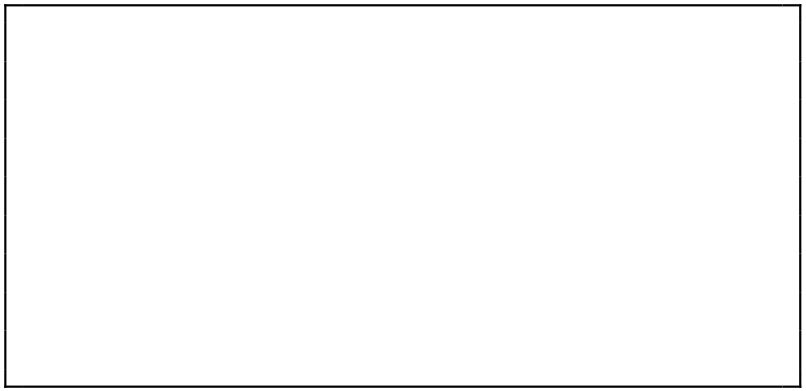
```
int i=109;
int ri=i; //Reference sur i
```



Exercice 1

- Incrémenter une variable i via son pointeur
- Incrémenter une variable i via sa référence





Passage de paramètres

En C++, il y a trois manières différentes de passer une variable (ou un objet) dans une méthode via ses paramètres :

- Passage par valeurs
- Passage par pointeurs
- Passage par référence

```
void test(int i){
 //Passage par valeur
 //Copie /!\
}
```

```
void test(int* i){
 //Passage par pointeur
 //Pas de copie
}
```

```
void test(int& i){
 //Passage par reference
 //Pas de copie
}
```

Exercice 2

Implémenter une méthode qui incrémente la variable passée en paramètre et la fonction « main » qui montre son utilisation pour chaque sorte de passage.

```
//Passage par valeur
```

```
//Passage par pointeur
```

```
//Passage par reference
```

Pointeurs Vs Références

- Un pointeur peut être réaffecté mais pas une référence
  Une référence est assignée uniquement lors de son initialisation
- Un pointeur a sa propre adresse mémoire (4 bytes sur x86) : c'est aussi une variable. En revanche, une référence partage la même adresse (que la variable originale)
- Un pointeur peut être assigné à NULL (ou nullptr), pas une référence

```
int *p = nullptr;
int *f = NULL;
int &r = nullptr; //Erreur de compilation
```

- Un pointeur peut être utilisé pour faire des itérations sur des tableaux, pas une référence

Conclusion : utiliser les références tout le temps sauf quand vous faites des opérations avec les pointeurs (tableau, liste, ...).

Les Constantes (rappel)

- Le mot-clé **const** empêche toute modification ultérieure de la valeur d'une variable.
- Le mot-clé **const** s'applique à ce qui se trouve directement à sa gauche ou, s'il n'y a rien qui le précède dans la déclaration à ce qui se trouve à sa droite.

```
int i, j;
int const* p = &i;
*p = j; //Erreur, la valeur pointee par p est constante.
p = &j; //Correct, le pointeur p n'est pas constant
```

- En C++, une méthode peut être déclarée **const** : cela signifie qu'elle ne modifie pas l'objet sur lequel elle est appelée

Références Vs Références constantes

- Par abus de langage, une « référence constante » est une référence vers un type constant !
- Une référence constante sur un objet
  = Une référence vers un objet constant
  = L'objet ne doit pas être modifier
  = Aucunes variables de l'objet ne doivent être modifiées

```
void setData(BigClass& myBigClass){
  //Objet lourd en memoire
  //Avec modification de l objet
  //=>Passage par reference
}
void analyse(const BigClass& myBigClass){
  //Objet lourd en memoire
  //Sans modification de l objet
  //=>Passage par reference constante
}
```

