Compilation en C++

C++ Visual Studio Code

Les étapes de la compilation

Processus de compilation C++

Le processus de compilation C++ comporte **4 étapes principales** :

1. **Écriture du programme**
   - Création du fichier source (ex : `Prog.cpp`)
2. **Compilation**
   - Transformation du code source en code objet (ex : `Prog.o`)
3. **Édition des liens**
   - Regroupement des fichiers objets pour produire l'exécutable (ex : `ProgExec`)
4. **Exécution**
   - Lancement du programme par l'utilisateur ou le système

Commandes associées à chaque étape

Étape	Action	Fichier/Résultat	Commande C++
1. Écriture	Code source C++	Prog.cpp	*(aucune commande à ce stade)*
2. Compilation	Génération du code objet	Prog.o	`g++ -c Prog.cpp -o Prog.o`
3. Édition des liens	Création de l'exécutable	ProgExec	`g++ Prog.o -o ProgExec`
4. Exécution	Lancement du programme	(résultat affiché)	`./ProgExec`

Détails

Étape	Action	Fichier/Résultat	Commande C++
1. Écriture	Code source C++	Prog.cpp	*(aucune commande)*
2. Prétraitement	Remplace les directives #, macros...	Prog.i	`g++ -E Prog.cpp -o Prog.i`
3. Compilation	Génère le code assembleur	Prog.s	`g++ -S Prog.cpp -o Prog.s`
4. Assemblage	Transforme l'assembleur en code objet	Prog.o	`g++ -c Prog.cpp -o Prog.o`

Étape	Action	Fichier/Résultat	Commande C++
5. Édition des liens	Crée l'exécutable final	ProgExec	`g++ Prog.o -o ProgExec`
6. Exécution	Lance le programme	(résultat affiché)	`./ProgExec`

Notes :

- `.i` et `.s` sont optionnels : ils sont utiles pour comprendre le processus ou pour déboguer.
- La compilation standard ne crée que `.o` ; les autres étapes sont incluses automatiquement.
- Chaque commande intermédiaire permet d'observer l'évolution du code de source à exécutable.

Commande unique pour compiler et lier directement (tout en un) :

```
g++ Prog.cpp -o ProgExec
```

Détails internes : du prétraitement à l'exécutable

Pour une gestion avancée, chaque étape interne peut être observée et générer un fichier :

- **Prétraitement** :
  Nettoie et adapte le code source (macros, directives)
  Commande : `g++ -E Prog.cpp`
  Fichier : `Prog.i`
- **Compilation** :
  Traduit le code prétraité en assembleur
  Commande : `g++ -S Prog.cpp`
  Fichier : `Prog.s`
- **Assemblage** :
  Traduit le code assembleur en objet binaire
  Commande : `g++ -c Prog.cpp`
  Fichier : `Prog.o`
- **Édition des liens** :
  Fusionne les fichiers objets et les bibliothèques
  Commande : `g++ Prog.o -o ProgExec`
  Fichier : `ProgExec`

Contenu du dossier après compilation

Après compilation et édition des liens, on peut trouver :

- `Prog.cpp` : le code source C++

- `Prog.i` : le code source après prétraitement
- `Prog.s` : le code assembleur généré
- `Prog.o` : le fichier objet intermédiaire
- `ProgExec` : l'exécutable final prêt à être lancé

Les fichiers générés

Programme source

```
/*****************************************************************
                          Prog.cpp
*****************************************************************/

#include <iostream>
using namespace std;

#define A 5   // macro remplacée au prétraitement

int main() {
    int b = 2;
    int c = A + b;

    cout << "Valeur de A : " << A << endl;
    cout << "Valeur de b : " << b << endl;
    cout << "La somme de " << A << " + " << b << " = " << c << endl;

    return 0;
}
```

Programme après prétraitement (extrait simplifié)

```
// Fichier Prog.i (extrait simplifié)

using namespace std;

int main() {
    int b = 2;
    int c = 5 + b;

    cout << "Valeur de A : " << 5 << endl;
    ...
}
```

