

Language C

Matej Stehlik

matej@irif.fr

Avec les slides fait par Iyan Nazarian

iyan.nazarian@etu.u-paris.fr

Headers, Makefile, Compilation made simple

Le but

- Ce cours a pour but de vous familiariser avec le concept de **headers** en C et comment les manipuler

Le but

- Ce cours a pour but de vous familiariser avec le concept de **headers** en C et comment les manipuler
- Visiter le concept de build system, notamment **make** et le fameux **Makefile**, et pourquoi c'est utile

Le but

- Ce cours a pour but de vous familiariser avec le concept de **headers** en C et comment les manipuler
- Visiter le concept de build system, notamment **make** et le fameux **Makefile**, et pourquoi c'est utile
- Observer les différentes configurations possibles en fonction des use-cases.

Compilation made simple

Résumé:

- 1) Les headers
- 2) Makefile
- 3) Exemple de petit projet

Les headers

- `stdio.h`, `stdlib.h`, `unistd.h`, ...

Les headers

- `stdio.h`, `stdlib.h`, `unistd.h`, ...
- Contiennent uniquement des **macros** et des **définitions**

Les headers

- `stdio.h`, `stdlib.h`, `unistd.h`, ...
- Contiennent uniquement des **macros** et des **définitions**
- Si je « `#include <stdio.h>` », lui même `#include` d'autres fichiers `.h`, avec l'un d'eux contenant :

```
int printf(const char *restrict format, ...);
```


Les headers

- `stdio.h`, `stdlib.h`, `unistd.h`, ...
- Contiennent uniquement des **macros** et des **définitions**
- Si je « `#include <stdio.h>` », lui même `#include` d'autres fichiers `.h`, avec l'un d'eux contenant :

```
int printf(const char *restrict format, ...);
```
- Et quand j'appelle `printf` dans mon fichier `.c`, je suis en train de dire au compilateur : "ne t'inquiète pas, au moment du linking, tu vas trouver l'implémentation de cette fonction"

Les headers

- `stdio.h`, `stdlib.h`, `unistd.h`, ...
- Contiennent uniquement des **macros** et des **définitions**
- Si je « `#include <stdio.h>` », lui même `#include` d'autres fichiers `.h`, avec l'un d'eux contenant :

```
int printf(const char *restrict format, ...);
```
- Et quand j'appelle `printf` dans mon fichier `.c`, je suis en train de dire au compilateur : "ne t'inquiète pas, au moment du linking, tu vas trouver l'implémentation de cette fonction"
- Et c'est bien le cas ! Grâce à la **libc**

La libc

- La **libc** contient quasiment toutes les **implémentations** des fonctions **définies** dans les headers comme `<stdio.h>`, `<stdlib.h>`, ...

La libc

- La **libc** contient quasiment toutes les **implémentations** des fonctions **définies** dans les headers comme `<stdio.h>`, `<stdlib.h>`, ...
- Celle-ci est considérée comme faisant partie du système d'exploitation des **Unix-like OSs**.

La libc

- La **libc** contient quasiment toutes les **implémentations** des fonctions **définies** dans les headers comme `<stdio.h>`, `<stdlib.h>`, ...
- Celle-ci est considérée comme faisant partie du système d'exploitation des **Unix-like OSs**.
- Certains Systèmes ont leur propre version de la libc, comme **BSD libc**, pour les systèmes de la famille **BSD**, ou la **glibc**, pour la plupart des systèmes Linux par exemple.

La libc

- La **libc** contient quasiment toutes les **implémentations** des fonctions **définies** dans les headers comme `<stdio.h>`, `<stdlib.h>`, ...
- Celle-ci est considérée comme faisant partie du système d'exploitation des **Unix-like OSs**.
- Certains Systèmes ont leur propre version de la libc, comme **BSD libc**, pour les systèmes de la famille **BSD**, ou la **glibc**, pour la plupart des systèmes Linux par exemple.
- "Mais comment le compilateur sait où se trouvent les fichiers `.h` et leurs implémentations, si on ne spécifie aucune option ou quelque chose du genre quand on fait le linking ?"

