

# **Systèmes temps réel**

## **Chapitre 4** **Multithreading**

Multithreading avec POSIX pthread

# **RAPPEL C/C++**

# La notion d'adresse

- Tout élément (variable, constante, instructions,...) manipulé par l'ordinateur est stocké dans sa mémoire.
- Cette mémoire est constituée d'une série de « cases »
- Pour avoir accès à un élément, il faut connaître le numéro de cette case : -> **l'adresse**
- Toute case mémoire a une adresse unique (sur 32 ou 64 bits).

# L'adresse d'une variable

```
#include <iostream.h>
#include <stdio.h>

int main() {

    int toto=45;

    cout<<toto<<endl;           //affiche la valeur 45
    printf("%d \n",toto);       //affiche la valeur 45

    cout<<&toto<<endl;           //affiche l'adresse de toto
    printf("%p \n",&toto);       //affiche l'adresse de toto

    return 0;
}
```

# Le pointeur

**Une adresse est une valeur.**

On peut stocker cette valeur dans une variable : les **pointeurs** sont des variables qui contiennent des adresses

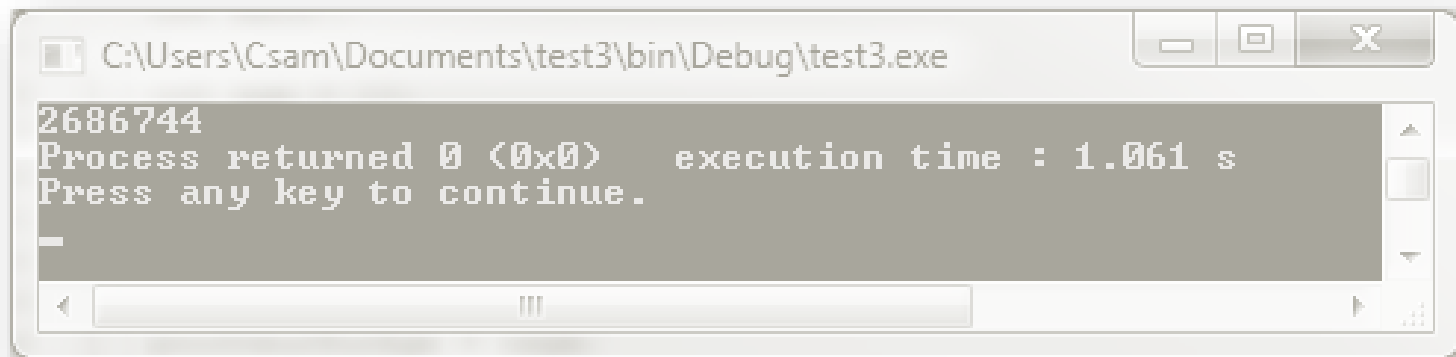
On dit que **le pointeur pointe sur la variable pointée**

La valeur d'un pointeur peut changer. *Cela ne signifie pas que la variable pointée est déplacée en mémoire, mais plutôt que le pointeur pointe sur autre chose.*

**Les pointeurs disposent d'un type** (qui correspond généralement au type de la variable pointée)

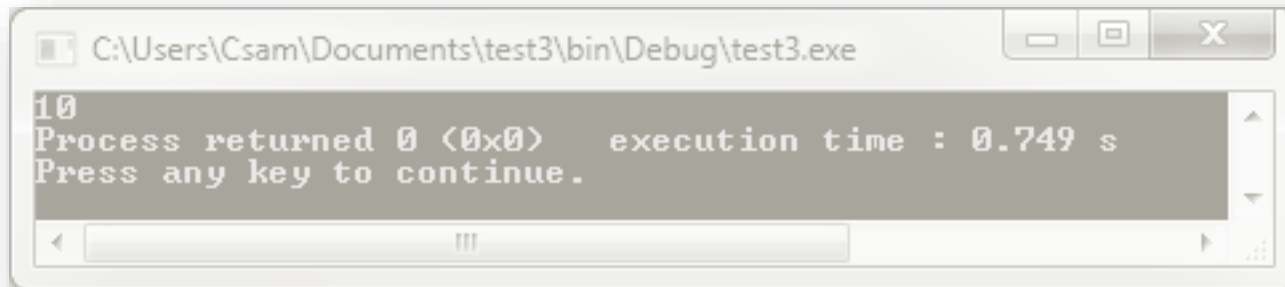
# Le pointeur

```
int age = 10;  
// "Je crée un pointeur"  
int *pointeurSurAge;  
// pointeurSurAge contient l'adresse de la variable age  
pointeurSurAge = &age;  
  
printf("%d", pointeurSurAge);
```



# Le pointeur

```
int age = 10;  
// "Je crée un pointeur"  
int *pointeurSurAge;  
// pointeurSurAge contient l'adresse de la variable age  
pointeurSurAge = &age;  
  
printf("%d", *pointeurSurAge);
```



# Le pointeur

```
int age = 10;  
// "Je crée un pointeur"  
int *pointeurSurAge;  
// pointeurSurAge contient l'adresse de la variable age  
pointeurSurAge = &age;  
  
*pointeurSurAge+=5;  
  
printf("%d", age);
```





# Le pointeur

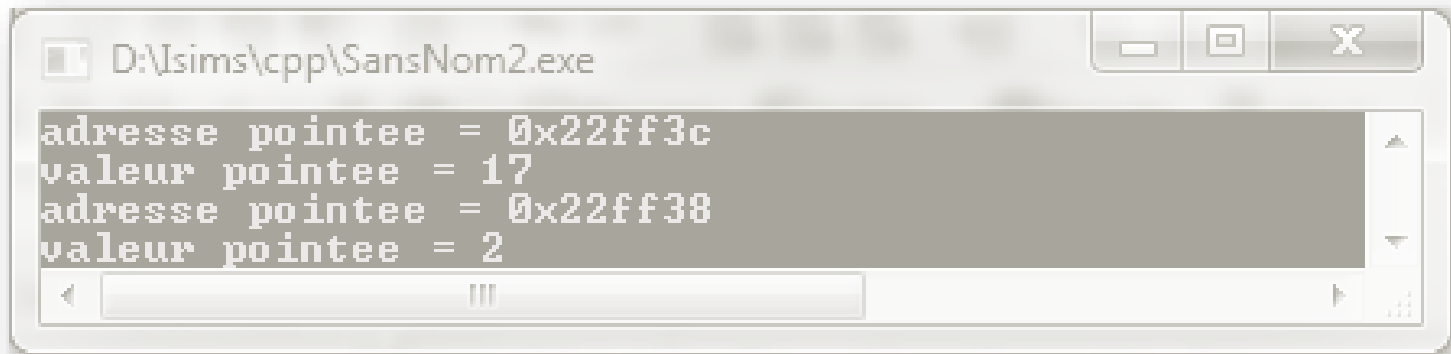
```
int a=17,b=2;  
int *pointeur=&a;
```

```
cout<<"adresse pointee = "<<pointeur<<endl;  
cout<<"valeur pointee = "<<*pointeur<<endl;
```

*/\*on peut décider de changer l'adresser stockée par le pointeur pour qu'il pointe ailleurs (pas possible avec une référence) \*/*

```
pointeur=&b;
```

```
cout<<"adresse pointee = "<<pointeur<<endl;  
cout<<"valeur pointee = "<<*pointeur<<endl;
```



```
D:\sims\cpp\SansNom2.exe  
adresse pointee = 0x22ff3c  
valeur pointee = 17  
adresse pointee = 0x22ff38  
valeur pointee = 2
```