Code assembleur (extrait)

```
.globl  _main
_main:
    pushq   %rbp
    movq    %rsp, %rbp
    ...
L_.str:
    .asciz  "Valeur de A : "
```

Fichier objet (extrait hexadécimal)

Fichier binaire, seulement lisible en hexadécimal extrait :

```
cffa edfe 0700 0001 0300 0000 0100 0000
0400 0000 ...
```

Exemple 1: Fichier unique

Fichier source : `Prog.cpp`

```cpp
#include <iostream>
int main() {
    std::cout << "C++, compilation !\n";
    return 0;
}
```

Commandes de compilation :

- Prétraitement :
  `g++ -E Prog.cpp -o Prog.i`
- Compilation (assembleur) :
  `g++ -S Prog.cpp -o Prog.s`
- Assemblage (objet) :
  `g++ -c Prog.cpp -o Prog.o`
- Edition des liens (exécutable) :
  `g++ Prog.o -o ProgExec`
- Exécution :
  `./ProgExec`

Commande unique (compilation + édition des liens d'un coup) :

```
g++ Prog.cpp -o ProgExec
```

Exemple 2 : Projet multi-fichiers

On sépare le code en trois fichiers pour illustrer la compilation des dépendances.

Fichiers à créer

- `main.cpp`
- `code.cpp`
- `code.h`

Contenu de chaque fichier :

code.h

```
#pragma once
void afficherMessage();
```

code.cpp

```
#include "code.h"
#include <iostream>
void afficherMessage() {
    std::cout << "C++, compilation multi-fichiers !\n";
}
```

main.cpp

```
#include "code.h"
int main() {
    afficherMessage();
    return 0;
}
```

Commandes de compilation et de liens :

- Compilation des objets :
  - `g++ -c main.cpp -o main.o`
  - `g++ -c code.cpp -o code.o`
- Edition des liens (génération de l'exécutable) :
  - `g++ main.o code.o -o ProgExec`

Commande unique (tout compiler d'un coup) :

```
g++ main.cpp code.cpp -o ProgExec
```

Résumé

- Dans un projet multi-fichiers, chaque fichier `.cpp` est compilé séparément en `.o`
- L'étape finale de l'édition des liens regroupe tous les `.o` pour produire l'exécutable
- Utiliser un fichier d'en-tête (.h) permet de factoriser les déclarations

---

Remarques

`#pragma` est une directive de préprocesseur en C/C++ qui sert à transmettre des instructions spécifiques au compilateur.
Dans un fichier `.h` (en-tête), on rencontre très souvent :

```
#pragma once
```

Utilité de `#pragma once`

- **But** : indiquer au compilateur d'inclure ce fichier d'en-tête **une seule fois** par compilation, même s'il est inclus plusieurs fois dans différents fichiers.
- **Effet** : évite les problèmes de double inclusion, donc protège contre les erreurs comme "redefinition", surtout pour les déclarations de fonctions, de classes ou de structures.

Ancienne méthode : include guard

```
#ifndef MONFICHIER_H
#define MONFICHIER_H

// déclarations

#endif // MONFICHIER_H
```

Méthode moderne : pragma once

```
#pragma once

// déclarations
```

Avantages de `#pragma once`

- Syntaxe **plus concise**
- **Moins d'erreurs** (pas besoin d'inventer un identifiant unique)
- **Lecture facilitée** du code
- Prise en charge par la plupart des compilateurs C++ modernes

---

Organisation d'un projet C++

Exemple d'arborescence classique pour un projet C++ structuré :

```
Projet C++
│
├── src/
│      ├── main.cpp
│      ├── prog1.cpp
│      └── prog2.cpp
├── include/
│      ├── header1.hpp
│      └── header2.hpp
├── build/    (dossier généré)
│      ├── prog1.o
│      ├── prog2.o
│      └── main.o
└── bin/
       └── ProgrammeFinal
```

La fonction `main`

Rôle de la fonction main

La fonction `main` est le **point d'entrée** d'un programme C++.
Elle retourne un entier pour indiquer l'état d'exécution :

- **0** : terminaison normale (succès)
- **valeur non nulle** : erreur ou terminaison anormale

Dans certains cas, `main` peut recevoir des arguments de la ligne de commande :
`./Programme arg1 arg2 arg3`

Les paramètres sont transmis automatiquement à `main` par l'environnement d'exécution.

