Compilation séparée

Ivan Noyer

Objectif et principe

- Plutôt que de mettre tout le code dans un seul fichier, on le distribue dans plusieurs endroits : des *modules*.

Objectif et principe

- Plutôt que de mettre tout le code dans un seul fichier, on le distribue dans plusieurs endroits : des *modules*.
- Dans un fichier donné, on met par exemple des outils de calculs mathématiques, dans un autre des fonctions de manipulation de chaînes de caractères etc.

Objectif et principe

- Plutôt que de mettre tout le code dans un seul fichier, on le distribue dans plusieurs endroits : des *modules*.
- Dans un fichier donné, on met par exemple des outils de calculs mathématiques, dans un autre des fonctions de manipulation de chaînes de caractères etc.
- Cela permet de bien organiser un projet, d'améliorer la lisibilité du code et de faciliter la *maintenance* : en cas de changement de quelques lignes de code seulement, on n'a pas besoin de tout recompiler mais seulement les fichiers concernés.

Crédits

- Ce cours d' Anne Canteault sur la compilation séparée.

Crédits

- Ce cours d' Anne Canteault sur la compilation séparée.
- Ce chapitre sur les directives préprocesseur d'un cours de OpenClassRoom

Crédits

- Ce cours d' Anne Canteault sur la compilation séparée.
- Ce chapitre sur les directives préprocesseur d'un cours de OpenClassRoom
- Ce tutoriel sur les Makefile

La directive `include`

- La syntaxe pour inclure l'interface d'un fichier de bibliothèque est de placer son nom entre chevrons comme dans `<stdio.h>`.

La directive `include`

- La syntaxe pour inclure l'interface d'un fichier de bibliothèque est de placer son nom entre chevrons comme dans `<stdio.h>`.
- Pour tout autre fichier, et en particulier pour NOS FICHIERS PERSONNELS, on écrit son nom entre guillements ; par exemple `#include "produit.h"`.

La directive `include`

- La syntaxe pour inclure l'interface d'un fichier de bibliothèque est de placer son nom entre chevrons comme dans `<stdio.h>`.
- Pour tout autre fichier, et en particulier pour NOS FICHIERS PERSONNELS, on écrit son nom entre guillements ; par exemple `#include "produit.h"`.
- Dans tous les cas, le préprocesseur insère le contenu du fichier cible à la place du `# include nomDuFichier`

Rappels sur la directive `define`

- Lorsqu'il lit `# define TOTO Tata`, le préprocesseur remplace toute occurrence du premier mot par le second (mais pas dans les commentaires ni les expressions entre guillemets).

Rappels sur la directive `define`

- Lorsqu'il lit `# define TOTO Tata`, le préprocesseur remplace toute occurrence du premier mot par le second (mais pas dans les commentaires ni les expressions entre guillemets).
- On peut effectuer des calculs arithmétiques en utilisant des constantes introduites par cette directive.

```
1 # define LARGEUR   60
2 # define LONGUEUR   80
3 # define AIRE   (LARGEUR * LONGUEUR)
4
```

Rappels sur la directive `define`

Hors programme.
On peut se servir de `# define` pour définir des *macros* sans paramètre :

```
1 # define HELLO()   printf("Salut !\n");\
2                    printf("Hi !\n");\
3                    printf("Hallo !\n");
4
5 void main(){
6   HELLO(); }
7
```

Observer les passages à la ligne avec un antislash « \ » comme en Python
Après compilation et exécution :

```
Salut !
Hi !
Hallo !
```

Rappels sur la directive `define`

Hors programme.

On peut se servir de `# define` pour définir des *macros* Avec paramètres :

```
1  #include <stdio.h>
2  #define APPRECIATION(nom, note)\
3          if (note<10)\
4            printf("%s, vous n\'avez pas la moyenne.\n",nom);\
5          else\
6          printf("%s, vous êtes reçu !\n",nom);
7
8  void main(){
9    APPRECIATION ("Dupont",8);
10   APPRECIATION ("Durand",18); }
```

Après compilation et exécution :

```
Dupont, vous n'avez pas la moyenne.
Durand, vous êtes reçu !
```

Définitions sans valeurs de substitutions ♡

- Parfois, on écrit juste :

```
1 #define NOM_DU_FICHIER
```

Définitions sans valeurs de substitutions ♡

- Parfois, on écrit juste :

```
1 #define NOM_DU_FICHIER
```

- Cette déclaration, combinée avec l'usage des *conditions de préprocesseur* (comme `# ifndef`) permet d'éviter les inclusions infinie (lorsqu'un fichier `A` importe un fichier `B` qui importe `A`).