# Passage par valeur vs variable

```
void valeurfois2(int valeur)           //passage par valeur
{
    valeur*=2;
    cout<<"valeurfois2 : valeur = "<<valeur<<endl;
}

void variablefois2(int *variable)      //passage par variable
{
    *variable*=2;
    cout<<"variablefois2 : variable = "<<*variable<<endl;
}

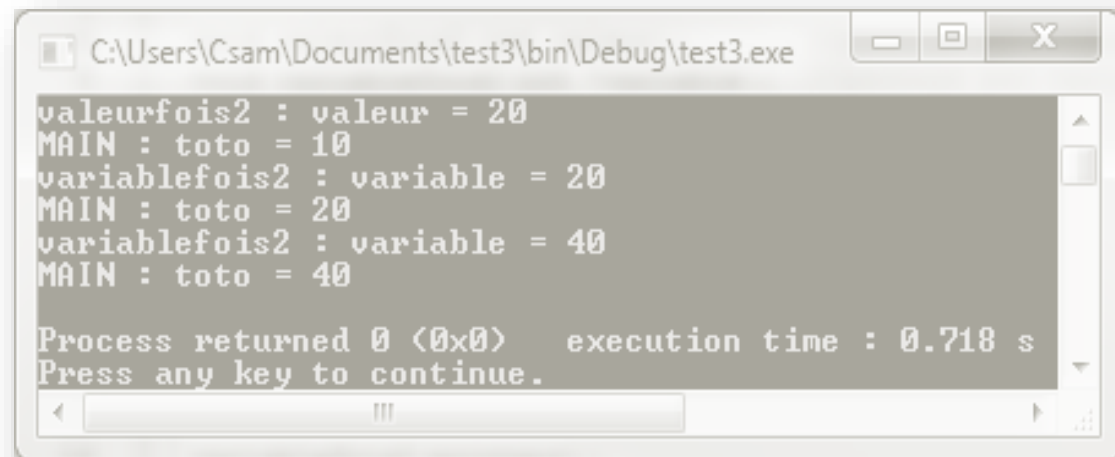
int main()
{
    int toto = 10;
    int *pointeur=&toto;

    valeurfois2(toto);
    cout<<"MAIN : toto = "<<toto<<endl;

    variablefois2(pointeur);
    cout<<"MAIN : toto = "<<toto<<endl;

    variablefois2(&toto);
    cout<<"MAIN : toto = "<<toto<<endl;

    return 0;
}
```



```
C:\Users\Csam\Documents\test3\bin\Debug\test3.exe
valeurfois2 : valeur = 20
MAIN : toto = 10
variablefois2 : variable = 20
MAIN : toto = 20
variablefois2 : variable = 40
MAIN : toto = 40

Process returned 0 (0x0)   execution time : 0.718 s
Press any key to continue.
```

# Le pointeur : NE JAMAIS OUBLIER 1

## Sur une variable :

**age** : valeur de la variable age

**&age** : l'adresse où se trouve la variable age

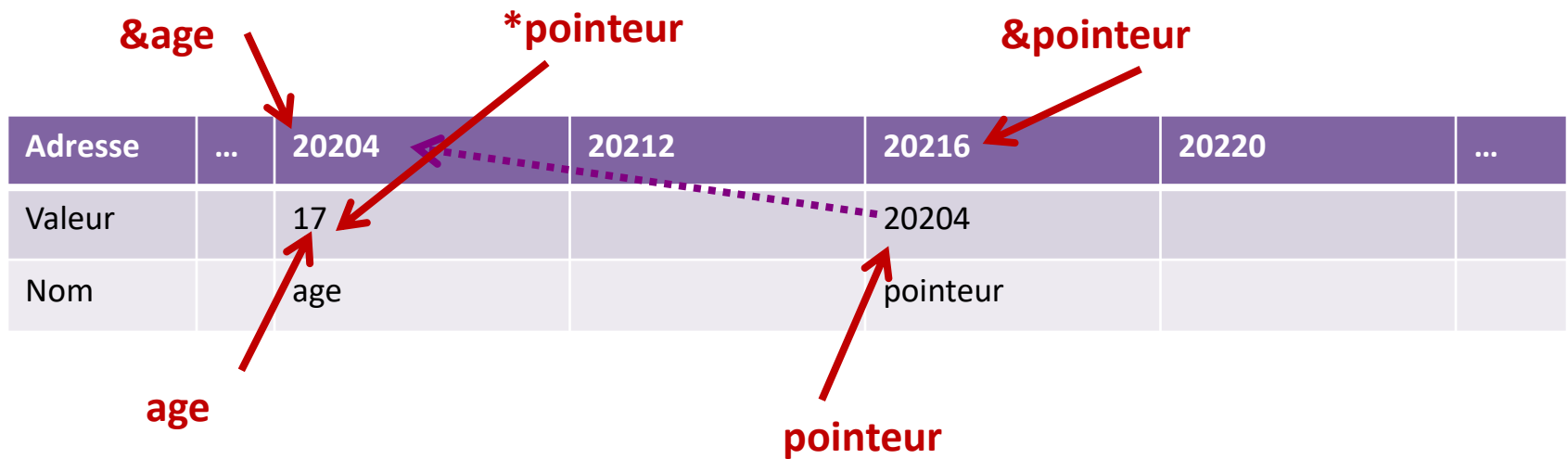
## Sur un pointeur :

**pointeur** : la valeur de pointeurSurAge  
(cette valeur étant une adresse).

**\*pointeur** : la valeur de la variable qui se trouve à l'adresse  
contenue dans pointeurSurAge

# Le pointeur : NE JAMAIS OUBLIER 2

```
int age=17;  
int *pointeur=&age;
```



# Le pointeur : utilité

## **Utilité N°1**

Ne pas inciter les gens à  
utiliser le C/C++

...

## **Utilité N°2**

Permet d'éviter de  
dupliquer les données  
entre les divers appels de  
fonction

- Économie de mémoire
- Meilleures performances

## **Utilité N°3**

**C'est le point suivant**

# Allocation dynamique

Il y a 2 façons de créer une variable (allouer de la mémoire) :

**Automatiquement (= statiquement) :**

```
int toto;
```

C'est la méthode que vous connaissez et que vous utilisez.

**Manuellement (= dynamiquement) :**

La mémoire est allouée pendant l'exécution du programme

Les variables sont créées à la volée

# Allocation dynamique

Il faut inclure la bibliothèque `<stdlib.h>`

- **malloc** : demande au système d'exploitation la permission d'utiliser de la mémoire.
- **free** : la place en mémoire est libérée,

Les 3 étapes de l'allocation dynamique :

- **malloc** pour demander de la mémoire
- **Vérifier la valeur retournée par malloc** pour savoir si l'OS a bien réussi
- Une fois qu'on a fini d'utiliser la mémoire, on doit la libérer avec **free**.

# Allocation dynamique

Nombre d'OCTET à allouer



```
int main()
{
    int* memoireAllouee = NULL;

    memoireAllouee = (int*) malloc( sizeof(int) );

    if (memoireAllouee == NULL) // On vérifie si la mémoire a été allouée
    {
        cout<<"ERREUR : malloc"<<endl;
        exit(0); // Erreur : on arrête tout !
    }
    // On peut utiliser ici la mémoire
    cout<<"Encodez un nombre : "<<endl;
    cin>>*memoireAllouee;
    cout<<"Le nombre : "<<*memoireAllouee<<endl;

    free(memoireAllouee);
    // On n'a plus besoin de la mémoire, on la libère

    return 0;
}
```



# L'allocation dynamique : utilité

## **Utilité N°1**

Justifier l'utilité des  
pointeurs

...

## **Utilité N°2**

Avec un programme il n'est pas  
toujours possible de prévoir le  
nombre de valeurs qu'il devra  
traiter.