Paramètres de la fonction main

La fonction `main` peut prendre deux paramètres :

- `argc` : nombre d'arguments (y compris le nom du programme)
- `argv` : tableau contenant chaque argument sous forme de chaîne de caractères (`char*`)

Ainsi :

- `argv[0]` : nom de l'exécutable
- `argv[1]` : premier argument
- `argv[argc-1]` : dernier argument

Le prototype complet :

```
int main(int argc, char *argv[]);
```

Structure de `argv`

En mémoire, `argv` est un tableau de pointeurs vers des chaînes de caractères :

- `argv[0]` → `"NomDuProgramme"`
- `argv[1]` → `"arg1"`
- `argv[2]` → `"arg2"`
- ...
- `argv[argc-1]` → dernier argument
- `argv[argc]` → `NULL` (fin du tableau)

Chaque `argv[i]` pointe vers une zone mémoire contenant la chaîne de caractères correspondante.

Exemple : somme de deux entiers en arguments

Exemple en C++, utilisant la gestion d'erreur :

```
#include <iostream>
#include <cstdlib>   // std::atoi

using namespace std;

int main(int argc, char** argv) {
    if (argc != 3) {
        cerr << "Usage : " << argv[0] << " <int> <int>\n";
        return EXIT_FAILURE;
    }

    int a = std::atoi(argv[1]);
    int b = std::atoi(argv[2]);

    cout << "La somme de " << a << " et " << b
         << " vaut : " << a + b << endl;

    return EXIT_SUCCESS;
}
```

Exécution :

Commandé :
`./Programme 15 10`
Affichage :
`La somme de 15 et 10 vaut : 25`

Affichage de tous les arguments reçus

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

int main(int argc, char** argv) {
    cout << "Nombre d'arguments : " << argc << endl;

    for (int i = 0; i < argc; ++i) {
        cout << "Argument " << i << " : " << argv[i] << endl;
    }
    return 0;
}
```

Commande :
`./NomProExe Charles Carnus`

Affichage :

```
Nombre d'arguments : 3
Argument 0 : ./NomProExe
Argument 1 : Charles
Argument 2 : Carnus
```

Prototypes valides de la fonction main

En C++ : **Seulement deux prototypes officiels et portables :**

- `int main()`
  (sans argument)
- `int main(int argc, char *argv[])`
  (avec arguments)

Remarques :

- `int main()` est équivalent à `int main(void)` (“aucun argument”)
- La valeur de retour est un entier ;
  si on omet `return 0;`, le standard considère que le `main` retourne 0 automatiquement
- Tout autre prototype (`main(int, char**, char**)`, etc.) n’est **pas standard** et dépend du compilateur

---

Directives de préprocesseur

Le préprocesseur C/C++ intervient **avant la compilation** et transforme le code source à l’aide de directives débutant par `#`.

Types principaux de directives

- **Inclusion de fichiers** : `#include`
- **Définition de constantes/macros** : `#define`
- **Compilation conditionnelle** : `#if`, `#ifdef`, `#ifndef`, `#elif`, `#else`, `#endif`

Directive `#include`

Elle insère le contenu d'un fichier externe (souvent un fichier d'en-tête).

- `#include <fichier>` : recherche dans les dossiers système (exemple : bibliothèque standard)
- `#include "fichier"` : recherche prioritairement dans le dossier local du projet
- Ajout d'autres dossiers possible via l'option `-I` du compilateur `g++`

Définir des constantes et macros

La directive `#define` sert à créer des :

- **Constantes symboliques**
- **Macros avec paramètres**

Pour supprimer une macro : `#undef`

Exemple :