Références Vs Références constantes

- Une référence non constante ne peux pas être initialisée par un objet temporaire

```
//Ne fonctionne pas !
Bottle operator+(Bottle& bottle1, Bottle& bottle1) {
  Bottle tmp; //Un objet temporaire
  Bottle.size = bottle1.size + bottle2.size;
  return Bottle;
}
```

```
//Fonctionne !
Bottle operator+(const Bottle& bottle1,const Bottle& bottle1) {
  Bottle tmp; //Un objet temporaire
  Bottle.size = bottle1.size + bottle2.size;
  return Bottle;
}
```

```
int main(){
  Bottle b1(5),b2(2),b3(1);
  Bottle b0 = b1 + b2 + b3;
}
```

- Quand un objet est passé par référence constante, il ne peut appeler que les méthodes constantes de l'objet

Références Vs Références constantes

Une bonne habitude est de prendre :

- Tous paramètres non modifiables par références constantes, excepté les types primitifs
- Tous paramètres modifiables par références non constantes sauf pour faire des opérations avec des pointeurs sur certaines structures de données (tableau, liste, . . .)

C++ et <thread>

Lancer des *threads*

```
#include <iostream>
#include <thread>
#include <chrono>
void work1(){
  for (int i = 0; i != 4; i++){
    std::cout << "work1 : i = " << i << std::endl;
    std::this_thread::sleep_for(std::chrono::milliseconds(2000));
  }
}
void work2(){
  for (int i = 0; i != 4; i++){
    std::cout << "work2 : i = " << i << std::endl;
    std::this_thread::sleep_for(std::chrono::milliseconds(1000));
  }
}

int main()
{
  std::thread thread1(work1);
  std::thread thread2(work2);
  thread1.join();
  thread2.join();
  return 0;
}
```

g++ work.cpp -pthread -std=c++11 -o work
./work

Arrêter des *threads*

Vous êtes obliger d'utiliser une de ces deux solutions :

- `.detach()` : Détacher le *thread* de son créateur

```
int main(){
  std::thread t(work);
  t.detach();//Vie ta vie
  return 0;
}
```

- `.join())` : Le créateur attend que le *thread* se termine

```
int main(){
  std::thread t(work);
  //Du travail, encore du travail
  t.join();
  return 0;
}
```

Avec `.join()`, vous devez assurer que le *thread* se termine, sinon interblocage !

Partager des données entre *threads*

```
#include<iostream>
#include<thread>
using namespace std;

void f1(int& ret){ret=5;}
void f2(int* ret){*ret=5;}

int main() {
  int ret=0;
  thread t1(f1, ret);
  t1.join();
  cout << "ret=" << ret << endl;
  thread t2(f2, &ret);
  t2.join();
  cout << "ret=" << ret << endl;
}
```

```
ret=0
ret=5
```

Partager des données entre *threads*

```
#include<iostream>
#include<thread>
using namespace std;

void f1(int& ret){ret=5;}
void f2(int* ret){*ret=5;}

int main() {
  int ret=0;
  thread t1(f1, ret);
  t1.join();
  cout << "ret=" << ret << endl;
  thread t2(f2, &ret);
  t2.join();
  cout << "ret=" << ret << endl;
}
```

```
ret=0
ret=5
```

Quel est le problème ?

Partager des données entre *threads*

Le constructeur de `std::thread` déduit les types et les stock par valeur !

Donc tous les arguments dans le constructeur de `std::thread` sont toujours copiés !

Deux solutions :

- `std::ref`

```
int ret=0;
std::thread t1(f1, std::ref(ret));//Ne copie plus la variable
t1.join();
```

- Passer un pointeur

Exercice 3

```
#include<iostream>
#include<thread>

static const int nbThreads=100;

void increment(int& i){
  ++i;
}

int main(){
  int x = 0;
  std::thread t[nbThreads];
  for (int i = 0; i < nbThreads; ++i){
    t[i] = std::thread(increment, std::ref(x));
  }
  for (int i = 0; i < nbThreads; ++i){
    t[i].join();
  }
}
```

Exercice 3

```
#include<iostream>
#include<thread>

static const int nbThreads=100;

void increment(int& i){
  ++i;
}

int main(){
  int x = 0;
  std::thread t[nbThreads];
  for (int i = 0; i < nbThreads; ++i){
    t[i] = std::thread(increment, std::ref(x));
  }
  for (int i = 0; i < nbThreads; ++i){
    t[i].join();
  }
}
```

Quel est le problème ?