Exemple :

Quand vous tapez du texte  
dans Word et dès qu'une 2<sup>e</sup>  
page se crée c'est une  
allocation dynamique qui se  
produit.

# Allocation dynamique d'un tableau

```
int nombre=0;

cout<<"combien ? ";

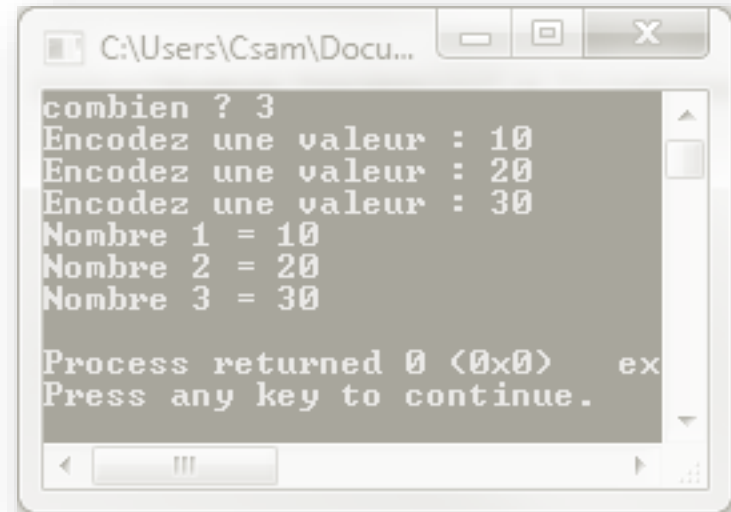
cin>>nombre;
int *tableauDYN=NULL;

tableauDYN=(int*)malloc(nombre*sizeof(int));

for(int i=0;i<nombre;i++)
{
    cout<<"Encodez une valeur : ";
    cin>>tableauDYN[i];
}

for(int i=0;i<nombre;i++)
    cout<<"Nombre "<<(i+1)<<" = "<<tableauDYN[i]<<endl;

free(tableauDYN);
```



# Allocation dynamique en C++

```
int *pointeur=NULL;
```

```
pointeur = new int;           // Allocation (le malloc du C)
```

```
delete pointeur;             //On libère la case mémoire (le free du C)
```

```
//Tableau Dynamique :
```

```
int taille=99;
```

```
int *TabDyn=NULL;
```

**Nombre d'ELEMENT à allouer**



```
TabDyn = new int[taille];
```

```
delete[] TabDyn;
```

# pointeurs multiples et allocation dynamique

// crée dynamiquement un tableau à 2D en C++

```
int **ary;    //double pointeur
```

```
ary = new int*[sizeY]; //allocation d'un vecteur de pointeurs
```

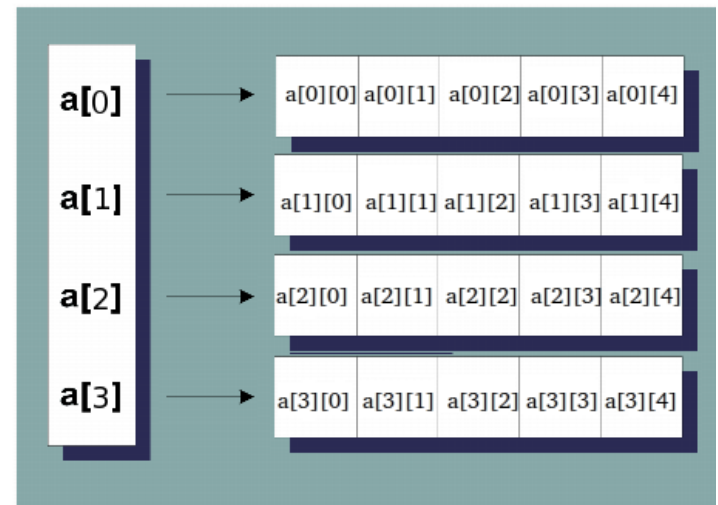
//allocation des cases pour le stockage des données

```
for(int i = 0; i < sizeY; ++i) {  
    ary[i] = new int[sizeX];  
}
```

...

//désallocation

```
for(int i = 0; i < sizeY; ++i) {  
    delete [] ary[i];  
}  
delete [] ary;
```



# Allocation dynamique

Ne pas mélanger **malloc** ( C ) et **delete** ( C++ ) avec un même pointeur

Ne pas mélanger **new**( C++ ) et **free** ( C ) avec un même pointeur

**Toujours libérer la mémoire**

sinon **perte de mémoire**

**Bien initialiser les pointeurs à NULL**

sinon « **pointeurs fous** »

# Erreur de segmentation

## Segmentation Fault

est un plantage d'une application qui a tenté d'accéder à un emplacement mémoire qui ne lui était pas alloué

**Avant d'appeler à l'aide,  
tu vérifies tes pointeurs !**



# Les structures

Les types complexes peuvent se construire à l'aide de structures

```
struct nom_structure
{
    type champs_1;
    type champs_2;
    ...
    Type champs_n
};
```

Exemple :

```
struct Client
{
    int Age;
    int Taille;
};
```

## Utilité des structures ?

- Structurer vos données



- Permet d'utiliser **un seul pointeur** pour envoyer une multitude de données

# Les structures

```
struct Client
{
    int Age;
    int Taille;
};
```

*Facultatif en C++*

```
struct Client roger;
struct Client nadine = { 69 , 154 };
struct Client clients[2];
```

```
roger.Age=45;
```

```
nadine.Taille=145;
```

```
clients[0].Age=17;
```



# Exercice 1 : multiplication matricielle

**Veillez réaliser un programme qui :**

1. demande à l'utilisateur une valeur entière  $N$
2. Alloue dynamiquement 3 matrices (A, B et C) d'entiers et de taille  $N \times N$
3. Remplit les matrices A et B de valeurs (aléatoires ou non)
4. Effectue la multiplication matricielle  $C=A*B$
5. Affiche la durée qui a été nécessaire pour le calcul (et exclusivement pour le calcul)
6. N'oubliez pas de désallouer

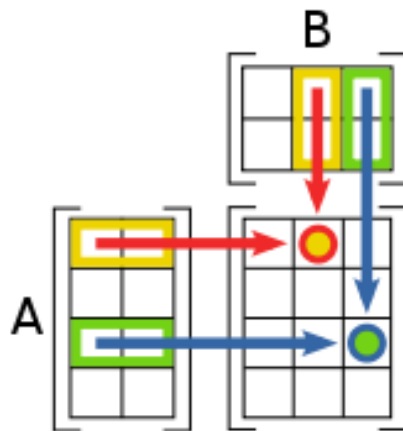
**Conservez bien votre programme, car vous allez en avoir besoin plus tard (dossier à réaliser)**

**Si ce programme n'est pas fonctionnel  
et/ou  
vous n'arrivez pas à réaliser ce programme par vous-même**

**La réalisation des manip  
suivantes vous sera  
impossible**



# Exercice 1 : multiplication matricielle



$$c_{12} = \sum_{r=1}^2 a_{1r} b_{r2} = a_{11} b_{12} + a_{12} b_{22}$$

$$c_{33} = \sum_{r=1}^2 a_{3r} b_{r3} = a_{31} b_{13} + a_{32} b_{23}$$

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} \times \begin{pmatrix} 7 & 8 \\ 9 & -1 \\ -2 & -3 \end{pmatrix} = \begin{pmatrix} 1 \times 7 + 2 \times 9 - 3 \times 2 & 1 \times 8 - 2 \times 1 - 3 \times 3 \\ 4 \times 7 + 5 \times 9 - 2 \times 6 & 4 \times 8 - 5 \times 1 - 6 \times 3 \end{pmatrix}$$

$$= \begin{pmatrix} 19 & -3 \\ 61 & 9 \end{pmatrix}$$

# Mesurer le temps

```
#include <chrono>
```

```
...
```

```
std::chrono::time_point<std::chrono::system_clock> start, end;
```

```
...
```

```
start = std::chrono::system_clock::now();
```

