

# Installer et utiliser Google Benchmark avec Visual Studio 2019

---

Hélène Toussaint, novembre 2021

Le but de ce document est d'aider à la prise en main de Google Benchmark, une bibliothèque qui fournit des fonctionnalités pour faire du benchmark de code C++. Il existe actuellement deux manières d'utiliser cette bibliothèque :

- en l'installant sur son PC
- en passant par le site web <https://quick-bench.com/>

L'avantage du site web est qu'il n'y a rien à installer, les résultats sont affichés sous forme graphique, on peut choisir entre différents compilateurs et option d'optimisation. L'inconvénient du site web c'est que les tests tournent sur un serveur, et donc suivant la charge du serveur les temps de calcul peuvent varier.

Dans ce document on utilise la dernière release à date de Google Benchmark (v1.6.0), un PC Windows 10 avec Visual Studio 2019 (version 14.27.29110) et Python 3.7.9 préalablement installés.

## Table des matières

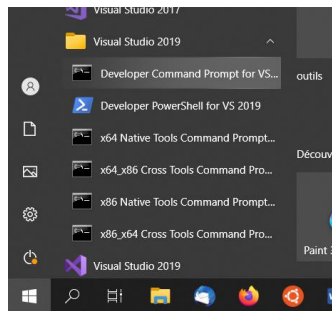
1	Télécharger et installer Google benchmark sous Windows .....	2
2	Utiliser Google Benchmark.....	3
2.1	Créer et paramétrer un projet VS 2019 pour utiliser Google Benchmark .....	3
2.2	Tester son installation sur un exemple .....	4
2.3	Utilisation basique de Google Benchmark .....	4
3	Google Benchmark vs évaluation à la main .....	6
3.1	Comparaison en utilisant Google Benchmark sous Windows / Visual Studio 2019 .....	7
3.2	Comparaison en utilisant Google Benchmark sous Linux / gcc 9.3.0.....	9
3.3	Utilisation du site web <a href="https://quick-bench.com">https://quick-bench.com</a> .....	9
3.4	Synthèse des différents tests .....	10
4	Références.....	11

# 1 Télécharger et installer Google benchmark sous Windows

Les instructions d'installation de Google Benchmark sont données sur son github [1]. Il s'agit simplement d'utiliser git et cmake en ligne de commande. Ceci est facilement réalisable sous Windows avec l'environnement de développement Visual Studio.

En effet, lorsque l'on installe Visual Studio pour développer en C++, tous les outils de programmation liés au C++ (compilateur, cmake, git, ...) sont automatiquement installés. Il est alors possible de les utiliser, soit via l'interface graphique de Visual Studio, soit en ligne de commande [2]. C'est cette dernière option qui nous intéresse ici.

Pour utiliser les outils du C++ en ligne de commande, il suffit d'ouvrir un des *Developer prompts* installés dans le répertoire Visual Studio (pour cela ouvrir le menu Démarrer, puis dans le répertoire « Visual Studio 2019 » cliquer sur le *Developer Command Prompt* souhaité). Il existe différents prompts qui correspondent à différentes architectures cibles (voir [3]). *Developer PowerShell for VS 2019* s'utilise comme un terminal linux, c'est celui que nous utiliserons dans la suite de ce document.



**NB.** Pour éviter les problèmes de droits, ouvrir le *Developer PowerShell* avec les droits d'administrateur : faire un clic droit et choisir « Exécuter avec les droits administrateur »

Créer un répertoire (par exemple "E:\Visual Studio 2019\GoogleBench") qui servira à stocker les fichiers téléchargés et se positionner dedans à partir du *Developer PowerShell* :

```
Administrateur : Developer PowerShell for VS 2019
*****
** Visual Studio 2019 Developer PowerShell v16.7.4
** Copyright (c) 2020 Microsoft Corporation
*****
PS C:\Users\helene\source\repos> cd "E:\Visual Studio 2019\GoogleBench"
PS E:\Visual Studio 2019\GoogleBench>
```

