Compilation séparée
L'outil Make

Viviane Pons

Mail : viviane.pons@universite-paris-saclay.fr
Page professionnelle : http://www.lri.fr/~pons

14 mars 2025

Plan

Passage à l'échelle : utiliser plusieurs fichiers

Dès que l'on veut faire un programme un peu gros, tout écrire dans un seul fichier n'est pas pratique :

- Difficile de se repérer dans un fichier trop gros
- Travail à plusieurs, en équipe
- Réutilisation de bibliothèques

Rappels sur la compilation

Retenir

Langage machine *: le seul langage que la machine parle directement (langage binaire, spécifique à chaque machine).*

Par exemple, sur ma machine

```
01001000 10000011 11000000 00010101 = 48 83 c0 15
```

signifie «ajouter 21 (=15 en hexadécimal) au registre `rax`».

Retenir

Langage assembleur *: décodage du langage machine lisible par un humain.*

L'instruction précédente est notée

```
add $0x15,%rax
```

Rappels sur la compilation

Retenir

Langage machine *: le seul langage que la machine parle directement (langage binaire, spécifique à chaque machine).*

Par exemple, sur ma machine

`01001000 10000011 11000000 00010101 = 48 83 c0 15`

signifie «ajouter 21 (=15 en hexadécimal) au registre `rax`».

Retenir

Langage assembleur *: décodage du langage machine lisible par un humain.*

L'instruction précédente est notée

```
add $0x15,%rax
```

Compilation

Retenir

*La **compilation** est la traduction d'un programme écrit en langage évolué vers un programme exécutable par la machine.*

Elle est faite en deux étapes :

- *La compilation proprement dite*
- *L'édition des liens*

Compilation

Retenir

Compilation *: traduction du langage évolué vers langage machine.*

Produit un fichier objet « `.o` *» (pas de rapport avec la prog. objet)*

À la fin, le compilateur a bien **écrit les instructions en langage machine** mais n'a pas encore

- décidé des emplacement mémoires
- résolu les appels vers les fonctions externes

Retenir

*C'est le rôle de l'****édition des liens.***

Fichier objet

Retenir

*Un **fichier objet** (`.o`) :*

- *fichier intermédiaire qui contient du code en langage machine produit par un assembleur ou un compilateur.*
- *est lié à d'autres fichiers lors du processus d'édition des liens pour obtenir un exécutable.*

Contient du **code machine**, ainsi que des meta-données

- nécessaires à l'édition de liens (symboles)
- nécessaires lors de la phase de déboguage.

Exemple

Soit le programme C++

compil.cpp

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5    int m = 47521, n = 62; // b9a1 et 3e en hexadécimal
6    cout << m + n << endl;
7    return 0;
8  }
```

Pour une compilation seule (obtention d'un fichier .o) on compile avec l'option `-c`

```
g++ -std=c++11 -Wall -c compil.cpp
```

On peut ensuite désassembler le fichier `compil.o` produit avec

```
objdump -dS compil.o
```

Exemple de fichier objet

- L'emplacement du programme n'est pas décidé (début à 0)
- Les constantes du programme apparaissent bien
- Les adresses des fonctions externes sont inconnues

compil.o.dS