Job job job

```
end = std::chrono::system_clock::now();
```

```
long long int microseconds = chrono::duration_cast<chrono::microseconds>(end - start).count();
```

Multithreading

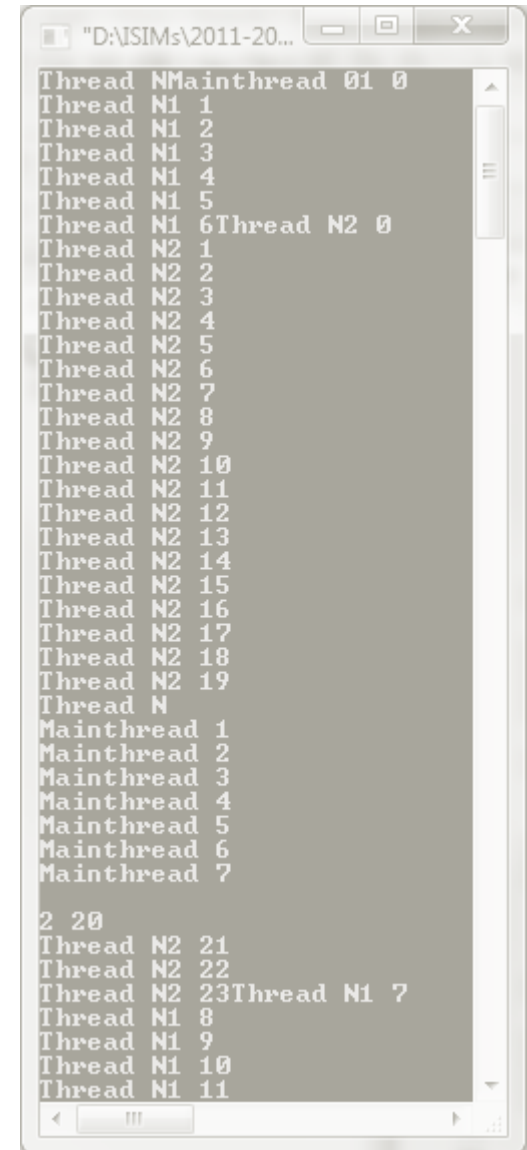
# **MULTITHREADING EN C++**

# Premier programme multithread

```

1  #include <iostream>
2  #include <thread>
3
4  using namespace std;
5
6  void fonction(int id)
7  {
8      for (int i = 0; i < 50; i++)
9      {
10         cout << "Thread N" << id << " " << i << endl;
11     }
12 }
13
14 int main()
15 {
16     thread t1(fonction, 1);
17     thread t2(fonction, 2);
18
19     for (int i = 0; i < 50; i++)
20     {
21         cout << "Mainthread " << i << endl;
22     }
23
24     return 0;
25 }

```



```

Thread NMainthread 01 0
Thread N1 1
Thread N1 2
Thread N1 3
Thread N1 4
Thread N1 5
Thread N1 6Thread N2 0
Thread N2 1
Thread N2 2
Thread N2 3
Thread N2 4
Thread N2 5
Thread N2 6
Thread N2 7
Thread N2 8
Thread N2 9
Thread N2 10
Thread N2 11
Thread N2 12
Thread N2 13
Thread N2 14
Thread N2 15
Thread N2 16
Thread N2 17
Thread N2 18
Thread N2 19
Thread N
Mainthread 1
Mainthread 2
Mainthread 3
Mainthread 4
Mainthread 5
Mainthread 6
Mainthread 7

2 20
Thread N2 21
Thread N2 22
Thread N2 23Thread N1 7
Thread N1 8
Thread N1 9
Thread N1 10
Thread N1 11

```

# Jthread (1)



```
1  #include <iostream>
2  #include <thread>
3
4  void fct()
5  {
6      int x=0;
7      while (1)
8      {
9          x++;
10         std::cout << x << std::endl;
11     }
12 }
13
14
15 int main()
16 {
17     std::thread t(fct);
18
19     std::cout << "attente du thread" << std::endl;
20     t.join();
21     std::cout << "fin du programme" << std::endl;
22     return 0;
23 }
```

← Ne termine jamais

# Jthread (2)



```
1  #include <iostream>
2  #include <thread>
3
4  void fct()
5  {
6      int x=0;
7      while (1)
8      {
9          x++;
10         std::cout << x << std::endl;
11     }
12 }
13
14
15 int main()
16 {
17     std::jthread t(fct); //<---- ici
18
19     std::cout << "attente du thread" << std::endl;
20     t.request_join();    //<---- et là
21     t.join();
22     std::cout << "fin du programme" << std::endl;
23     return 0;
24 }
```

Ne termine toujours pas

# Jthread (3)



```
1  #include <iostream>
2  #include <thread>
3
4  void fct(std::stop_token st)
5  {
6      int x=0;
7      while(!st.stop_requested())
8      {
9          x++;
10         std::cout << x << std::endl;
11     }
12 }
13
14 int main()
15 {
16     std::jthread t(fct);
17
18     std::cout << "attente du thread" << std::endl;
19     t.requested_join();
20     t.join();
21     std::cout << "fin du programme" << std::endl;
22     return 0;
23 }
```

Ok le thread se coupe  
et libère automatiquement  
ses ressources

... les ressources statiques !

...

*Mécanisme risqué...  
autant utiliser des  
threads conventionnels*





Multithreading avec C++

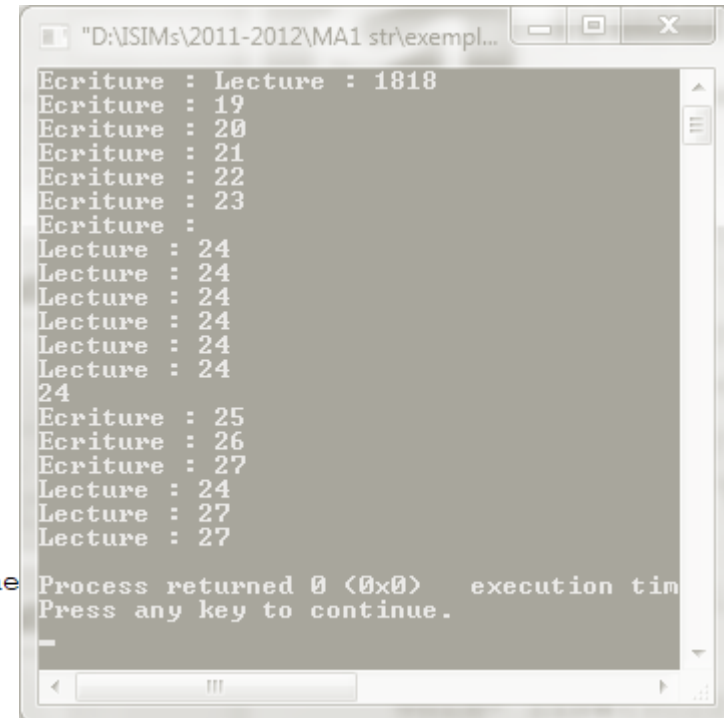
# MUTEX

# Partage de ressources

```

1  #include <iostream>
2  #include <thread>
3
4  using namespace std;
5
6  void ecrire(int *donnee)
7  {
8      for (int i = 0; i<10; i++)
9      {
10         *donnee += 1;
11         cout << "Ecriture : " << *donnee << endl;
12     }
13 }
14
15 void lire(int *donnee)
16 {
17     for (int i = 0; i<10; i++) cout << "Lecture : " << *donnee << endl;
18 }
19
20
21 int main()
22 {
23     int valeur = 17;
24     thread t_ecriture(ecrire, &valeur);
25     thread t_lecture(lire, &valeur);
26
27     t_ecriture.join();
28     t_lecture.join();
29
30     return 0;
31 }

```



```

Ecriture : Lecture : 1818
Ecriture : 19
Ecriture : 20
Ecriture : 21
Ecriture : 22
Ecriture : 23
Ecriture :
Lecture : 24
Lecture : 24
Lecture : 24
Lecture : 24
Lecture : 24
24
Ecriture : 25
Ecriture : 26
Ecriture : 27
Lecture : 24
Lecture : 27
Lecture : 27

Process returned 0 (0x0)   execution time
Press any key to continue.