Exercice 3 : solution

```
#include<iostream>
#include<thread>

static const int nbThreads=100;
std::mutex m;

void increment(int& i){
  m.lock();
  ++i;
  m.unlock();
}

int main(){
  int x = 0;
  std::thread t[nbThreads];
  for (int i = 0; i < nbThreads; ++i){
    t[i] = std::thread(increment, std::ref(x));
  }
  for (int i = 0; i < nbThreads; ++i){
    t[i].join();
  }
}
```

Et une *race condition* corrigée !

Oublier de `unlock`

C'est une erreur fréquente, surtout dans cette situation

```
std::mutex m;
int main(){
  m.lock();
  if(bidouille1){
    m.unlock();
    return 0;
  }
  if(bidouille2){
    m.unlock();
    return 1;
  }
  m.unlock();
}
```

La solution : `std::lock_guard`

```
std::mutex m;
int main(){
  std::lock_guard<std::mutex> guard(m);
  if(bidouille1)return 0;
  if(bidouille2)return 1;
}
```

Oublier de `unlock`

Définition :

`std::lock_guard` *lock* le mutex passé en paramètre du constructeur à la création de l'objet. Dès que l'objet est détruit (à la fin de sa porté et en fonction des accolades), le mutex est *unlock* « automatiquement ». Dans la porté du `std::lock_guard`, le mutex est protégé et ne peut pas être *lock* ou *unlock* via l'appel aux méthodes `.lock()` et `.unlock()`.

Définition :

`std::unique_lock` a le même effet que `std::lock_guard` sauf que le mutex associé n'est pas protégé. Il peut donc être *lock* ou *unlock* via l'appel aux méthodes `.lock()` et `.unlock()` dans la porté du `std::unique_lock`.

Variable Condition

Définition :

Une variable condition est associé avec un évènement ou une condition. Grâce au variable condition, un *thread* est capable de :

- Changer une condition et notifier les autres threads sur cette condition : `.notify_one()`
- Attendre qu'une condition est satisfaite : `.wait()`

Variable Condition

La méthode `wait` :

```
void wait(std::unique_lock<std::mutex>& lock,Predicate pred);
```

Quand la méthode `wait` est appelé :

Réalise ces trois tâches **atomiquement** :

- *unlock* le mutex
- Ajoute dans une liste le thread current
- Bloque l'exécution du thread current tant que :
  - `notify_one()/notify_all()` n'est pas appelé par la même condition variable
  - la condition `pred` n'est pas satisfaite

Quand le `wait` se réveil :

- *lock* le mutex

Variable Condition

```
cv.wait(lock,pred);
```

est équivalent à :

```
while (!pred()) {
  cv.wait(lock);
}
```

Cela sert à éviter les faux réveils (**spurious wakeup**) : Car un thread peut être réveillé (sortir de `.wait()`) même si aucun thread n'a notifié (`.notify_one()`) la variable condition !

Variable Condition : Notification

La méthode `.notify_one()` :

Tente de réveiller (c'est une **notification**) un seul thread en attente (cela dépend de `pred`)

La méthode `.notify_all()` :

Tente de réveiller (c'est une **notification**) tous les threads en attente (cela dépend de `pred`)

Remarque :