```
0000000000000000 <main>:
   0:   55                      push   %rbp                                  // mise en place
   1:   48 89 e5                mov    %rsp,%rbp                             //   du tableau d'activation
   4:   48 83 ec 10             sub    $0x10,%rsp                            // réserve 16 octets dans le TA
   8:   c7 45 f8 a1 b9 00 00    movl   $0xb9a1,-0x8(%rbp)                    // Init. de m (adresse -0x8(%rbp))
   f:   c7 45 fc 3e 00 00 00    movl   $0x3e,-0x4(%rbp)                      // Init. de n (adresse -0x4(%rbp))
  16:   8b 55 f8                mov    -0x8(%rbp),%exd                       // Chargement de m dans %edx
  19:   8b 45 fc                mov    -0x4(%rbp),%eax                       // Chargement de n dans %eax
  1c:   01 d0                   add    %edx,%eax                             // Ajout de m et n dans %eax
  1e:   89 c6                   mov    %eax,%esi                             // Appel
  20:   48 8d 3d 00 00 00 00    lea    0x0(%rip),%rdi        # 27 <main+0x27>  //   d'une fonction
  27:   e8 00 00 00 00          callq  2c <main+0x2c>                        //   (operator << sur cout)
  2c:   48 89 c2                mov    %rax,%rdx
  2f:   48 8b 05 00 00 00 00    mov    0x0(%rip),%rax        # 36 <main+0x36>
  36:   48 89 c6                mov    %rax,%rsi
  39:   48 89 d7                mov    %rdx,%rdi
  3c:   e8 00 00 00 00          callq  41 <main+0x41>                        // Deuxième appel
  41:   b8 00 00 00 00          mov    $0x0,%eax                             // valeur de retour <- 0
  46:   c9                      leaveq
  47:   c3                      retq                                         // return
```

Exemple de fichier exécutable

Le même extrait **après l'édition des liens** avec la commande

```
g++ compil.o -o compil
```

compil.dS

```
000000000000088a <main>:
 88a:   55                      push   %rbp
 88b:   48 89 e5                mov    %rsp,%rbp
 88e:   48 83 ec 10             sub    $0x10,%rsp
 892:   c7 45 f8 a1 b9 00 00    movl   $0xb9a1,-0x8(%rbp)                    // Init. de m (adresse -0x8(%rbp))
 899:   c7 45 fc 3e 00 00 00    movl   $0x3e,-0x4(%rbp)                      // Init. de n (adresse -0x4(%rbp))
 8a0:   8b 55 f8                mov    -0x8(%rbp),%edx
 8a3:   8b 45 fc                mov    -0x4(%rbp),%eax
 8a6:   01 d0                   add    %edx,%eax
 8a8:   89 c6                   mov    %eax,%esi
 8aa:   48 8d 3d 6f 07 20 00    lea    0x20076f(%rip),%rdi        # 201020 <_ZSt4cout@@GLIBCXX_3.4>
 8b1:   e8 aa fe ff ff          callq  760 <_ZNSolsEi@plt>
 8b6:   48 89 c2                mov    %rax,%rdx
 8b9:   48 8b 05 10 07 20 00    mov    0x200710(%rip),%rax      # 200fd0 <_ZSt4endlIcSt11char_traitsIcEERSt13basic_ost
 8c0:   48 89 c6                mov    %rax,%rsi
 8c3:   48 89 d7                mov    %rdx,%rdi
 8c6:   e8 75 fe ff ff          callq  740 <_ZNSolsEPFRSoS_E@plt>
 8cb:   b8 00 00 00 00          mov    $0x0,%eax                             // valeur de retour <- 0
 8d0:   c9                      leaveq
 8d1:   c3                      retq
```

Compilation / édition des liens

Retenir

La compilation est faite en deux phases

1. *la **compilation** proprement dite : traduction des instructions en langage machine. Les symboles (variables, constantes, fonctions, . . .) ont seulement besoin d'être déclarés pour être utilisés. Fabrication de **fichier objet «.o»***
2. *l'**édition des liens** : on rassemble le code en vérifiant que chaque symbole utilisé est défini une fois et une seule. Fabrication de **fichier exécutable** ou de bibliothèques (fichiers «.a» ou «.so» sous Linux, «.dll» sous Windows).*

Bilan : Compilation / édition des liens

Retenir

Les types de fichiers :

- *« `.cpp` » : fichier source, contient les définitions des fonctions et méthodes*
- *« `.hpp` » (aussi « `.h` ») : fichier d'entête (header), contient les déclarations des fonctions et classes*
- *« `.o` » : fichier objet (pas de lien avec la programmation objet), contient des composants de programmes compilés*
- *fichier exécutable (sans extension, « `.exe` » sous Windows), application proprement dite.*

Compilation séparée

Retenir

Fichier d'entête « `.hpp` *» (aussi «* `.h` *») :*

- *déclarations des fonctions et classes*
- ***doit être inclus** par*