```
#define NB_LIGNES 10
#define NB_COLONNES 6
#define TAILLE (NB_LIGNES * NB_COLONNES)
```

Macros paramétrées

Syntaxe :

```
#define NOM(paramètres) corps
```

Exemple :

```
#define MAX(a, b) ((a) > (b) ? (a) : (b))
```

- Le préprocesseur effectue uniquement un **remplacement textuel** de `MAX(x,y)` par `((x) > (y) ? (x) : (y))`
- Toujours entourer paramètres et corps de parenthèses pour éviter les erreurs d'évaluation

```
#define CARRE(a) ((a) * (a))
```

Compilation conditionnelle

Permet de (dés)activer du code selon les conditions ou la présence d'un symbole pré-défini.

Structures types :

```
#if EXPRESSION
    // code
#elif AUTRE_EXPRESSION
    // autre code
#else
    // code par défaut
#endif
```

ou

```
#ifdef SYMBOLE
    // code si SYMBOLE existe
#else
    // code sinon
#endif
```

Exemple d'utilisation :

```
#define DEBUG

#ifdef DEBUG
    std::cout << "Debug actif\n";
#endif
```

Limiter l'usage des macros en C++ moderne

Les macros présentent des risques :

- **Pas de typage** → erreurs silencieuses
- **Portée globale**
- **Débogage difficile**
- Simple **remplacement textuel**

Alternatives modernes :

- `const int`, `constexpr` pour les constantes typées
- Fonctions `inline` pour remplacer les petites macros fonctionnelles
- `enum class` pour des ensembles de valeurs

À privilégier : Utiliser `#define` seulement si les solutions C++ ne conviennent pas.

Makefile — Automatiser la compilation

Un **Makefile** permet à l'outil `make` d'automatiser la compilation, de ne recompiler que ce qui change, de gérer les dépendances et de centraliser les options.
Avantage : compilation efficace, moins d'erreurs, commande unique : `make`.

Structure d'une règle

Une règle Makefile :

```
cible : dépendances
    commande   # indenter par tabulation !
```

- **cible** : à produire (`main`, `.o`, etc.)
- **dépendances** : nécessaires pour la cible
- **commande** : exécutée par `make`

Exemple (C++)

```
main: main.o
    g++ -o main main.o

main.o: main.cpp
    g++ -c main.cpp
```

Commande :

`make`

Fonctionnement

- `make` vérifie la cible (ex : `main`)
- Si un fichier requis est absent ou modifié (ex : `main.o`), la commande associée est exécutée
- Les dépendances (.cpp, .h) déclenchent la recompilation des .o concernés
- Si rien n'a changé : `Nothing to be done.`

Projet multi-fichiers — Exemple

```
# main.cpp -- point.cpp -- carre.cpp

main: main.o point.o carre.o
    g++ -o main main.o point.o carre.o

main.o: main.cpp point.h carre.h
    g++ -c main.cpp

point.o: point.cpp point.h
    g++ -c point.cpp

carre.o: carre.cpp carre.h point.h
    g++ -c carre.cpp
```

Optimisation — Variables et règles

Définir des variables pour ne pas répéter les commandes :

```
CXX      = g++
CXXFLAGS = -Wall -Wextra -std=c++20
OBJ      = main.o point.o carre.o

main: $(OBJ)
    $(CXX) -o main $(OBJ)

main.o: main.cpp point.h carre.h
    $(CXX) $(CXXFLAGS) -c main.cpp
```

Cibles factices

Des cibles comme `clean` n'ont pas de fichier associé :

```
.PHONY: all clean

all: main

clean:
    rm -f *.o main
```

Variables automatiques et règles génériques

- `$@` : la cible
- `$<` : première dépendance

- `$^` : toutes les dépendances

Règle générique :