Il ne reste qu'à suivre les instructions données dans la section installation de [1]. C'est-à-dire taper les commandes suivantes dans notre terminal *Developer PowerShell* :

```
git clone https://github.com/google/benchmark.git
```

```
cd benchmark
```

```
cmake -E make_directory "build"
```

```
cmake -E chdir "build" cmake -DBENCHMARK_DOWNLOAD_DEPENDENCIES=on -
DCMAKE_BUILD_TYPE=Release ../
```

```
cmake --build "build" --config Release
```

```
cmake -E chdir "build" ctest --build-config Release
```

```
cmake --build "build" --config Release --target install
```

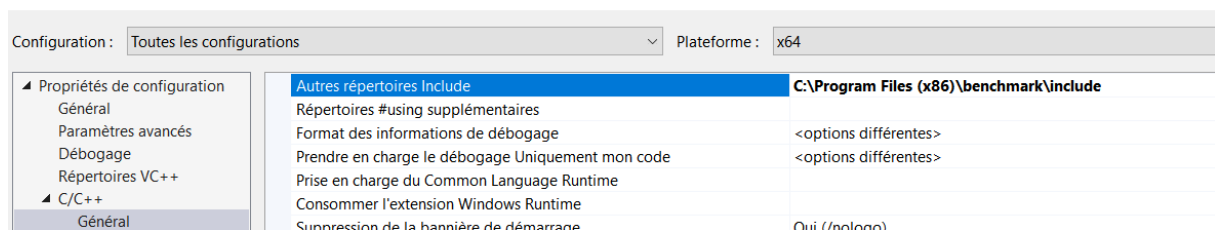
L'avant-dernière commande permet de vérifier que l'installation est réussie en lançant quelques tests de manière automatique.

La dernière commande permet d'installer Google Benchmark de manière globale : c'est-à-dire que vous verrez apparaître un répertoire `benchmark` dans `C:/Program Files (x86)/`

## 2 Utiliser Google Benchmark

### 2.1 Créer et paramétrer un projet VS 2019 pour utiliser Google Benchmark

- Créer un projet Visual Studio vide
- Dans les propriétés du projet, onglet C/C++ > Général > Autres répertoires Include : ajouter le chemin vers les fichiers include de Google Benchmark `C:\Program Files (x86)\benchmark\include`

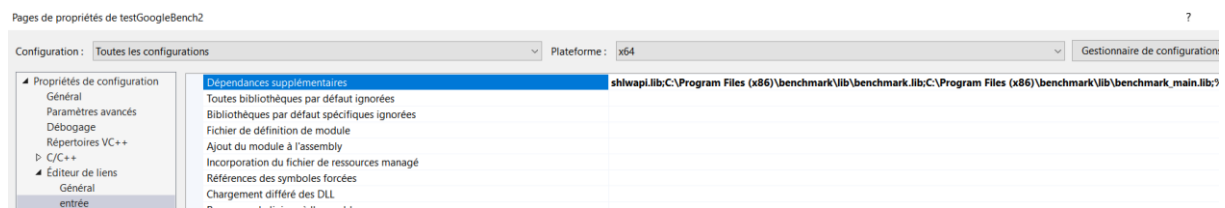


Dans les propriétés du projet, onglet Editeur de liens > entrée > Dépendances supplémentaires : ajouter le chemin vers les fichiers .lib de Google Benchmark ainsi que vers la bibliothèque `shlwapi ([4])`.

`C:\Program Files (x86)\benchmark\lib\benchmark.lib;`

`C:\Program Files (x86)\benchmark\lib\benchmark_main.lib;`

`shlwapi.lib`



**Remarque :** comme on a compilé la bibliothèque Google Benchmark en mode Release (ce qui paraît logique car les benchmarks se font a priori toujours en Release), Visual Studio va signaler un problème de runtime si on essaie de compiler avec Google Benchmark en mode Debug.

Il n'y a pas vraiment de raison pour utiliser Google Benchmark en mode Debug mais si on tient à le faire, on peut, soit recompiler Google Benchmark en mode Debug et utiliser les .lib ainsi générés à la

place des fichiers .lib précédents lorsque l'on compile en Debug (option la plus propre), soit supprimer le flag `_DEBUG` dans les propriétés du projet, onglet C/C++ > préprocesseur > Définitions de préprocesseurs.