Un projet en deux fichiers

- Dans un 1er fichier `main.c` :

```
1  #include <stdio.h>
2  #include "produit.h" //observer la syntaxe
3
4  int main(void)
5  {
6    int a, b, c;
7    scanf("%d",&a);    scanf("%d",&b);
8    printf("\nle produit vaut %d\n", produit(a,b));
9    return 0; }
10
```

Un projet en deux fichiers

- Dans un 1er fichier `main.c` :

```c
#include <stdio.h>
#include "produit.h" //observer la syntaxe

int main(void)
{
  int a, b, c;
  scanf("%d",&a);    scanf("%d",&b);
  printf("\nle produit vaut %d\n", produit(a,b));
  return 0; }

```

- Dans un 2nd fichier `produit.h` (qui n'est pas vraiment un fichier d'en-tête).

```c
int produit(int a, int b)
{
  return(a * b); }

```

Compilation en une fois

- On peut compiler ce projet à 2 fichiers en une seule fois avec la commande `gcc main.c -o main`

Compilation en une fois

- On peut compiler ce projet à 2 fichiers en une seule fois avec la commande `gcc main.c -o main`
- Le préprocesseur va juste regrouper (en les concaténant et en supprimant les commentaires) les deux fichiers en un seul et produira l'exécutable à partir de ce fichier concaténé.

Compilation en une fois

- On peut compiler ce projet à 2 fichiers en une seule fois avec la commande `gcc main.c -o main`
- Le préprocesseur va juste regrouper (en les concaténant et en supprimant les commentaires) les deux fichiers en un seul et produira l'exécutable à partir de ce fichier concaténé.
- Le seul avantage de cette démarche est de rendre le code plus lisible mais on n'a rien gagné en terme de maintenance du projet.

Introduction d'un fichier d'en-tête

- Séparons les prototypes des corps des fonctions de `produit.h`.

Introduction d'un fichier d'en-tête

- Séparons les prototypes des corps des fonctions de `produit.h`.
- On crée pour cela un fichier d'interface `produit.h` contenant les prototypes et un fichier `produit.c` contenant les codes.

Introduction d'un fichier d'en-tête

- Séparons les prototypes des corps des fonctions de `produit.h`.
- On crée pour cela un fichier d'interface `produit.h` contenant les prototypes et un fichier `produit.c` contenant les codes.
- Dans le fichier `produit.c`, incluons l'en-tête :

```
1 # include "produit.h"
2 int produit(int a, int b){ return(a * b); }
3
```

Introduction d'un fichier d'en-tête

- Séparons les prototypes des corps des fonctions de `produit.h`.
- On crée pour cela un fichier d'interface `produit.h` contenant les prototypes et un fichier `produit.c` contenant les codes.
- Dans le fichier `produit.c`, incluons l'en-tête :

```
1 # include "produit.h"
2 int produit(int a, int b){ return(a * b); }
3
```

- Toutes les fonctions de `produit.c` destinées à être appelées par d'autres fichiers que `produit.c` lui-même devraient avoir leur prototype listé dans `produit.h`.

Introduction d'un fichier d'en-tête

- Séparons les prototypes des corps des fonctions de `produit.h`.
- On crée pour cela un fichier d'interface `produit.h` contenant les prototypes et un fichier `produit.c` contenant les codes.
- Dans le fichier `produit.c`, incluons l'en-tête :

```
1 # include "produit.h"
2 int produit(int a, int b){ return(a * b); }
3
```

- Toutes les fonctions de `produit.c` destinées à être appelées par d'autres fichiers que `produit.c` lui-même devraient avoir leur prototype listé dans `produit.h`.
- Comme de plus les interfaces listées dans `produit.h` correspondent à des fonctions dont le code est écrit ailleurs, on peut l'indiquer explicitement par le mot clé `extern`.