```

# Partage des ressources et exclusion mutuelle

Dans l'illustration suivante, le résultat n'est pas prévisible :

Thread A	Thread B
1A: Lire variable V	1B: Lire variable V
2A: Add 1 à la variable V	2B: Add 1 à la variable V
3A: Écrire la variable V	3B: Écrire la variable V

Lire, Lire, Add, Add, Écrire, Écrire                      -> V+1

Lire, Add, Écrire, Lire, Add, Écrire                      -> V+2

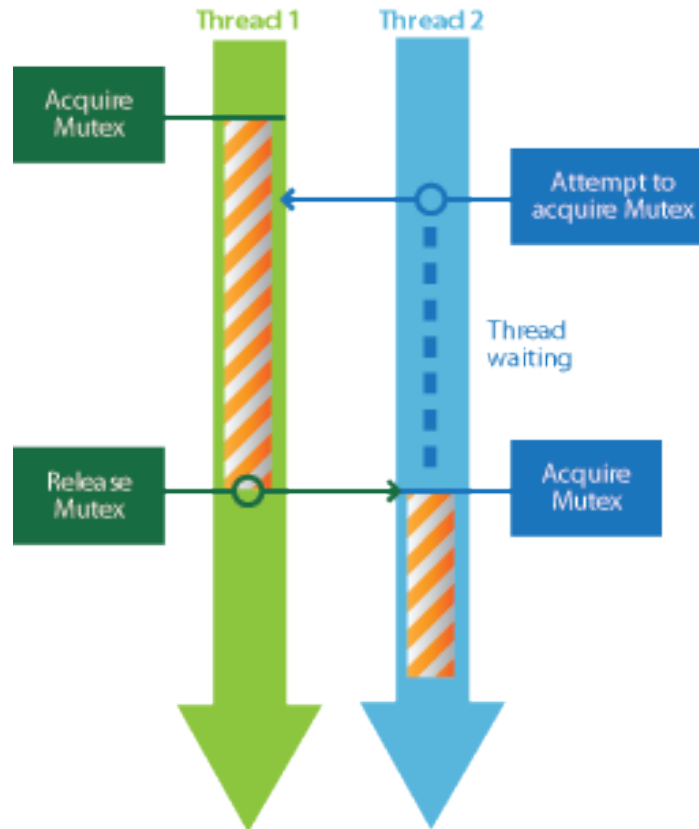
Ce phénomène est connu sous le nom de **situation de compétition**

**Solution :**

Thread A	Thread B
1A: Verrouiller la variable V	1B: Verrouiller la variable V
2A: Lire la variable V	2B: Lire la variable V
3A: Add 1 à la variable V	3B: Add 1 à la variable V
4A: Écrire la variable V	4B: Écrire la variable V
5A: déverrouiller la variable V	5B: déverrouiller la variable V

# Le Mutex (1)

Pour se prémunir contre ce problème, on doit délimiter **une section critique**

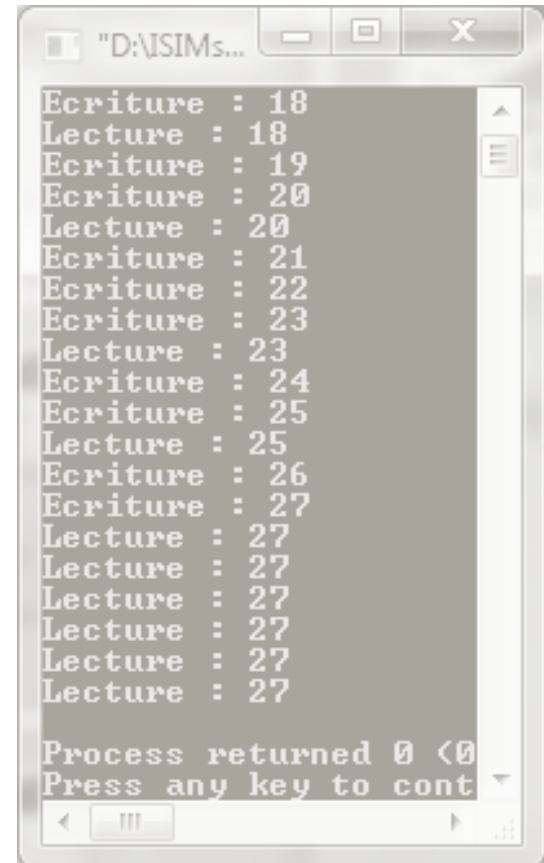


# Le Mutex (2)

```

1  #include <iostream>
2  #include <thread>
3  #include <mutex>
4
5  std::mutex mtx;
6
7  void ecrire(int *donnee)
8  {
9      for (int i = 0; i<10; i++)
10     {
11         mtx.lock();
12         *donnee += 1;
13         std::cout << "Ecriture : " << *donnee << std::endl;
14         mtx.unlock();
15     }
16 }
17
18 void lire(int *donnee)
19 {
20     for (int i = 0; i < 10; i++)
21     {
22         mtx.lock();
23         std::cout << "Lecture : " << *donnee << std::endl;
24         mtx.unlock();
25     }
26 }
27
28 int main()
29 {
30     int valeur = 17;
31     std::thread t_ecriture(ecrire, &valeur);
32     std::thread t_lecture(lire, &valeur);
33
34     t_ecriture.join();
35     t_lecture.join();
36
37     return 0;
38 }

```



```

Ecriture : 18
Lecture : 18
Ecriture : 19
Ecriture : 20
Lecture : 20
Ecriture : 21
Ecriture : 22
Ecriture : 23
Lecture : 23
Ecriture : 24
Ecriture : 25
Lecture : 25
Ecriture : 26
Ecriture : 27
Lecture : 27
Lecture : 27
Lecture : 27
Lecture : 27
Lecture : 27

Process returned 0 (0)
Press any key to cont

```



## Interblocage



Une situation de blocage (2 tâches s'attendent mutuellement) est appelé **étreinte fatale** ou **interblocage (deadlock)**.

TâcheA :

Obtenir M1

Obtenir M2

Action nécessitant les deux verrous

Rendre M2

Rendre M1

TâcheB :

Obtenir M2

Obtenir M1

Action nécessitant les deux verrous

Rendre M1

Rendre M2



**Solution préventive :**

acquérir les mutex dans le même ordre.