## 2.2 Tester son installation sur un exemple

Pour vérifier que le projet a été paramétré correctement, on peut tester l'exemple suivant fourni sur le github de Google Benchmark :

```
#include <benchmark/benchmark.h>

static void BM_StringCreation(benchmark::State& state) {
    for (auto _ : state)
        std::string empty_string;
}

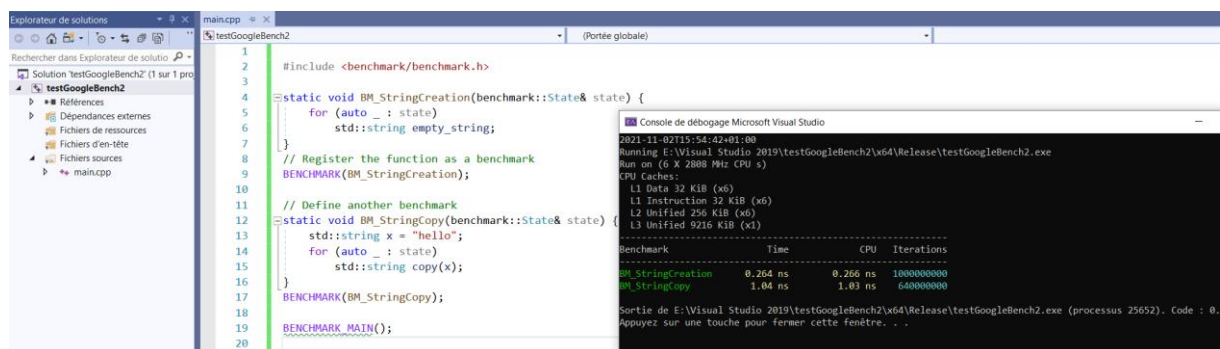
// Register the function as a benchmark
BENCHMARK(BM_StringCreation);

// Define another benchmark
static void BM_StringCopy(benchmark::State& state) {
    std::string x = "hello";
    for (auto _ : state)
        std::string copy(x);
}

BENCHMARK(BM_StringCopy);

BENCHMARK_MAIN();
```

Pour cela, il suffit de copier ce bout de code dans un fichier main.cpp et d'exécuter :



```
2021-11-02T15:54:42+01:00
Running E:\Visual Studio 2019\testGoogleBench2\x64\Release\testGoogleBench2.exe
Run on (6 X 2888 MHz CPU s)
CPU Caches:
  L1 Data 32 KIB (x6)
  L1 Instruction 32 KIB (x6)
  L2 Unified 256 KIB (x6)
  L3 Unified 9216 KIB (x1)
-----
Benchmark              Time           CPU Iterations
-----
BM_StringCreation      0.264 ns       0.266 ns   1000000000
BM_StringCopy          1.04 ns        1.03 ns    640000000

Sortie de E:\Visual Studio 2019\testGoogleBench2\x64\Release\testGoogleBench2.exe (processus 25652). Code : 0.
Appuyez sur une touche pour fermer cette fenêtre. . .
```

## 2.3 Utilisation basique de Google Benchmark

1. Créer la fonction dont on souhaite connaître la performance. Par exemple, créons une fonction qui calcule la somme de tous les éléments d'un tableau :

```
long my_accumulate(const vector<int> & v, long start)
{
    for (int i : v)
        start += i;
}
```

```
    return start;
}
```

2. Appeler cette fonction dans un *Wrapper* qui prend en paramètre un objet de type `benchmark::State`. La fonction à évaluer doit être appelée dans une boucle qui itère sur les éléments de `benchmark::State` :

```
const int vSize = 10000;

void my_accumulate_bench(benchmark::State& state)
{
    vector<int> v(vSize);
    srand(777);
    generate(v.begin(), v.end(), [] {return rand() % 5; });

    long result = 0;

    for (auto _ : state)
        benchmark::DoNotOptimize(result = my_accumulate(v, 0));
}
```

Le nombre d'itérations de la boucle `for (auto _ : state)` sera déterminé dynamiquement par la bibliothèque : elle itère jusqu'à obtenir une mesure du temps CPU statistiquement stable de ce qui se trouve à l'intérieur de la boucle.

La fonction `benchmark::DoNotOptimize()` force le compilateur à sauvegarder le résultat de l'appel de la fonction dans la mémoire. Ceci évite que le compilateur optimise cette boucle en la supprimant (en effet, le compilateur est capable de se rendre compte que cette boucle est inutile au programme car le résultat n'est jamais utilisé).