Introduction d'un fichier d'en-tête

- Séparons les prototypes des corps des fonctions de `produit.h`.
- On crée pour cela un fichier d'interface `produit.h` contenant les prototypes et un fichier `produit.c` contenant les codes.
- Dans le fichier `produit.c`, incluons l'en-tête :

```
1 # include "produit.h"
2 int produit(int a, int b){ return(a * b); }
3
```

- Toutes les fonctions de `produit.c` destinées à être appelées par d'autres fichiers que `produit.c` lui-même devraient avoir leur prototype listé dans `produit.h`.
- Comme de plus les interfaces listées dans `produit.h` correspondent à des fonctions dont le code est écrit ailleurs, on peut l'indiquer explicitement par le mot clé `extern`.
- Le fichier d'en-tête `produit.h` contient donc :

```
1 extern int produit(int, int);
2
```

Mot clé `extern` pour les fonctions

- Pour une fonction les deux déclarations suivantes sont équivalentes `extern int fct(char c, float x)` et `int fct(char c, float x)`.

Mot clé `extern` pour les fonctions

- Pour une fonction les deux déclarations suivantes sont équivalentes `extern int fct(char c, float x)` et `int fct(char c, float x)`.
- La fonction sera utilisable dans les 2 cas à l'extérieur du fichier source. Son nom devient ce qu'on appelle un *identificateur externe*, c'est à dire qu'il est accessible à l'éditeur de lien.

Mot clé `extern` pour les fonctions

- Pour une fonction les deux déclarations suivantes sont équivalentes `extern int fct(char c, float x)` et `int fct(char c, float x)`.
- La fonction sera utilisable dans les 2 cas à l'extérieur du fichier source. Son nom devient ce qu'on appelle un *identificateur externe*, c'est à dire qu'il est accessible à l'éditeur de lien.
- Le mot `static` empêche que l'identificateur de la fonction soit utilisé à l'extérieur du fichier source où elle est définie. On parle alors de fonction *cachée* ou *privée*.
  Exemple :

```
static int fct(char c, float x)...
```

Mot clé `extern` pour les variables

- Le mot clé `extern` devrait être réservé aux *redéclaration*

Mot clé `extern` pour les variables

- Le mot clé `extern` devrait être réservé aux *redéclaration*
- Une déclaration contenant une initialisation constitue toujours une définition.
  `int x = 2;` et `extern int x = 2;` sont équivalents. `extern` ne sert à rien.

Mot clé `extern` pour les variables

- Le mot clé `extern` devrait être réservé aux *redéclaration*
- Une déclaration contenant une initialisation constitue toujours une définition.
  `int x = 2;` et `extern int x = 2;` sont équivalents. `extern` ne sert à rien.
- Une déclaration avec `extern` sans initialisation constitue toujours une redéclaration (donc jamais une définition) d'une variable pouvant , éventuellement, être définie ailleurs (donc dans le même fichier source ou un autre).
  `extern int a;` fait référence à une variable `a` définie ailleurs.

Mot clé `extern` pour les variables

- Le mot clé `extern` devrait être réservé aux *redéclaration*
- Une déclaration contenant une initialisation constitue toujours une définition.
  `int x = 2;` et `extern int x = 2;` sont équivalents. `extern` ne sert à rien.
- Une déclaration avec `extern` sans initialisation constitue toujours une redéclaration (donc jamais une définition) d'une variable pouvant , éventuellement, être définie ailleurs (donc dans le même fichier source ou un autre).
  `extern int a;` fait référence à une variable `a` définie ailleurs.
- On retient la règle suivante de bonne pratique :

Mot clé `extern` pour les variables

- Le mot clé `extern` devrait être réservé aux *redéclaration*
- Une déclaration contenant une initialisation constitue toujours une définition.
  `int x = 2;` et `extern int x = 2;` sont équivalents. `extern` ne sert à rien.
- Une déclaration avec `extern` sans initialisation constitue toujours une redéclaration (donc jamais une définition) d'une variable pouvant , éventuellement, être définie ailleurs (donc dans le même fichier source ou un autre).
  `extern int a;` fait référence à une variable `a` définie ailleurs.
- On retient la règle suivante de bonne pratique :
  Dans les définitions on n'utilise jamais `extern` et on s'interdit les déclarations multiples au sein d'un même fichier. On peut ou non initialiser.