# Départ volontaire



```

5  std::mutex mtx;
6
7  void ecrire(int *donnee)
8  {
9      for (int i = 0; i<10; i++)
10     {
11         mtx.lock();
12         *donnee += 1;
13         std::cout << "Ecriture : " << *donnee << std::endl;
14         return;
15         mtx.unlock();
16     }
17 }
18
19 void lire(int *donnee)
20 {
21     for (int i = 0; i < 10; i++)
22     {
23         mtx.lock();
24         std::cout << "Lecture : " << *donnee << std::endl;
25         mtx.unlock();
26     }
27 }
28
29 int main()
30 {
31     int valeur = 17;
32     std::thread t_ecriture(ecrire, &valeur);
33     std::thread t_lecture(lire, &valeur);
34
35     t_ecriture.join();
36     t_lecture.join();
37
38     return 0;
39 }

```

**Situation de blocage !!!**

# Mutex et section critique

```

7 void ecrire(int *donnee)
8 {
9     for (int i = 0; i<10; i++)
10    {
11        mtx.lock();
12        *donnee += 1;
13        std::cout << "Ecriture : " << *donnee << std::endl;
14        mtx.unlock();
15    }
16 }

```

*Section critique*

**Il ne faut pas confondre l'utilisation d'un mutex avec une section critique**

Un mutex est un mécanisme logiciel qui permet d'implémenter une section critique mais il existe d'autres façons de faire

Attention à ne pas abuser des sections critiques car cela risque de "séquentialiser" l'exécution



# Mutex : syntaxe alternative

```

7 void ecrire(int *donnee)
8 {
9     for (int i = 0; i<10; i++)
10    {
11
12        std::lock_guard<std::mutex> lock(mtx);
13        *donnee += 1;
14        std::cout << "Ecriture : " << *donnee << std::endl;
15    }
16 }
17

```

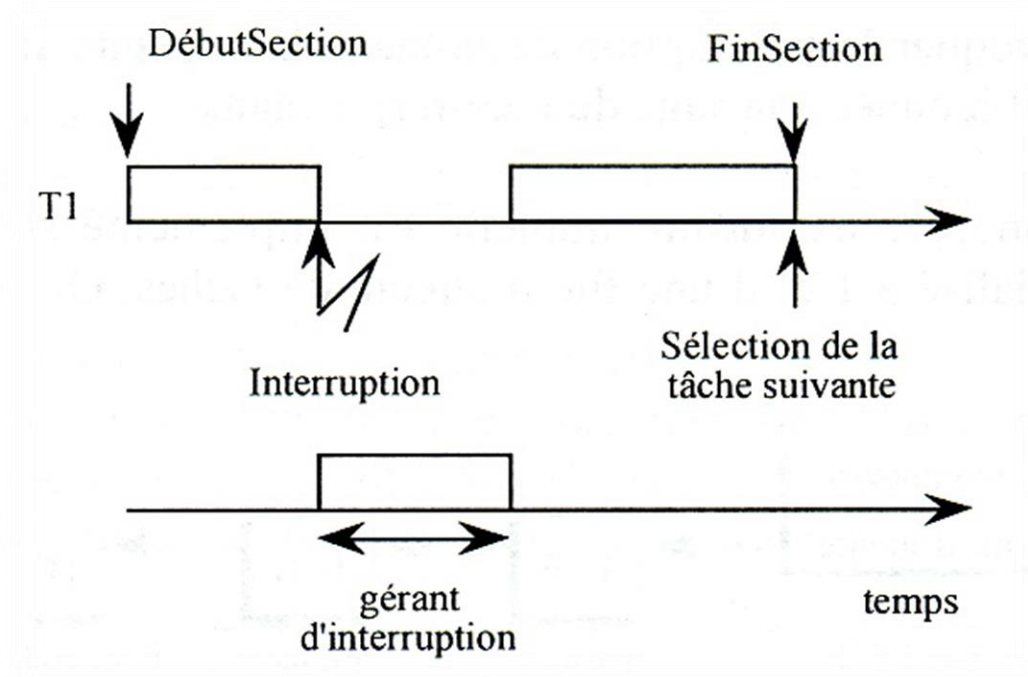
*Section critique*

Libère automatiquement le mutex une fois arrivé ici



# Masquage logiciel

Avec un système temps réel, le masquage logiciel ne permet d'assurer l'atomicité que vis-à-vis des tâches.



*Masquage logiciel*

Le masquage matériel permet d'assurer l'atomicité d'une séquence vis-à-vis des tâches et des interruptions.

# Performances des mutex

Un mutex est une primitive logicielle

Son exécution peut varier d'une cible à l'autre  
(matériel / OS / compilateur)

**Dans le meilleur des cas**, le thread en attente d'un mutex reste actif sur le CPU jusqu'à ce qu'il puisse poursuivre (rare)

**Dans le pire des cas** (le plus courant), le thread passe en veille et est déchargé du CPU. Il devra donc être rechargé complètement quand le mutex sera libéré.

De plus, la libération d'un mutex n'est pas forcément synonyme de réveil des threads en attente du mutex.

Ce point peut être important en fonction du contexte et **du grain de parallélisme**

# Acquisition non bloquante du mutex

Variante avec un "timed\_mutex"

```
std::timed_mutex mutex;
```

```
bool ok=false;

while(!ok)
{
    if(le_mutex.try_lock())
    {
        //ok j'ai le mutex
        ok=true;
        le_mutex.unlock();
    }
    else
    {
        //je m'occupe
    }
}
```

```
bool ok=false;

while(!ok)
{
    if(mutex.try_lock_for(chrono::milliseconds(100)))
    {
        //ok j'ai le mutex
        ok=true;
        mutex.unlock();
    }
    else
    {
        //je m'occupe
    }
}
```

Il existe aussi *try\_lock\_until( l'heure )*

...

# Le mutex partagé



`std::shared_mutex pastouche;`

Peut être verrouillé comme un mutex classique

ou

verrouillé en mode partagé.

Le mode partagé n'est pas un verrou exclusif

Les accès (partagés) peuvent être concurrents

Pendant qu'un mutex partagé est verrouillé de façon exclusive  
il ne pourra pas être verrouillé de façon partagée

et vice versa

Existe en version "timed"

```
//Lecteur 1
```

```
pastouche.lock_shared();
```

Lecture

```
pastouche.unlock_shared();
```

```
//Lecteur 2
```

```
pastouche.lock_shared();
```

Lecture

```
pastouche.unlock_shared();
```

```
//Lecteur 3
```

```
pastouche.lock_shared();
```

Lecture

```
pastouche.unlock_shared();
```

```
//Écrivain
```

```
pastouche.lock();
```

Écriture

```
pastouche.unlock();
```

## Exercice 2 : multiplication matricielle

**Veillez réaliser un programme qui :**

1. Fait la même chose que l'exercice 1
2. Le calcul est cette fois réalisé par des threads
3. Instanciation d'un thread par case de la matrice C

**Conservez bien votre programme, car vous allez en avoir besoin plus tard (dossier à réaliser)**

Multithreading en C++

# **SYNCHRONISATIONS**

# Sémaphore à compte (1)

Le mutex partagé ne permet pas de limiter le nombre d'acquisitions

Le sémaphore oui car il est composé d'un compteur et d'une file de tâches en attente.

le compteur est initialisé avec le nombre d'éléments de ressources initialement disponible.