```
%.o: %.cpp
    $(CXX) $(CXXFLAGS) -c $< -o $@
```

Makefile complet

```
CXX      = g++
CXXFLAGS = -Wall -Wextra -std=c++20

SRC = main.cpp point.cpp carre.cpp
OBJ = $(SRC:.cpp=.o)

main: $(OBJ)
    $(CXX) -o main $(OBJ)

%.o: %.cpp
    $(CXX) $(CXXFLAGS) -c $< -o $@

clean:
    rm -f *.o main

.PHONY: clean
```

Synthèse

- Compilation automatique, précise et rapide
- Fichiers recompilés uniquement si nécessaire
- Règles et variables assurent la lisibilité et la maintenance
- Un seul fichier : tout le projet se recompile avec `make`

TP : Maîtriser Makefile en C++

Objectif

Structurer un projet C++ multi-fichiers avec un Makefile professionnel :

- Compiler automatiquement avec `make`
- Organiser le projet
- Nettoyer les fichiers intermédiaires

Organisation du projet

Créer un dossier `tp-make/` avec :

- `main.cpp`
- `carre.cpp`, `carre.h`
- `Makefile` (vide au départ)

But : construire un programme manipulant un carré.

Codes fournis

carre.h

```
#pragma once

double aireCarre(double cote);
```

carre.cpp

```
#include "carre.h"

double aireCarre(double cote) {
    return cote * cote;
}
```

main.cpp

```
#include <iostream>
#include "carre.h"

int main() {
    std::cout << aireCarre(5) << std::endl;
}
```

Exercice 1 : Compilation manuelle

Compiler le projet étape par étape :

1. `g++ -c carre.cpp`
2. `g++ -c main.cpp`
3. `g++ -o main main.o carre.o`

Exercice 2 : Makefile minimal

Créer un Makefile :

```
main: main.o carre.o
    g++ -o main main.o carre.o

main.o: main.cpp carre.h
    g++ -c main.cpp

carre.o: carre.cpp carre.h
    g++ -c carre.cpp
```

Commande : `make`

Exercice 3 : Utiliser des variables

Ajouter :

```
CXX      = g++
CXXFLAGS = -Wall -Wextra -std=c++20
OBJ      = main.o carre.o

main: $(OBJ)
    $(CXX) -o main $(OBJ)
```

Exercice 4 : Ajouter la cible clean

```
clean:
    rm -f *.o main

.PHONY: clean
```

Commande : `make clean`

Exercice 5 : Règle générique (.cpp → .o)

Ajouter :

```
%.o: %.cpp
    $(CXX) $(CXXFLAGS) -c $< -o $@
```

- `$<` : première dépendance (.cpp)
- `$@` : cible (.o)

Exercice 6 : Makefile final à produire

À partir des exercices précédents, produire un Makefile :

- Propre et lisible
- Utilisant les variables
- Avec une règle générique
- Avec la cible `clean`

Commandes :

```
make
./main
```

Correction : Makefile final

```
CXX      = g++
CXXFLAGS = -Wall -Wextra -std=c++20

SRC = main.cpp carre.cpp
OBJ = $(SRC:.cpp=.o)

main: $(OBJ)
    $(CXX) -o main $(OBJ)

%.o: %.cpp
    $(CXX) $(CXXFLAGS) -c $< -o $@

clean:
    rm -f *.o main

.PHONY: clean
```

Commandes à tester :

- `make`
- `./main`
- `make clean`

Synthèse

- Compilation automatisée et sélective
- Facilité de maintenance et d'évolution
- Nettoyage rapide du projet

Exemple 1 : Programme avec ou sans découpage

Programme en un seul fichier

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

#define a 5 // macro

int main()
{
    int b = 2;
    int c = a + b;

    cout << "a = " << a << endl;
    cout << "b = " << b << endl;
    cout << "c = a + b = " << c << endl;

    return 0;
}
```