Mot clé `extern` pour les variables

- Le mot clé `extern` devrait être réservé aux *redéclaration*
- Une déclaration contenant une initialisation constitue toujours une définition.
  `int x = 2;` et `extern int x = 2;` sont équivalents. `extern` ne sert à rien.
- Une déclaration avec `extern` sans initialisation constitue toujours une redéclaration (donc jamais une définition) d'une variable pouvant , éventuellement, être définie ailleurs (donc dans le même fichier source ou un autre).
  `extern int a;` fait référence à une variable `a` définie ailleurs.
- On retient la règle suivante de bonne pratique :
  Dans les définitions on n'utilise jamais `extern` et on s'interdit les déclarations multiples au sein d'un même fichier. On peut ou non initialiser.
  Dans les redéclarations On utilise systématiquement `extern` et on s'abstient de toute initialisation.

Introduction d'un fichier d'en-tête

Compilation séparée

- Fabriquons le fichier objet relatif au `main.c` :

```
gcc -c main.c
```

Introduction d'un fichier d'en-tête

Compilation séparée

- Fabriquons le fichier objet relatif au `main.c` :

```
gcc -c main.c
```

- Fabriquons le fichier objet relatif au `produit.c` :

```
gcc -c produit.c
```

Introduction d'un fichier d'en-tête

Compilation séparée

- Fabriquons le fichier objet relatif au `main.c` :

```
gcc -c main.c
```

- Fabriquons le fichier objet relatif au `produit.c` :

```
gcc -c produit.c
```

- Compilons l'ensemble

```
gcc main.o produit.o -o prod
```

Introduction d'un fichier d'en-tête

Compilation séparée

- Fabriquons le fichier objet relatif au `main.c` :

```
gcc -c main.c
```

- Fabriquons le fichier objet relatif au `produit.c` :

```
gcc -c produit.c
```

- Compilons l'ensemble

```
gcc main.o produit.o -o prod
```

- Remarquons qu'on peut toujours compiler tout en même temps :

```
gcc produit.c main.c -o prod
```

Importation unique

- Comme on le constate, le fichier d'en-tête est importé deux fois :

Importation unique

- Comme on le constate, le fichier d'en-tête est importé deux fois :
  - une première fois par `produit.c`,

Importation unique

- Comme on le constate, le fichier d'en-tête est importé deux fois :
  - une première fois par `produit.c`,
  - une seconde fois par `main.c`

Importation unique

- Comme on le constate, le fichier d'en-tête est importé deux fois :
  - une première fois par `produit.c`,
  - une seconde fois par `main.c`
- Dans le fichier exécutable généré par `gcc`, l'inclusion apparaît donc deux fois.

Importation unique

- Comme on le constate, le fichier d'en-tête est importé deux fois :
  - une première fois par `produit.c`,
  - une seconde fois par `main.c`
- Dans le fichier exécutable généré par `gcc`, l'inclusion apparaît donc deux fois.
- Pour éviter cela, on crée une constante `PRODUIT_H`.

Importation unique

- Comme on le constate, le fichier d'en-tête est importé deux fois :
  - une première fois par `produit.c`,
  - une seconde fois par `main.c`
- Dans le fichier exécutable généré par `gcc`, l'inclusion apparaît donc deux fois.
- Pour éviter cela, on crée une constante `PRODUIT_H`.
- Cette constante, partagée par tout le projet, est définie la première fois qu'on inclue le fichier `produit.h`.

Importation unique

- Comme on le constate, le fichier d'en-tête est importé deux fois :
  - une première fois par `produit.c`,
  - une seconde fois par `main.c`
- Dans le fichier exécutable généré par `gcc`, l'inclusion apparaît donc deux fois.
- Pour éviter cela, on crée une constante `PRODUIT_H`.
- Cette constante, partagée par tout le projet, est définie la première fois qu'on inclue le fichier `produit.h`.
- La seconde fois qu'on importe le fichier d'en-tête, la constante est déjà définie et on se débrouille pour empêcher une nouvelle inclusion des prototypes de `produit.c`.