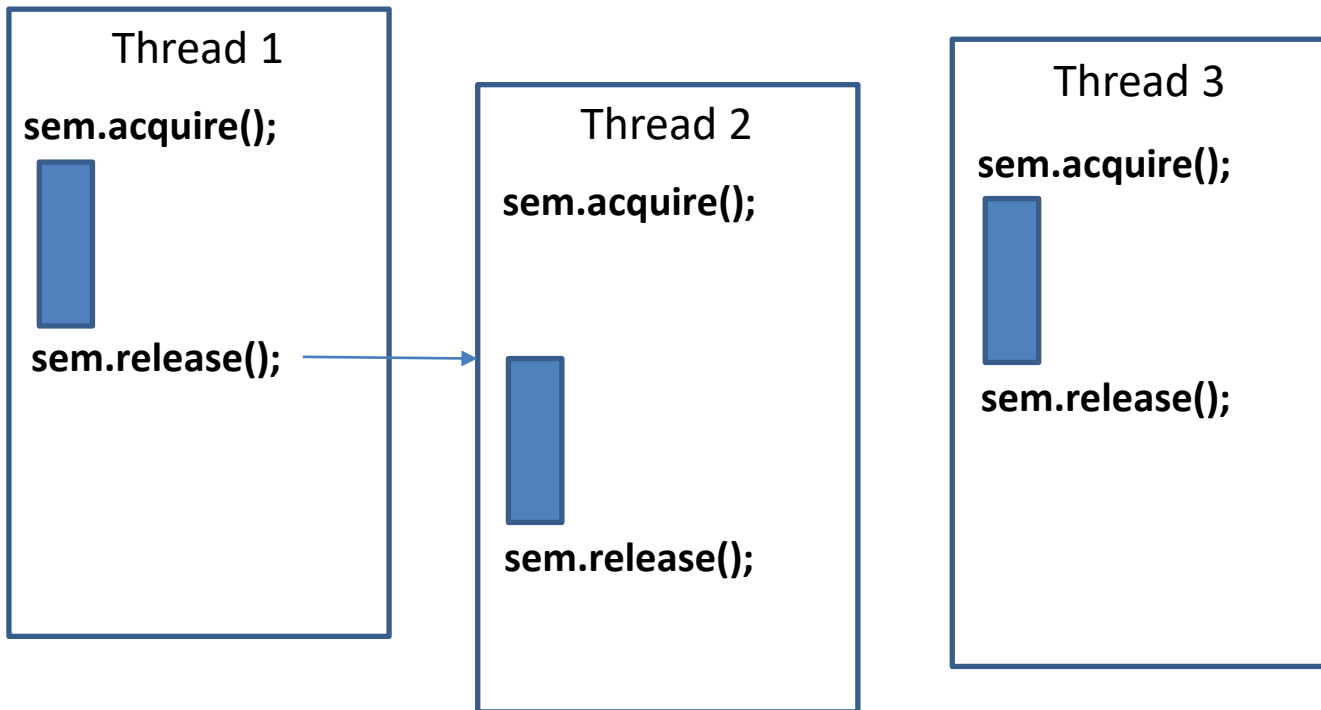
La prise d'un sémaphore dont le compteur est négatif ou nul bloque l'appelant



# Sémaphore à compte (2)



`std::counting_semaphore<2> sem(2);`



# Sémaphore à compte (3)



```
1  #include <iostream>
2  #include <thread>
3  #include <semaphore>
4
5  #define NOMBRE 10
6
7  int tableau[NOMBRE];
8  std::counting_semaphore<NOMBRE> semEcrire(NOMBRE);
9  std::counting_semaphore<NOMBRE> semLire(0);
10
12 void fprod()
13 {
14     int cpt = 0, donnee=0;
15     for (int i = 0; i < NOMBRE * 2; i++)
16     {
17         donnee = rand()%1024;
18         printf("Ecriture[%d]=%d\n", i, donnee);
19
20         semEcrire.acquire();
21         tableau[cpt] = donnee;
22         semLire.release();
23
24         cpt++;
25         if (cpt >= NOMBRE) cpt = 0;
26     }
27 }
```

```
44 int main()
45 {
46     std::thread tprod(fprod);
47     std::thread tcons(fcons);
48
49     tprod.join();
50     tcons.join();
51
52     return 0;
53 }
```

```
29 void fcons()
30 {
31     int cpt = 0, donnee = 0;
32     for (int i = 0; i < NOMBRE * 2; i++)
33     {
34         semLire.acquire();
35         donnee= tableau[cpt];
36         semEcrire.release();
37
38         printf("\tLecture[%d]=%d\n", i, donnee);
39         cpt++;
40         if (cpt >= NOMBRE) cpt = 0;
41     }
42 }
```

```
Ecriture[0]=3736
Ecriture[1]=1020
    Lecture[0]=3736
    Lecture[1]=1020
Ecriture[2]=9713
    Lecture[2]=9713
Ecriture[3]=8494
    Lecture[3]=8494
Ecriture[4]=9066
    Lecture[4]=9066
Ecriture[5]=8903
    Lecture[5]=8903
Ecriture[6]=2985
    Lecture[6]=2985
Ecriture[7]=7693
    Lecture[7]=7693
Ecriture[8]=8650
Ecriture[9]=8927
    Lecture[8]=8650
    Lecture[9]=8927
Ecriture[10]=9811
    Lecture[10]=9811
Ecriture[11]=2889
Ecriture[12]=5114
    Lecture[11]=2889
    Lecture[12]=5114
```

# Sémaphore binaire



Un sémaphore binaire peut être utilisé pour protéger une section critique comme avec un mutex

Mais ... ce n'est pas son utilité !

	Sémaphore binaire	Mutex
Utilité	Mécanisme de synchronisation	Protéger une section critique
Qui peut release/unlock ?	N'importe quel thread	Uniquement le thread qui a acquis le mutex

# Synchronisation

**En multi-tâches, l'ordre d'exécution des instructions de tâches différentes est indéterminé.**

C'est pourquoi il existe un **mécanisme** permettant la **synchronisation**.

Par exemple:

- bloquer la tâche courante en attente d'un évènement
- ou la débloquent dès le déclenchement de cet évènement.

**La synchronisation consiste à imposer un ordre sur l'exécution des instructions des processus.**

# Condition

```

1  #include <iostream>
2  #include <string>
3  #include <thread>
4  #include <mutex>
5  #include <condition_variable>
6
7  std::mutex m;
8  std::condition_variable cv;
9  std::string data;
10
11 void worker_thread()
12 {
13     std::unique_lock<std::mutex> lk(m);
14     cv.wait(lk);
15
16     std::cout << "Thread : je boss" << std::endl;
17     data += " que je complete";
18
19     std::cout << "Thread a fini" << std::endl;
20
21     lk.unlock();
22     cv.notify_one();
23 }
24
25 int main()
26 {
27     std::thread worker(worker_thread);
28
29     data = "du blabla";
30
31     std::cout << "main: ok ready" << std::endl;
32
33     cv.notify_one();
34
35     std::unique_lock<std::mutex> lk(m);
36     cv.wait(lk);
37     lk.unlock();
38
39     std::cout << "Main : data = " << data << '\n';
40
41     worker.join();
42 }