3. Ajouter cette fonction à la liste des benchmark : macro `BENCHMARK`

```
BENCHMARK(my_accumulate_bench);
```

4. Créer la fonction main : utiliser la macro fournie par la bibliothèque

```
BENCHMARK_MAIN();
```

Code complet :

```
#include <benchmark/benchmark.h>
#include <vector>

using namespace std;
const int vSize = 10000;

long my_accumulate(const vector<int> & v, long start)
{
    for (int i : v)
        start += i;

    return start;
}
```

```

void my_accumulate_bench(benchmark::State& state)
{
    vector<int> v(vSize);
    srand(777);
    generate(v.begin(), v.end(), [] {return rand() % 5; });

    long result = 0;

    for (auto _ : state)
        benchmark::DoNotOptimize(result = my_accumulate(v, 0));
}

BENCHMARK(my_accumulate_bench);

BENCHMARK_MAIN();

```

## 5. Lire le résultat de l'exécution

```

2021-11-03T10:54:03+01:00
Running C:\Users\helene\Documents\PROJETS_VS2019\testGoogleBench2\
Run on (8 X 1800 MHz CPU s)
CPU Caches:
  L1 Data 32 KiB (x4)
  L1 Instruction 32 KiB (x4)
  L2 Unified 256 KiB (x4)
  L3 Unified 6144 KiB (x1)
-----
Benchmark                Time          CPU    Iterations
-----
my_accumulate_bench      732 ns         732 ns      896000

```

La bibliothèque affiche des informations sur la taille des différents niveaux de cache, puis un tableau dans lequel chaque ligne correspond à l'évaluation d'une fonction :

- la colonne Time donne le temps complet moyen d'un appel à la fonction ;
- la colonne CPU donne le temps moyen pour exécuter une fois la fonction ;
- la colonne Iterations donne le nombre d'itérations total de la boucle `for (auto _ : state)` qui a été effectué pour arriver à une mesure CPU statistiquement stable de la fonction.

**NB.** la valeur dans la colonne Time peut être légèrement supérieure à celle dans la colonne CPU car elle donne le temps d'exécution moyen du *wrapper* (pas seulement l'appel à la fonction évaluée).

Pour plus d'information sur les différentes fonctionnalités de Google Benchmark voir [5].

## 3 Google Benchmark vs évaluation à la main

On peut se demander quel est l'impact de la bibliothèque sur les temps de calcul. Autrement dit, les mesures de CPU fournis par la bibliothèque sont-elles fiables ?

Pour répondre à cette question on va comparer deux fonctions qui somment les éléments d'un tableau : la fonction qui a servi d'exemple dans la section précédente et la fonction *accumulate* ([6]) de la bibliothèque standard.

### 3.1 Comparaison en utilisant Google Benchmark sous Windows / Visual Studio 2019

Le code suivant utilise Google Benchmark pour comparer l'efficacité des 2 fonctions :

```
#include <benchmark/benchmark.h>
#include <vector>
#include <algorithm>
#include <numeric>

using namespace std;
const int vSize = 10000;

long my_accumulate(const vector<int> & v, long start)
{
    for (int i : v)
        start += i;

    return start;
}

void my_accumulate_bench(benchmark::State& state)
{
    vector<int> v(vSize);
    srand(777);
    generate(v.begin(), v.end(), [] {return rand() % 5; });

    long result = 0;
    for (auto _ : state)
        benchmark::DoNotOptimize(result = my_accumulate(v, 0));
}

void accumulate_bench(benchmark::State& state)
{
    vector<int> v(vSize);
    srand(777);
    generate(v.begin(), v.end(), [] {return rand() % 5; });

    long result = 0;
    for (auto _ : state)
        benchmark::DoNotOptimize(result = accumulate(v.begin(), v.end(), 0));
}

BENCHMARK(accumulate_bench);
BENCHMARK(my_accumulate_bench);

BENCHMARK_MAIN();
```

Le résultat est le suivant :

Benchmark	Time	CPU	Iterations
accumulate_bench	1036 ns	1050 ns	640000
my_accumulate_bench	720 ns	715 ns	896000

On constate un CPU moyen de 715 nano secondes pour notre fonction contre 1050 nano secondes pour la fonction accumule de la STL soit un gap de 46%.