Importation unique

- Comme on le constate, le fichier d'en-tête est importé deux fois :
  - une première fois par `produit.c`,
  - une seconde fois par `main.c`
- Dans le fichier exécutable généré par `gcc`, l'inclusion apparaît donc deux fois.
- Pour éviter cela, on crée une constante `PRODUIT_H`.
- Cette constante, partagée par tout le projet, est définie la première fois qu'on inclue le fichier `produit.h`.
- La seconde fois qu'on importe le fichier d'en-tête, la constante est déjà définie et on se débrouille pour empêcher une nouvelle inclusion des prototypes de `produit.c`.
- Il y a une convention de nommage pour cette constante : nom du fichier d'en-tête mis en majuscules (mais sans l'extension `.h`) ; le tout suivi de `_H`.

ifndef

- L'idée est de mettre tous les prototypes et déclarations diverses de `produit.h` dans la branche positive d'une instruction conditionnelle du préprocesseur.

ifndef

- L'idée est de mettre tous les prototypes et déclarations diverses de `produit.h` dans la branche positive d'une instruction conditionnelle du préprocesseur.
- Si la condition est vérifiée, alors ces prototypes et déclarations sont bien ajoutés au fichier généré. Sinon, ils sont ignorés.

ifndef

- L'idée est de mettre tous les prototypes et déclarations diverses de `produit.h` dans la branche positive d'une instruction conditionnelle du préprocesseur.
- Si la condition est vérifiée, alors ces prototypes et déclarations sont bien ajoutés au fichier généré. Sinon, ils sont ignorés.
- Le fichier `produit.h` devient alors :

```
1  #ifndef PRODUIT_H // si PRODUIT_H non déjà déclarée
2
3  #define PRODUIT_H //... alors on la déclare !...
4
5  // Et on liste les prototypes à inclure dans les autres
       fichiers :
6
7  int produit(int, int);
8
9  #endif // FIN instruction conditionnelle
10
```

`ifndef`

- Dans le fichier `produit.h` on a ajouté l'instruction de préprocessing `#ifndef PRODUIT_H` dont l'esprit est de signifier ≪ si la constante PRODUIT_H n'a pas déjà été déclarée alors voici les prototypes à inclure : ... ≫.

ifndef

- Dans le fichier `produit.h` on a ajouté l'instruction de préprocessing `#ifndef PRODUIT_H` dont l'esprit est de signifier ≪ si la constante PRODUIT_H n'a pas déjà été déclarée alors voici les prototypes à inclure : ... ≫.
- et on compile en 3 temps :

```
gcc -c main.c
gcc -c produit.c
gcc main.o produit.o -o prod
```

Présentation

- Il ne rentre pas dans les compétences exigibles des étudiants de CPGE qu'ils sachent écrire un `Makefile`.
  C'est toutefois un outil très pratique pour automatiser la compilation d'un projet et optimiser le temps passé pour cette opération.

Présentation

- Il ne rentre pas dans les compétences exigibles des étudiants de CPGE qu'ils sachent écrire un `Makefile`.
  C'est toutefois un outil très pratique pour automatiser la compilation d'un projet et optimiser le temps passé pour cette opération.
- On ne donne ici que quelques rudiments de cette technique qui nécessiterait un manuel à part entière. Consulter par exemple developpez.com pour plus d'informations.

Syntaxe

Un Makefile est constitué d'un ensemble de règles de la forme :

```
cible : dependance
    commandes
```

Utilisation

Supposons qu'on ait créé le fichier `Makefile` dans le répertoire courant. Il y a deux possibilités d'appel :

Exécution de la première règle rencontrée Cela s'obtient très facilement en tapant dans un terminal :

```
$ make
```

Utilisation

Supposons qu'on ait créé le fichier `Makefile` dans le répertoire courant. Il y a deux possibilités d'appel :

Exécution de la première règle rencontrée Cela s'obtient très facilement en tapant dans un terminal :

```
$ make
```

Exécution d'une règle spécifique Il suffit de saisir dans un terminal.

```
$ make nomDeLaRegle
```

Évaluation des règles

L'évaluation d'une règle se fait en plusieurs étapes :

Analyse des dépendances : si une dépendance est la cible d'une autre règle du Makefile, cette règle est préalablement évaluée.