```

***notify\_all();*** pour notifier tous les threads en attente

```

C:\SIMs\2021-2022\STR\Threads\Projet-condition\Debu
main: ok ready
Thread : je boss
Thread a fini
Main : data = du blabla que je complete

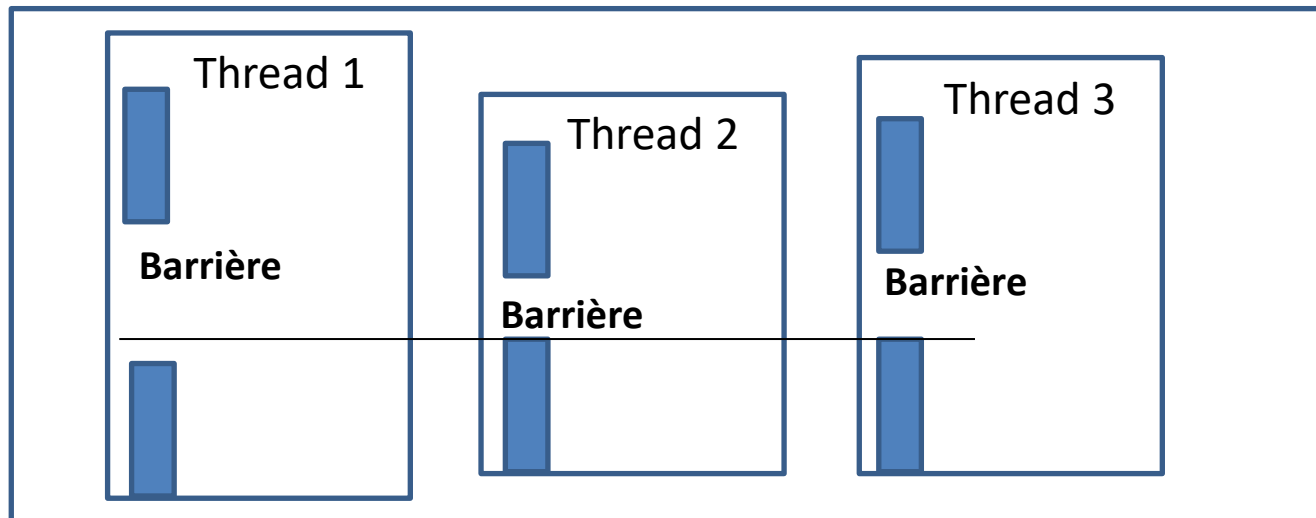
```

# Barrière de synchronisation

Chaque tâche qui arrivera sur la barrière devra attendre jusqu'à ce que le nombre spécifié de tâches soient également arrivées

En C++20 il y a les barrières :

- `std::latch` = à usage unique
- `std::barrier` = réutilisable



# std::latch (1)

```
1  #include <iostream>
2  #include <latch>
3  #include <thread>
4
5  std::latch jobdone(5);
6  std::latch dodo(1);
7
8  void fct(int id)
9  {
10     printf("Je boss moi le thread %d \n", id);
11     jobdone.count_down();
12
13     dodo.wait();
14     printf("A+ de thread %d \n", id);
15 }
16
17
18 int main() {
19
20     std::thread* ttab[5];
21     for (int i = 0; i < 5; i++) ttab[i] = new std::thread(fct, i);
22
23     jobdone.wait();
24
25     std::cout << "Enfin ils ont fini !" << std::endl;
26
27     dodo.count_down();
28
29     for (int i = 0; i < 5; i++) ttab[i]->join();
30
31     for (int i = 0; i < 5; i++) delete ttab[i];
32 }
```

```
Je boss moi le thread 0
Je boss moi le thread 1
Je boss moi le thread 2
Je boss moi le thread 3
Je boss moi le thread 4
Enfin ils ont fini !
A+ de thread 4
A+ de thread 0
A+ de thread 2
A+ de thread 1
A+ de thread 3
```

## std::latch (2)



```
1  #include <iostream>
2  #include <latch>
3  #include <thread>
4
5  std::latch jobdone(5);
6
7  void fct(int id)
8  {
9      printf("Je boss moi le thread %d \n", id);
10     jobdone.arrive_and_wait();
11     printf("A+ de thread %d \n", id);
12 }
13
14
15 int main() {
16
17     std::thread* ttab[5];
18     for (int i = 0; i < 5; i++) ttab[i] = new std::thread(fct, i);
19
20     for (int i = 0; i < 5; i++) ttab[i]->join();
21
22     for (int i = 0; i < 5; i++) delete ttab[i];
23 }
```

```
Je boss moi le thread 0
Je boss moi le thread 1
Je boss moi le thread 2
Je boss moi le thread 3
Je boss moi le thread 4
A+ de thread 4
A+ de thread 2
A+ de thread 3
A+ de thread 0
A+ de thread 1
```



## std::barrier (2)



```
1  #include <iostream>
2  #include <barrier>
3  #include <thread>
4
5  std::barrier jobdone(5);
6
7  void fct(int id)
8  {
9      for (int i = 0; i < 3; i++)
10     {
11         printf("Job %d ok pour le thread %d \n", i, id);
12         jobdone.arrive_and_wait();
13     }
14     printf("A+ de thread %d \n", id);
15 }
16
17
18 int main() {
19
20     std::thread* ttab[5];
21     for (int i = 0; i < 5; i++) ttab[i] = new std::thread(fct, i);
22
23     for (int i = 0; i < 5; i++) ttab[i]->join();
24
25     for (int i = 0; i < 5; i++) delete ttab[i];
26 }
```

```
Job 0 ok pour le thread 0
Job 0 ok pour le thread 1
Job 0 ok pour le thread 2
Job 0 ok pour le thread 3
Job 0 ok pour le thread 4
Job 1 ok pour le thread 4
Job 1 ok pour le thread 3
Job 1 ok pour le thread 1
Job 1 ok pour le thread 2
Job 1 ok pour le thread 0
Job 2 ok pour le thread 0
Job 2 ok pour le thread 2
Job 2 ok pour le thread 1
Job 2 ok pour le thread 4
Job 2 ok pour le thread 3
A+ de thread 3
A+ de thread 4
A+ de thread 1
A+ de thread 2
A+ de thread 0
```

Multithreading en C++

**ENCORE ++**

# Spinlock

Même principe qu'un mutex à l'exception que :  
les threads qui essayent d'acquérir le spinlock ne sont pas préemptés  
ils attendent la libération du verrou.

**Pas officiellement implémenté en C++**

Existe en POSIX

**Gains de performances élevés (+/- 2x)** pour de faibles grains de parallélisme

**Mécanisme risqué**

# #include <future>

Des primitives permettant de lire/écrire/attendre/etc. des ressources asynchrones (un peu à la façon d'AJAX).

Defined in header <future>

<b>promise</b> (C++11)	stores a value for asynchronous retrieval (class template)
<b>packaged_task</b> (C++11)	packages a function to store its return value for asynchronous retrieval (class template)
<b>future</b> (C++11)	waits for a value that is set asynchronously (class template)
<b>shared_future</b> (C++11)	waits for a value (possibly referenced by other futures) that is set asynchronously (class template)
<b>async</b> (C++11)	runs a function asynchronously (potentially in a new thread) and returns a <code>std::future</code> that will hold the result (function template)
<b>launch</b> (C++11)	specifies the launch policy for <code>std::async</code> (enum)
<b>future_status</b> (C++11)	specifies the results of timed waits performed on <code>std::future</code> and <code>std::shared_future</code> (enum)

## Future errors

<b>future_error</b> (C++11)	reports an error related to futures or promises (class)
<b>future_category</b> (C++11)	identifies the future error category (function)
<b>future_errc</b> (C++11)	identifies the future error codes (enum)

# Les opérations atomiques

**#include <atomic>**

Ensemble de primitives permettant de garantir des accès atomiques à une ressource unique (une variable basique)

Pas besoin de primitive de synchronisation

Bien plus rapide qu'un mutex

Cependant... il peut être nécessaire de définir l'ordonnancement des accès mémoires pour utiliser les opérations atomiques correctement

## Exercice 3 : multiplication matricielle

**Veillez réaliser un programme qui :**

1. Fait la même chose que l'exercice 1
2. Le calcul est cette fois réalisé par des threads
3. Instanciation d'un thread par ligne de la matrice C

**Conservez bien votre programme, car vous allez en avoir besoin plus tard (dossier à réaliser)**

## Exercice 4 : multiplication matricielle

**Veillez réaliser un programme qui :**

1. Fait la même chose que l'exercice 1
2. Le calcul est cette fois réalisé par des threads
3. Le nombre de threads à instancier est demandé à l'utilisateur

**Conservez bien votre programme, car vous allez en avoir besoin plus tard (dossier à réaliser)**