```
#include <fichier> // pour un fichier du système
ou #include "fichier.hpp" // pour un fichier personnel
```

dès que l'on utilise une fonction ou une classe qui y est déclarée

Retenir

Fichier source « `.cpp` *»*

- *définition des fonctions et classes*
- ***inclut le fichier d'entête correspondant** ainsi que les fichiers d'entête qu'il utilise*

Un mini exemple

Entête :

pgcd.hpp

```
1  #ifndef PGCD_HPP
2  #define PGCD_HPP
3
4  /** Le plus grand diviseur commun de deux nombres
5  * @param[in] a, b : deux entiers
6  * @return le pgcd, toujours positif de a et b
7  **/
8  int pgcd(int a, int b);
9
10 #endif  // PGCD_HPP
```

Utilisation :

pgcd-main.cpp

```
1  #include <iostream>
2  #include <stdexcept>
3  #define DOCTEST_CONFIG_IMPLEMENT
4  #include "doctest.h"
5  using namespace std;
6
7  #include "pgcd.hpp"
8
9  int main(int argc, const char** argv) {
10   int m, n;
11   cout << "Calcul du pgcd de deux nombres:" << endl
12        << "Entrez les deux nombres : ";
13   cin >> m >> n;
14   cout << "Le pgcd est " << pgcd(m, n) << endl;
15 }
```

Source :

pgcd.cpp

```
1  #include <cstdlib>   // pour la fonction abs
2  // Seulement inclure doctest ici
3  // Ne pas mettre le #define DOCTEST_CONFIG_IMPLEMENT
4  // Ni #define DOCTEST_CONFIG_IMPLEMENT_WITH_MAIN
5  #include "doctest.h"
6
7  // Inclure aussi le fichier d'entête
8  #include "pgcd.hpp"
9
10 /** Le plus grand diviseur commun à deux entiers */
11 int pgcd(int a, int b) {
12   a = abs(a); b = abs(b);
13   while (b != 0) {
14     int r = a % b;
15     a = b;
16     b = r;
17   }
18   return a;
19 }
20 TEST_CASE("fonction pgcd") {
21   CHECK(pgcd(15, 12) == 3);
22   CHECK(pgcd(15, 0) == 15);
23   CHECK(pgcd(0, 12) == 12);
24   CHECK(pgcd(-15, 12) == 3);
25   CHECK(pgcd(42, 1) == 1);
26   CHECK(pgcd(1, 42) == 1);
27   CHECK(pgcd(42, 42) == 42);
28   CHECK(pgcd(42, -42) == 42);
29   CHECK(pgcd(-42, -42) == 42);
30   CHECK(pgcd(-42, -42) == 42);
31 }
```

Gardes d'inclusions multiples

Si l'on **déclare deux fois** le même symbole lors de la compilation, le compilateur signale une **erreur**. Cela peut arriver de manière imprévisible, quand un fichier a été inclus deux fois par suite d'une série d'inclusions. Pour résoudre ce problème, chaque fichier `.hpp` doit être gardé contre les inclusions :

Retenir (Garde d'inclusion multiple)

Chaque fichier `.hpp` *doit* ***commencer par*** *:*

```
#ifndef NOMDUFICHIER_HPP
#define NOMDUFICHIER_HPP
```

et ***se terminer par*** *:*

```
#endif
```

Bilan : que doit-on avoir dans un `.hpp`

- Garde d'inclusions multiples :

```
#ifndef NOMDUFICHIER_HPP
#define NOMDUFICHIER_HPP
```

- inclusion des fichiers standards

```
#include <iotream>
```

- inclusion des autres fichiers utilisés

```
#include "monFichierDeBase.hpp"
```

- Définition des constantes

```
const int MAXTAILLE=1000;
```

- Déclaration des fonctions structures et classes
- Fin de la garde d'inclusions multiples