Programme découpé en trois fichiers

main.cpp

```cpp
#include <iostream>
using namespace std;
#include "source.hpp"

int main()
{
    afficher(5, 2);
    return 0;
}
```

source.hpp

```cpp
#pragma once

void afficher(int a, int b);
```

source.cpp

```cpp
#include <iostream>
#include "source.hpp"
using namespace std;

void afficher(int a, int b)
{
    cout << "a = " << a << '\n';
```

```
    cout << "b = " << b << endl;
    cout << "c = a + b = " << a + b << '\n';
}
```

Exemple 2 : Découpage d'une fonction

Programme sans découpage

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

double fonc(double x, double y);

int main()
{
    double u = 3, v = 4;
    double w = fonc(u, v);
    cout << "Le résultat est " << w << endl;
    return 0;
}

double fonc(double x, double y)
{
    return x*x + y*y;
}
```

Programme découpé en trois fichiers

main.cpp

```
#include <iostream>
using namespace std;
#include "source.hpp"

int main()
{
    double u = 3, v = 4;
    double w = fonc(u, v);
    cout << "Le résultat est " << w << endl;
    return 0;
}
```

source.hpp

```
```

```
#pragma once

double fonc(double x, double y);
```

source.cpp

```
#include "source.hpp"

double fonc(double x, double y)
{
    return x*x + y*y;
}
```

Schéma du découpage des fichiers

- `.hpp` : déclarations (signatures)
- `.cpp` : définitions (implémentations)
- `main.cpp` : utilise les fonctions déclarées dans le `.hpp`
- Chaque `.cpp` → compilé séparément
- L'éditeur de liens rassemble tout (`linking`)

Relations :

- `main.cpp → source.hpp`
- `source.cpp → source.hpp`
- `main.o + source.o` → `main` (exécutable)

TP : Découper un programme simple

Énoncé du TP

Objectif : Découper le programme ci-dessous en trois fichiers : `main.cpp`, `calculs.hpp`, `calculs.cpp`.

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

int addition(int a, int b) {
    return a + b;
}

int multiplication(int a, int b) {
    return a * b;
}

int main() {
```

```
    int x = 5, y = 3;
    cout << "Addition : " << addition(x, y) << endl;
    cout << "Multiplication : " << multiplication(x, y) << endl;
    return 0;
}
```

Consignes

1. Créer un fichier `calculs.hpp` pour déclarer les fonctions `addition` et `multiplication`.
2. Créer un fichier `calculs.cpp` pour définir les fonctions.
3. Modifier `main.cpp` pour inclure `calculs.hpp` et appeler les fonctions.
4. Compiler avec : `g++ main.cpp calculs.cpp -o mon_programme`

Bonus : Ajouter la fonction `soustraction(int a, int b)` dans `calculs.cpp` et l'utiliser dans `main.cpp`.

Solution découpée en trois fichiers

`calculs.hpp` **(fichier d'en-tête)**

```
#pragma once

int addition(int a, int b);
int multiplication(int a, int b);
int soustraction(int a, int b); // Bonus
```

`calculs.cpp` **(fichier source des fonctions)**

```
#include "calculs.hpp"

int addition(int a, int b) {
    return a + b;
}

int multiplication(int a, int b) {
    return a * b;
}

int soustraction(int a, int b) {
    return a - b;
}
```

`main.cpp` **(programme principal)**

```cpp
#include <iostream>
#include "calculs.hpp"
using namespace std;

int main() {
    int x = 5, y = 3;

    cout << "Addition : " << addition(x, y) << endl;
    cout << "Multiplication : " << multiplication(x, y) << endl;
    cout << "Soustraction : " << soustraction(x, y) << endl; // Bonus

    return 0;
}
```

Compilation

```
g++ main.cpp calculs.cpp -o mon_programme
./mon_programme
```

Résultat attendu :

```
Addition : 8
Multiplication : 15
Soustraction : 2
```