Évaluation des règles

L'évaluation d'une règle se fait en plusieurs étapes :

Analyse des dépendances : si une dépendance est la cible d'une autre règle du Makefile, cette règle est préalablement évaluée.

Exécution des commandes : Après analyse des dépendances, si

Évaluation des règles

L'évaluation d'une règle se fait en plusieurs étapes :

Analyse des dépendances : si une dépendance est la cible d'une autre règle du Makefile, cette règle est préalablement évaluée.

Exécution des commandes : Après analyse des dépendances, si

1. la cible ne correspond pas à un fichier existant

Évaluation des règles

L'évaluation d'une règle se fait en plusieurs étapes :

Analyse des dépendances : si une dépendance est la cible d'une autre règle du Makefile, cette règle est préalablement évaluée.

Exécution des commandes : Après analyse des dépendances, si

1. la cible ne correspond pas à un fichier existant
2. ou si un fichier dépendance est plus récent que la règle, les différentes commandes sont exécutées.

Un Makefile minimum

```
prod : produit.o main.o
	gcc -o prod produit.o main.o

produit.o : produit.c
	#compilation sans lien, avec messages d'erreur
	gcc -o produit.o -c produit.c -Wall -O3

main.o : main.c
	#compilation sans lien, avec messages d'erreur
	gcc -o main.o -c main.c -Wall -O3
```

- La commande `make` lancée dans un terminal depuis le répertoire courant cherche à réaliser la première règle rencontrée : `prod`.

Un Makefile minimum

```
prod : produit.o main.o
	gcc -o prod produit.o main.o

produit.o : produit.c
	#compilation sans lien, avec messages d'erreur
	gcc -o produit.o -c produit.c -Wall -O3

main.o : main.c
	#compilation sans lien, avec messages d'erreur
	gcc -o main.o -c main.c -Wall -O3
```

- La commande `make` lancée dans un terminal depuis le répertoire courant cherche à réaliser la première règle rencontrée : `prod`.
- Il y a deux dépendances `produit.o` et `main.o` qui sont d'ailleurs des règles.

Dépendances de la règle `prod`

- On commence par exécuter la règle `produit.o` (1ere dépendance de la règle `prod`). L'unique dépendance de cette règle est un fichier `produit.c` qui n'est pas une règle. Le Makefile exécute la commande de cette règle `produit.o` si une des conditions suivantes est réalisée :

Dépendances de la règle `prod`

- On commence par exécuter la règle `produit.o` (1ere dépendance de la règle `prod`). L'unique dépendance de cette règle est un fichier `produit.c` qui n'est pas une règle. Le Makefile exécute la commande de cette règle `produit.o` si une des conditions suivantes est réalisée :
  - le fichier `produit.o` n'existe pas dans le répertoire courant,

Dépendances de la règle prod

- On commence par exécuter la règle `produit.o` (1ere dépendance de la règle `prod`). L'unique dépendance de cette règle est un fichier `produit.c` qui n'est pas une règle. Le Makefile exécute la commande de cette règle `produit.o` si une des conditions suivantes est réalisée :
  - le fichier `produit.o` n'existe pas dans le répertoire courant,
  - ou bien le fichier `produit.c` est plus récent que `produit.o`. Cela signifie que `produit.c` a été modifié depuis la dernière création du module objet `produit.o`. Il faut donc reconstruire ce dernier.

Dépendances de la règle prod

- On commence par exécuter la règle `produit.o` (1ere dépendance de la règle `prod`). L'unique dépendance de cette règle est un fichier `produit.c` qui n'est pas une règle. Le Makefile exécute la commande de cette règle `produit.o` si une des conditions suivantes est réalisée :
  - le fichier `produit.o` n'existe pas dans le répertoire courant,
  - ou bien le fichier `produit.c` est plus récent que `produit.o`. Cela signifie que `produit.c` a été modifié depuis la dernière création du module objet `produit.o`. Il faut donc reconstruire ce dernier.
- Si une des deux conditions ci-dessus est vérifiée, on exécute la commande `gcc -o produit.o -c produit.c -Wall -O3` (compilation sans édition de lien `-c`, avec Warnings `-Wall` et optimisation complète `-O3`).