```
#endif
```

Attention

Acune définition de fonctions ne doit figurer dans un `.hpp`, sauf les définitions en ligne incluses dans les déclarations.

Bilan : que doit-on avoir dans un `.cpp`

Retenir

Deux sortes de fichiers sources

1. *bibliothèques de fonctions déclarées dans un `.hpp`*
2. *programme principal avec un `main`*

- inclusion des fichiers standard
- inclusion des autres fichiers utilisés
- si bibliothèque de fonction inclusion du `.hpp` correspondant.
- définition des fonctions et méthodes.
- cas de tests des fonctions et méthodes.

Possibilité de déclarer et définir des fonctions, constantes, classes que l'on ne veux pas montrer à l'utilisateur.

Bilan : que doit-on avoir dans un `.cpp`

Retenir

Deux sortes de fichiers sources

1. *bibliothèques de fonctions déclarées dans un* `.hpp`
2. *programme principal avec un* `main`

- inclusion des fichiers standard
- inclusion des autres fichiers utilisés
- si bibliothèque de fonction inclusion du `.hpp` correspondant.
- définition des fonctions et méthodes.
- cas de tests des fonctions et méthodes.

Possibilité de déclarer et définir des fonctions, constantes, classes que l'on ne veux pas montrer à l'utilisateur.

Commandes de compilation et d'édition des liens

Retenir (Compilation)

L'option `-c` *du compilateur demande de n'effectuer que la compilation :*

`g++` ⟨`options du compilateur`⟩ `-c fichier.cpp`

compile le fichier `fichier.cpp` *en un fichier objet* `fichier.o`

Retenir (Édition des liens)

La commande

`g++` ⟨`options du lieur`⟩ `-o nomExec` ⟨`liste de .o`⟩

lie les fichiers objet `.o` *en un exécutable nommé* `nomExec`.

Exemple de commandes de compilation et d'édition des liens

Pour compiler le programme du pgcd vu précédement, il faut donc lancer les trois commandes suivantes :

- Compile `pgcd.cpp` en `pgcd.o`

  ```
  g++ -std=c++11 -Wall -c pgcd.cpp
  ```

- Compile `pgcd-main.cpp` en `pgcd-main.o`

  ```
  g++ -std=c++11 -Wall -c pgcd-main.cpp
  ```

- Lie `pgcd.o pgcd-main.o` dans l'exécutable `pgcd-main`

  ```
  g++ -o pgcd-main pgcd.o pgcd-main.o
  ```

Plan

L'outil make

Retenir

*Make est un outil qui sert à **exécuter un ensemble d'actions**, comme la compilation d'un projet, l'archivage de document, la mise à jour de site web, . . .*

Par comparaison de date de création/mise à jour il ne fait que le travail nécessaire (ex : ne pas recompiler inutilement).

Les règles de construction sont dans un fichier `Makefile`.

Attention

Les Makefiles ne sont malheureusement pas normalisés. J'utiliserai la version GNU (développé par R. Stallman et R. McGrath). La syntaxe peut varier si vous en utilisez un autre.

L'outil make

Retenir

*Make est un outil qui sert à **exécuter un ensemble d'actions**, comme la compilation d'un projet, l'archivage de document, la mise à jour de site web, . . .*

Par comparaison de date de création/mise à jour il ne fait que le travail nécessaire (ex : ne pas recompiler inutilement).

Les règles de construction sont dans un fichier `Makefile`.

Attention

Les Makefiles ne sont malheureusement pas normalisés. J'utiliserai la version GNU (développé par R. Stallman et R. McGrath). La syntaxe peut varier si vous en utilisez un autre.

Le fichier Makefile

Syntaxe

Un fichier `Makefile` *est composé de* ***règles*** *de la forme*

⟨`cible`⟩ : ⟨`dépendances`⟩
⟶⟨`commande de compilation`⟩

où

- `cible` *est le fichier que l'on veut construire*
- `dépendance` *est la liste des fichiers dont dépend la cible*
- ⟶ *est le caractère de tabulation*

On peut mettre plusieurs lignes de commande si besoin.

Que mettre dans les dépendances

Retenir

Un fichier objet `.o` *dépend*

- *des fichiers* `.cpp`, `.hpp` *de même nom ;*
- *des fichiers* `.hpp` *inclus.*

Attention

Les dépendances d'un fichier `.o` doivent contenir non seulement les deux fichiers `.cpp` et `.hpp` de même nom, mais aussi **tous les fichier `.hpp` qui on été inclus, directement ou indirectement.**

Retenir

Un fichier exécutable dépend de tous les fichiers `.o` *que l'on va lier.*

Que mettre dans les dépendances

Retenir

Un fichier objet `.o` *dépend*

- *des fichiers* `.cpp`, `.hpp` *de même nom ;*
- *des fichiers* `.hpp` *inclus.*

Attention

Les dépendances d'un fichier `.o` doivent contenir non seulement les deux fichiers `.cpp` et `.hpp` de même nom, mais aussi **tous les fichier** `.hpp` **qui on été inclus, directement ou indirectement.**

Retenir

Un fichier exécutable dépend de tous les fichiers `.o` *que l'on va lier.*

Un premier exemple de Makefile

Makefile.v1