- Il est impossible via une notification de réveiller et de débloquer un thread qui à démarrer son attente juste aprés que `.notify_one()` soit appelé.
- `.notify_one()` n'a pas bessoin de *lock* le mutex.
- *lock* le mutex et notifier peut néenmoins être nécessaire quand un certain ordre des événements est requis !

```
std::condition_variable cv;
std::mutex cv_m;
int i = 0;
bool done = false;
void waits(){
  std::unique_lock<std::mutex> lk(cv_m);
  std::cout << "Attente... \n";
  cv.wait(lk, [](){return i == 1;});
  std::cout << "Fin de l'attente i == 1\n";
  done = true;
}

void signals(){
  std::this_thread::sleep_for(std::chrono::seconds(1));
  std::cout << "Notification...\n";
  cv.notify_one();

  std::unique_lock<std::mutex> lk(cv_m);
  i = 1;
  while (!done) {
    lk.unlock();
    std::this_thread::sleep_for(std::chrono::seconds(1));
    lk.lock();
    std::cerr << "Notification ...\n";
    cv.notify_one();
  }//Le réveil se réalisera seulement dans le prochain tour de
   //boucle, dans lk.unlock();
}

int main(){
  std::thread t1(waits), t2(signals);
  t1.join(); t2.join();
}
```

```
std::condition_variable cv;
std::mutex cv_m;
int i = 0;
bool done = false;

void waits(){
  std::unique_lock<std::mutex> lk(cv_m);
  std::cout << "Attente... \n";
  cv.wait(lk, []{return i == 1;});
  std::cout << "Fin de l'attente i == 1\n";
  done = true;
}

void signals(){
  std::this_thread::sleep_for(std::chrono::seconds(1));
  std::cout << "Notification...\n";
  cv.notify_one();
  std::unique_lock<std::mutex> lk(cv_m);
  i = 1;
  while (!done){
    std::cout << "Notification...\n";
    lk.unlock();
    cv.notify_one();//Le réveil se réalise immédiatement
    std::this_thread::sleep_for(std::chrono::seconds(1));
    lk.lock();
  }//Donc, on va sortir de la boucle tout de suite
}

int main(){
  std::thread t1(waits), t2(signals);
  t1.join();
  t2.join();
}
```

Atomique

Définition :

L'atomicité désigne une opération ou un ensemble d'opérations d'un programme qui s'exécutent entièrement sans pouvoir être interrompues (par un autre processus, par un autre thread, par un signal, ...) avant la fin de leur déroulement. Une opération (ou ces opérations), vérifiant cette propriété, est qualifiée d'**atomique**.

CompareAndSwap

Définition :

L'algorithme CompareAndSwap permet d'écrire dans une variable d'une manière atomique (peut échouer).

- C'est le matériel qui assure l'atomicité de l'algorithme.
- On ne peut donc **pas implémenter** un CompareAndSwap.

```
//Exemple pour un entier
//!\ c'est de l'algorithme, ne peut pas être implémenté.
bool compareAndSwap(int* p, int oldValue, int newValue){
  if(*p != oldValue)return false;//Compare
  *p=newValue;//Swap
  return true;
}
//Utilisation
int x = 5;
std::cout << x << std::endl;//Affiche 5
while(!compareAndSwap(i,x,10));std::cout << x << std::endl;//Affiche 10
```

std::atomic

Définition :

En c++,std::atomic représente des types, sur lesquels différents threads peuvent simultanément opérer sur leurs propres instances sans avoir de comportements indéfinis (race condition).

```
std::atomic<long> value(0);
value++; //Ceci est une opération atomique
value+=5; //Celle la aussi
```

std::memory_order

Définition :

Le comportement par défaut pour toutes les opérations sur les variables atomiques est `memory_order_seq_cst` qui, tout comme le mot-clé volatile de Java, applique la cohérence séquentielle. Cela coûte trop cher.

Définition :

Si nous voulons aucune contraine (aucune synchronisation) mais garder l'atomicité, nous devons utiliser `memory_order_relaxed`.

std::memory_order

```
#include <vector>
#include <iostream>
#include <thread>
#include <atomic>

std::atomic<int> cnt = {0};

void f()
{
  for (int n = 0; n < 1000; ++n) {
    cnt.fetch_add(1, std::memory_order_relaxed);
  }
}

int main()
{
  std::vector<std::thread> v;
  for (int n = 0; n < 10; ++n)v.emplace_back(f);
  for (int n = 0; n < 10; ++n)v[n].join();
  std::cout << "Final counter value is " << cnt << std::endl;
}
```

std::atomic

Question :

Un entier normal (`int`) posséde des opérations `load` et `store` qui sont atomique ? A quoi sert donc `std::atomic<int>` ?