Dépendances de la règle prod

- On commence par exécuter la règle `produit.o` (1ere dépendance de la règle `prod`). L'unique dépendance de cette règle est un fichier `produit.c` qui n'est pas une règle. Le Makefile exécute la commande de cette règle `produit.o` si une des conditions suivantes est réalisée :
  - le fichier `produit.o` n'existe pas dans le répertoire courant,
  - ou bien le fichier `produit.c` est plus récent que `produit.o`. Cela signifie que `produit.c` a été modifié depuis la dernière création du module objet `produit.o`. Il faut donc reconstruire ce dernier.
- Si une des deux conditions ci-dessus est vérifiée, on exécute la commande `gcc -o produit.o -c produit.c -Wall -O3` (compilation sans édition de lien `-c`, avec Warnings `-Wall` et optimisation complète `-O3`).
- Une fois cette règle exécutée, on lance la règle `main.o` qui fonctionne sur le même principe.

Exécution de la première règle

- Toutes les dépendances de `prod` étant construites, on exécute la commande associée à la règle `prod` si :

Exécution de la première règle

- Toutes les dépendances de `prod` étant construites, on exécute la commande associée à la règle `prod` si :
  - l'un des fichiers objet `.o` n'existe pas dans le répertoire courant,

Exécution de la première règle

- Toutes les dépendances de `prod` étant construites, on exécute la commande associée à la règle `prod` si :
  - l'un des fichiers objet `.o` n'existe pas dans le répertoire courant,
  - ou l'un des fichiers objets `.o` est plus récent que `prod`.

Exécution de la première règle

- Toutes les dépendances de `prod` étant construites, on exécute la commande associée à la règle `prod` si :
  - l'un des fichiers objet `.o` n'existe pas dans le répertoire courant,
  - ou l'un des fichiers objets `.o` est plus récent que `prod`.
- Dans ce cas, c'est la commande `gcc -o prod produit.o main.o` qui est lancée

Nettoyage et régénération

Avec le Makefile précédent, on ne peut pas créer plusieurs exécutables, ni supprimer les fichiers intermédiaires `.o` (ils restent sur le disque dur) ni forcer la régénération complète du projet.

- En effet, les fichiers objets `produit.o` et `main.o` sont restés sur le disque dur. Lorsqu'on relance la commande `make`, on obtient le message suivant :

```
make: << prod >> est à jour.
```

Nettoyage et régénération

On rajoute alors trois règles qui portent, par convention, les noms `all`, `clean`, `mrproper` :

`all` : on la fait suivre de la ou des règles de construction d'exécutables ; pour nous c'est seulement `prod`.

Nettoyage et régénération

On rajoute alors trois règles qui portent, par convention, les noms `all`, `clean`, `mrproper` :

`all` : on la fait suivre de la ou des règles de construction d'exécutables ; pour nous c'est seulement `prod`.

`clean` : on décide ici de lui faire supprimer tous les fichiers objets (mais ce pourrait être d'autres types de fchiers).

Nettoyage et régénération

On rajoute alors trois règles qui portent, par convention, les noms `all`, `clean`, `mrproper` :

`all` : on la fait suivre de la ou des règles de construction d'exécutables ; pour nous c'est seulement `prod`.

`clean` : on décide ici de lui faire supprimer tous les fichiers objets (mais ce pourrait être d'autres types de fchiers).

`mrproper` : un nettoyage complet. Le Makefile engendre un fichier exécutable `prod` et des fichiers objets. On déclare `clean` comme dépendance, ce qui a pour effet de supprimer les fichiers objets. Puis il reste à supprimer le fichier exécutable `prod`.

Makefile plus complet

```
all: prod

prod: produit.o main.o
	gcc -o prod produit.o main.o

produit.o: produit.c
	gcc -o produit.o -c produit.c -W -O3

main.o: main.c
	gcc -o main.o -c main.c -W -O3

clean:
	rm -rf *.o

mrproper: clean
	rm prod
```

Makefile plus complet

```
all: prod

prod: produit.o main.o
	gcc -o prod produit.o main.o

produit.o: produit.c
	gcc -o produit.o -c produit.c -W -O3

main.o: main.c
	gcc -o main.o -c main.c -W -O3

clean:
	rm -rf *.o

mrproper: clean
	rm prod
```