```
1  # Les commentaires commencent par un #
2
3  # fichiers objets
4  pgcd-main.o: pgcd-main.cpp pgcd.hpp
5  ————→g++ -std=c++11 -Wall -c pgcd-main.cpp
6  pgcd.o: pgcd.cpp pgcd.hpp
7  ————→g++ -std=c++11 -Wall -c pgcd.cpp
8
9  # fichier exécutable
10 pgcd-main: pgcd.o pgcd-main.o
11 ————→g++ -o pgcd-main pgcd.o pgcd-main.o
```

Que fait make ?

Retenir

La commande du terminal :

`make` ⟨`cible`⟩

demande à make de construire la `cible`*.*

L'évaluation d'une règle se fait récursivement :

1. *les dépendances sont analysées : si une dépendance est la cible d'une autre règle, cette règle est à son tour évaluée ;*
2. *lorsque l'ensemble des dépendances a été analysé, et si une cible est plus ancienne que les dépendances, les commandes correspondant à la règle sont exécutées.*

Quelques autres cibles utiles (1)

L'ordre des cibles ne change rien dans un `Makefile`, sauf :

Retenir

Si l'on lance juste la commande

```
make
```

*la **première cible** du fichier* `Makefile` *est exécutée.*
Ainsi, il est habituel de rajouter une première nouvelle cible que l'on appelle par convention «`all`*» et qui dépend de ce que l'on veut compiler par défaut. On ne met pas de commande pour cette cible.*

Pour s'assurer que `make` crée le programme principal `pgcd-main`, on ajoute donc comme première cible

`Makefile.v2`

```
1 all: pgcd-main
```

Quelques autres cibles utiles (2)

Retenir

Très souvent, on ajoute dans le `Makefile` *d'autres* ***cibles qui ne correspondent pas à la fabrication d'un fichier****. Les exemples suivants sont des cibles standards recommandées :*

- `clean` *: pour supprimer les fichiers compilés (*`.o` *et exécutables)*
- `check` *: pour lancer les tests (après les avoir compilé au besoin)*
- `doc` *: pour fabriquer la documentation (par exemple à partir des chaînes de documentation des fonctions)*
- `install` *: pour installer le programme dans le système*

Quelques autres cibles utiles (3)

On va par exemple ajouter les règles suivantes (et faire un fichier `pgcd-test.cpp`)

Makefile.v2

```
1  check: pgcd-test
2  ————→./pgcd-test
3
4  clean:
5  ————→rm -f *.o pgcd-main pgcd-test
```

Activer les tests dans un fichier à part

Au début du fichier source `pgcd.cpp` :

pgcd.cpp

```
1  #include <cstdlib>   // pour la fonction abs
2  // Seulement inclure doctest ici
3  // Ne pas mettre le #define DOCTEST_CONFIG_IMPLEMENT
4  // Ni #define DOCTEST_CONFIG_IMPLEMENT_WITH_MAIN
5  #include "doctest.h"
6
7  // Inclure aussi le fichier d'entête
8  #include "pgcd.hpp"
```

Le fichier de test ne contient que :

pgcd-test.cpp