Le code suivant compare les deux mêmes fonctions sans utiliser Google Benchmark : on lance un million de fois chaque fonction pour avoir une moyenne fiable et on utilise le résultat de la fonction en l'affichant pour être sûr que le compilateur ne va pas supprimer la boucle.

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <numeric>
#include <chrono>

using namespace std;

int my_accumulate(const vector<int>& v, int start)
{
    for (int i : v)
        start += i;

    return start;
}

int main()
{
    const int vSize = 10000;
    const int nbRun = 1000000;

    vector<int> v(vSize);

    //-----
    //0. remplit le vector avec des nombres aléatoires
    generate(v.begin(), v.end(), [] {return rand() % 5; });

    int my_sum = 0, acc_sum = 0;

    //-----
    //1. test de notre fonction ad hoc

    auto start = chrono::steady_clock::now();

    for (int cpt = 0; cpt < nbRun; ++cpt)
        my_sum = my_accumulate(v, 0);

    auto end = chrono::steady_clock::now();
    chrono::duration<double> diff = end - start;
    cout << "my_accumulate : " << diff.count() << endl;

    //-----
    //2. test de accumulate de la STL

    start = chrono::steady_clock::now();

    for (int cpt = 0; cpt < nbRun; ++cpt)
        acc_sum = accumulate(v.begin(), v.end(), 0);

    end = chrono::steady_clock::now();
    diff = end - start;
    cout << "accumulate : " << diff.count() << endl;
```



```

    //on utilise les résultats pour éviter que le compilateur optimise en supprimant
    tout
    cout << my_sum << endl;
    cout << acc_sum << endl;