Réponse :

Cela est vrai que pour les architectures offrant une telle garantie d'atomicité. Il y a des architectures qui ne la possède pas.

std::atomic_compare_exchange

```
#include <atomic>

template<class T>
struct node
{
  T data;
  node* next;
  node(const T  data) : data(data), next(nullptr) {}
};

template<class T>
class stack
{
  std::atomic<node<T>*> head;
  public:
  void push(const T data){
    node<T>* new_node = new node<T>(data);
    // Met la valeur courante dans le prochain noeud
    new_node->next = head.load(std::memory_order_relaxed);
    while(!std::atomic_compare_exchange_weak_explicit(
    head,new_node->next,new_node,std::memory_order_release,std::memory_order_relaxed))
  }
};

int main(){
  stack<int> s;
  s.push(1);
  s.push(2);
  s.push(3);
}
```

thread_local

Définition :

- Une variable déclarée `thread_local` est alloué au démarrage du thread et libéré à la fin du thread. Chaque thread a sa propre instance de la variable.
- Doit être utilisé soit avec `static`, soit avec `extern`

```
public:
inline unsigned int getThreadId(){
  thread_local static unsigned int threadId=UINT_MAX;
  //If the id is already calculate, it is ok
  if (threadId!=UINT_MAX)return threadId;
  //Else loop on the threads to calculate the thread id.
  for (unsigned int i = 0;i < threads.size();i++){
    if (threads[i]->get_id() == std::this_thread::get_id()){
      threadId=i;
      return threadId;
    }
  }
  return threadId;
}
private:
std::vector<std::thread*> threads;
```

thread_local

```
#include <iostream>
#include <string>
#include <thread>
#include <mutex>

thread_local unsigned int rage = 1;
std::mutex cout_mutex;

void increase_rage(const std::string thread_name)
{
  ++rage;
  std::lock_guard<std::mutex> lock(cout_mutex);
  std::cout << "Rage counter for " << thread_name << ": " << rage << std::endl;
}

int main()
{
  std::thread a(increase_rage, "a"), b(increase_rage, "b");

  {
    std::lock_guard<std::mutex> lock(cout_mutex);
    std::cout << "Rage counter for main: " << rage << std::endl;
  }

  a.join();
  b.join();
}
```

volatile

Cela permet d'éviter certaines optimisations du compilateur sur la variable. Par exemple :

```
bool x = false;
while(f(x));
```

peut-être optimisé par le compilateur en :

```
while(f(false));
```

Mais `x` peut aussi être modifié par un autre thread ! Et dans ce cas, l'optimisation du compilateur pose problème.

```
volatile bool x = false;
while(f(x));
```

volatile

Créer un programme avec 4 threads où chaque thread se termine quand une variable `bool` est à vrai. Mettre la variable à vrai au bout de 10 secondes dans le `main` (thread principal). Cela fonctionne ? Ajouter volatile à cette variable.

Exercice 4

Il y a deux *threads* :

- La mère nettoie le linge
- L'enfant salit le linge

```
enum class Linge {PROPRE, SALE};

std::mutex mutex;
Linge LingeEnfant = Linge::PROPRE;
std::condition_variable cv;

bool estPropre(){
  return LingeEnfant == Linge::PROPRE;
}
bool estSale(){
  return LingeEnfant == Linge::DIRTY;
}
```

```
int main(){
  std::thread maman(nettoyerLinge);
  std::thread enfant(salirLinge);
  maman.join();
  enfant.join();
}
```

Exercice 5

La course des fainéants : 8 coureurs démarrent une course, mais quand l'un des coureurs a atteint la ligne d'arrivée, tous les autres s'arrêtent. Nous considérons qu'ils doivent parcourir 1000m et que le coureur n avance de x mètres en 10ms avec x un nombre aléatoire entre 0 inclue et 10 inclue. Modéliser ce problème via des threads (coureurs). Il s'agit la d'arrêter tous les threads dès qu'un coureur a atteint 1000m.