```
1  #define DOCTEST_CONFIG_IMPLEMENT_WITH_MAIN
2  #include "doctest.h"
```

Il est **compilé puis lié** avec tous les autres fichiers objets qui contiennent les tests :

Makefile.v2

```
1  pgcd-test: pgcd.o pgcd-test.o
2  ————→g++ -o pgcd-test pgcd.o pgcd-test.o
3  pgcd-test.o: pgcd-test.cpp pgcd.hpp
4  ————→g++ -std=c++11 -Wall -c pgcd-test.cpp
```

Plan

Les variables du Makefile

Retenir

Il est possible de définir des ***variables*** *dans un Makefile. Elles se définissent sous la forme* `NOM=valeur` *et sont appelées sous la forme* `$(NOM)`.

Quelques variables standards :

- `CXX` : le compilateur C++ utilisé
- `CXXFLAGS` : les options de compilation
- `CC`, `CFLAGS` : même chose pour le C
- `LDFLAGS` : les options d'édition de liens

Les variables du Makefile

Retenir

Il est possible de définir des ***variables*** *dans un Makefile. Elles se définissent sous la forme* `NOM=valeur` *et sont appelées sous la forme* `$(NOM)`*.*

Quelques variables standards :

- `CXX` : le compilateur C++ utilisé
- `CXXFLAGS` : les options de compilation
- `CC`, `CFLAGS` : même chose pour le C
- `LDFLAGS` : les options d'édition de liens

Nouveau makefile

Makefile.v3

```
1  CXX=g++    # On pourrait mettre clang++
2  CXXFLAGS= -Wall -std=c++11 # options du compilateur (ex -g : debug -O3 : optimize)
3  LDFLAGS= # options de l'éditeur de lien (rien ici) -lm pour fonct. maths .avancées
4
5  EXEC_FILES= pgcd-main pgcd-test
6
7  all: pgcd-main
8
9  pgcd-main: pgcd.o pgcd-main.o
10 ⟶$(CXX) $(LDFLAGS) -o pgcd-main pgcd.o pgcd-main.o
11 pgcd-main.o: pgcd-main.cpp pgcd.hpp
12 ⟶$(CXX) $(CXXFLAGS) -c pgcd-main.cpp
13 pgcd.o: pgcd.cpp pgcd.hpp
14 ⟶$(CXX) $(CXXFLAGS) -c pgcd.cpp
15 pgcd-test: pgcd.o pgcd-test.o
16 ⟶$(CXX) $(LDFLAGS) -o pgcd-test pgcd.o pgcd-test.o
17 pgcd-test.o: pgcd-test.cpp pgcd.hpp
18 ⟶$(CXX) $(CXXFLAGS) -c pgcd-test.cpp
19
20 check: pgcd-test
21 ⟶./pgcd-test
22
23 clean:
24 ⟶rm -f *.o $(EXEC_FILES)
```

Les variables internes au Makefile

Compléments

Certaines variables sont **définies automatiquement** lors de l'exécution d'une commande ;

- `$@` : nom de la cible
- `$<` : nom de la première dépendance
- `$^` : liste des dépendances
- `$?` : liste des dépendances plus récentes que la cible
- `$*` : nom d'un fichier sans son suffixe

On peut donc réécrire toutes les règles...

Nouveau makefile

Makefile.v4