    return 0;
}

```

On obtient un temps CPU de 0.725 secondes pour un million d'appels à `my_accumulate`, soit un CPU moyen de 725 nano secondes par appel.

On obtient un temps CPU de 0.708 secondes pour un million d'appels à `std::accumulate`, soit un CPU moyen de 708 nano secondes par appel.

Pour la fonction ad hoc `my_accumulate` les résultats obtenus « à la main » correspondent à peu près aux résultats obtenus par Google Benchmark (725 ns vs 715 ns). Par contre pour la fonction de la STL `std::accumulate` les résultats sont très différents : 708 ns contre 1050 ns avec Google Benchmark.

Cette différence semble venir de l'utilisation de `benchmark::DoNotOptimize` (car quand on le supprime et qu'on affiche le résultat pour éviter que le compilateur n'optimise en supprimant complètement la boucle alors on retombe sur des résultats similaires).

## 3.2 Comparaison en utilisant Google Benchmark sous Linux / gcc 9.3.0

Testons maintenant la même chose sous linux en compilant avec gcc 9.3.0 et l'option `-O2`.

Le programme qui utilise Google Benchmark donne 4398 ns pour la fonction `accumulate` de la STL contre 6101 ns pour la nôtre.

Benchmark	Time	CPU	Iterations
<code>accumulate_bench</code>	4398 ns	4398 ns	159024
<code>my_accumulate_bench</code>	6101 ns	6101 ns	114848

Si on teste avec le second programme (qui n'utilise pas Google Benchmark) on trouve 4,42 secondes pour un million d'appels à `my_accumulate` (soit 4420 ns en moyenne par appel) contre 4,40 secondes pour un million d'appels à `std::accumulate` (soit 4400 ns en moyenne par appel).

De nouveau, il semble que la fonction `benchmark::DoNotOptimize` perturbe les temps d'exécution.

## 3.3 Utilisation du site web <https://quick-bench.com>

Le même code exécuté sur le site <https://quick-bench.com> donne des résultats légèrement meilleurs pour la fonction `std::accumulate` de la STL (qu'on utilise `benchmark::DoNotOptimize` ou pas). Sur la figure suivante le temps de calcul pour `std::accumulate` apparaît en bleu, et en rose pour notre fonction `my_accumulate`.

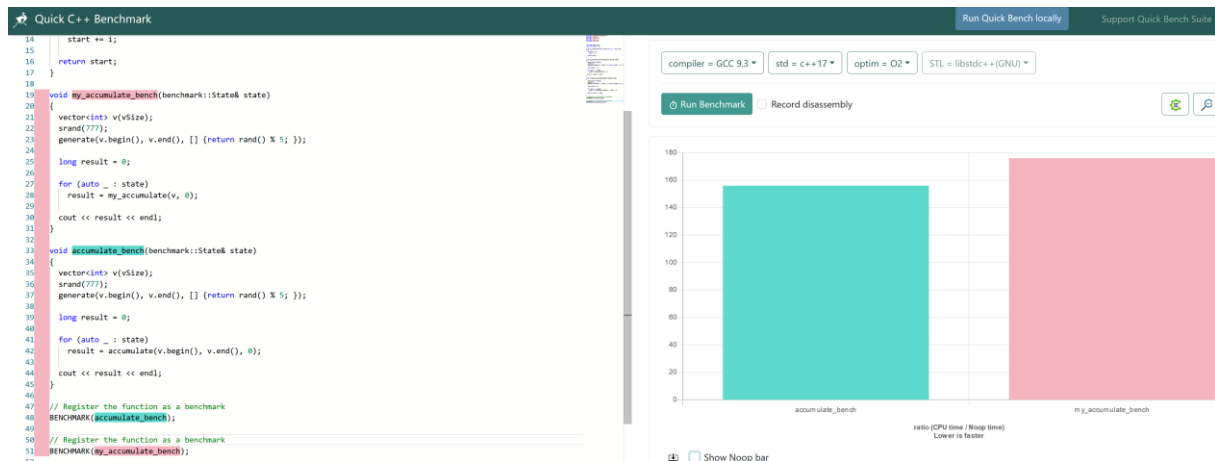


Figure 1. Résultats obtenus avec le site <https://quick-bench.com>  
(en bleu le cpu moyen de `std::accumulate` et en rose celui de `my_accumulate`)

### 3.4 Synthèse des différents tests

On résume dans le tableau ci-dessous les temps d'exécution moyen par appel obtenus.

	Evaluation avec Google Benchmark Et <code>DoNotOptimize</code>		Evaluation avec Google Benchmark Sans <code>DoNotOptimize</code>		Evaluation à la main	
	my_acc. CPU / appel	std::acc. CPU / appel	my_acc. CPU / appel	std::acc. CPU / appel	my_acc. CPU / appel	std::acc. CPU / appel
Windows + VS 2019	715 ns	1050 ns	698 ns	711 ns	725 ns	708 ns
Linux + gcc 9.3.0	6101 ns	4398 ns	4988 ns	4429 ns	4420 ns	4400 ns
<a href="https://quick-bench.com">https://quick-bench.com</a> gcc 9.3, c++17, -O2	167 ms	155 ms	174 ms	153 ms	-	-

Comme on peut le voir dans le tableau, une des difficultés du benchmark en C++ réside dans le fait que les résultats sont fortement impactés par la plateforme et le compilateur utilisé.

Cependant, on peut déduire du tableau de résultats, que l'utilisation de `benchmark::DoNotOptimize` perturbe assez fortement les temps de calcul. Il semble donc préférable de ne pas utiliser cette fonctionnalité. Le site <https://quick-bench.com> donne des résultats assez similaires à ce qu'on obtient en faisant des tests à la main. Il semble donc une option très intéressante étant donné sa facilité d'utilisation. Néanmoins il faudrait faire d'autres tests pour valider ces déductions.

## 4 Références

- [1] <https://github.com/google/benchmark>
- [2] <https://docs.microsoft.com/en-us/cpp/build/building-on-the-command-line?view=msvc-160>
- [3] [https://docs.microsoft.com/en-us/cpp/build/building-on-the-command-line?view=msvc-160#developer\\_command\\_prompt\\_shortcuts](https://docs.microsoft.com/en-us/cpp/build/building-on-the-command-line?view=msvc-160#developer_command_prompt_shortcuts)
- [4] [https://github.com/google/benchmark/blob/main/docs/platform\\_specific\\_build\\_instructions.md](https://github.com/google/benchmark/blob/main/docs/platform_specific_build_instructions.md)
- [5] Google benchmark. User Guide.  
[https://github.com/google/benchmark/blob/main/docs/user\\_guide.md](https://github.com/google/benchmark/blob/main/docs/user_guide.md)
- [6] Cpp Reference. `std::accumulate`. <https://en.cppreference.com/w/cpp/algorithm/accumulate>