```
1  CXX=g++  # On pourrait mettre clang++
2  CXXFLAGS= -Wall -std=c++11 # options du compilateur (ex -g : debug -O3 : optimize)
3  LDFLAGS= # options de l'éditeur de lien (rien ici) -lm pour fonct. maths .avancées
4  EXEC_FILES= pgcd-main pgcd-test
5
6  all: pgcd-main
7
8  pgcd-main: pgcd.o pgcd-main.o
9  ------>$(CXX) $(LDFLAGS) -o $@ $^
10 pgcd-main.o: pgcd-main.cpp pgcd.hpp
11 ------>$(CXX) $(CXXFLAGS) -c $<
12
13 pgcd.o: pgcd.cpp pgcd.hpp
14 ------>$(CXX) $(CXXFLAGS) -c $<
15
16 pgcd-test: pgcd.o pgcd-test.o
17 ------>$(CXX) $(LDFLAGS) -o $@ $^
18 pgcd-test.o: pgcd-test.cpp pgcd.hpp
19 ------>$(CXX) $(CXXFLAGS) -c $<
20
21 check: pgcd-test
22 ------>./pgcd-test
23
24 clean:
25 ------>rm -f *.o $(EXEC_FILES)
```

Plan

Règles génériques

Il y a beaucoup de redondances dans le précédent `Makefile`.

Compléments

On peut écrire des règles de fabrication génériques : il suffit d'utiliser le symbole % à la fois pour la cible et pour la dépendance (voir exemple suivant).

On peut aussi préciser séparément d'autres dépendances pour les cas particuliers.

Nouveau makefile

Makefile.v5

```
1  CXX=g++
2  CXXFLAGS= -Wall -std=c++11
3  LDFLAGS=
4  EXEC_FILES= pgcd-main pgcd-test
5
6  all: pgcd-main
7  # Règle générique pour fabriquer un .o a partir d'un .cpp
8  %.o: %.cpp
9  ⟶$(CXX) $(CXXFLAGS) -c $<
10
11 pgcd-main: pgcd.o pgcd-main.o
12 ⟶$(CXX) $(LDFLAGS) -o $@ $^
13 pgcd-test: pgcd.o pgcd-test.o
14 ⟶$(CXX) $(LDFLAGS) -o $@ $^
15
16 pgcd-main.o: pgcd.hpp
17 pgcd.o: pgcd.hpp
18 pgcd-test.o: pgcd.hpp
19
20 check: pgcd-test
21 ⟶./pgcd-test
22
23 clean:
24 ⟶rm -f *.o $(EXEC_FILES)
```

Faire des calculs dans le Makefile

Compléments

On peut faire des calculs dans un Makefile !

- Mettre dans `SRC` tous les fichiers `.cpp` :

```
SRC=$(wildcard *.cpp)
```

- remplacer l'extension `.cpp` par `.o` :

```
OBJ=$(SRC:.cpp=.o)    # Attention aux espaces
```

- afficher une variable

```
$(info la variable $$(OBJ) contient $(OBJ))
```

Exemple complet : les dates

Voir les fichiers sources sur le web :

date.hpp

```
/* Exemple de fichier d'entête : date.hpp */
```

date.cpp

```
/* Exemple de fichier source : date.cpp */
```

date-main.cpp

```
/* Exemple de fichier principal : date-main.cpp */
```

Exemple complet : les dates

Makefile

```
1  CXX=clang++
2  CXXFLAGS= -Wall -std=c++11 -g -O3
3  EXEC_FILES= date-main
4  SRC= $(wildcard *.cpp) # tous les fichiers .cpp du répertoire
5  OBJ=$(SRC:.cpp=.o)     # attention espaces qui sont pris en compte
6
7  all: $(EXEC_FILES)
8
9  # Règle générique
10 %.o: %.cpp
11 ⟶$(CXX) -c $< $(CFLAGS)
12
13 date-main: $(OBJ)
14 ⟶$(CXX) -o $@ $^ $(LDFLAGS)
15
16 $(OBJ): date.hpp  # tous les fichier .o dépendent de date.hpp
17
18 check: date-main
19 ⟶./date-main -e
20
21 clean:
22 ⟶rm -f $(OBJ) $(EXEC_FILES)
```

Pour plus d'informations

Il y a des dizaines d'introductions plus ou moins poussées à Make sur le Web. Voir par exemple : `https://gl.developpez.com/tutoriel/outil/makefile/`

Le manuel de référence de `GNU make` : `https://www.gnu.org/software/make/manual/html_node/index.html`