Comment gcc le sait-il ?

- Pour les définitions, c'est très simple, c'est grâce au `#include <stdio.h>`, plus précisément c'est grâce aux "<" et ">".

Comment gcc le sait-il ?

- Pour les définitions, c'est très simple, c'est grâce au `#include <stdio.h>`, plus précisément c'est grâce aux "<" et ">".
- La syntaxe <stdio.h> informe le compilateur que le fichier stdio.h se trouve dans les **system directories where standard headers are located**, généralement sur Linux c'est /usr/include/

Comment gcc le sait-il ?

- Pour les définitions, c'est très simple, c'est grâce au `#include <stdio.h>`, plus précisément c'est grâce aux "<" et ">".
- La syntaxe <stdio.h> informe le compilateur que le fichier stdio.h se trouve dans les **system directories where standard headers are located**, généralement sur Linux c'est /usr/include/
- Pour les implémentations, c'est au moment du Linking, le linker va automatiquement, à moins que le contraire soit précisé, linker la version de la libc présente sur le système donné.

Comment gcc le sait-il ?

- Pour les définitions, c'est très simple, c'est grâce au `#include <stdio.h>`, plus précisément c'est grâce aux "<" et ">".
- La syntaxe <stdio.h> informe le compilateur que le fichier stdio.h se trouve dans les **system directories where standard headers are located**, généralement sur Linux c'est /usr/include/
- Pour les implémentations, c'est au moment du Linking, le linker va automatiquement, à moins que le contraire soit précisé, linker la version de la libc présente sur le système donné.
- Et donc c'est pour cela qu'on peut utiliser les fonctions définies dans stdio.h sans manuellement dire où se trouve stdio.h, et sans dire au compilateur où se trouve l'implémentation des fonctions concernées.

Les headers

- Voilà donc comment cela fonctionne pour les **standard headers**, les headers qui contiennent les définitions des fonctions de la libc

Les headers

- Voilà donc comment cela fonctionne pour les **standard headers**, les headers qui contiennent les définitions des fonctions de la libc
- Comment faire pour nos propres headers ?

Our very own header

- Imaginons que nous voulons créer notre propre header, dans le cadre d'un petit projet, généralement cela va sans dire.

Our very own header

- Imaginons que nous voulons créer notre propre header, dans le cadre d'un petit projet, généralement cela va sans dire.
- Nous aurons une structure simple :

Our very own header

- Imaginons que nous voulons créer notre propre header, dans le cadre d'un petit projet, généralement cela va sans dire.
- Nous aurons une structure simple :

- ```
.
├── README.md <- pour des infos
├── src/
│ ├── les fichiers .c
└── includes/
 ├── les fichiers .h
```

# Our very own header

- Imaginons que nous voulons créer notre propre header, dans le cadre d'un petit projet, généralement cela va sans dire.
- Nous aurons une structure simple :

- ```
.  
├── README.md <- pour des infos  
├── src/  
│   ├── les fichiers .c  
└── includes/  
    ├── les fichiers .h
```

- Voyons comment gérer tout ça

On va continuer sur l'exemple du cours précédent, donc petit rappel :

On va continuer sur l'exemple du cours précédent, donc petit rappel :

- Un fichier *pretty_printing.c* qui contient des fonctions pour afficher des array

On va continuer sur l'exemple du cours précédent, donc petit rappel :

- Un fichier *pretty_printing.c* qui contient des fonctions pour afficher des array
- Un fichier *main.c* qui contient des exemples avec ces fonctions

On va continuer sur l'exemple du cours précédent, donc petit rappel :

- Un fichier *pretty_printing.c* qui contient des fonctions pour afficher des array
- Un fichier *main.c* qui contient des exemples avec ces fonctions

On avait rencontré le problème d'utiliser ces fonctions la dans *main.c*, et cela a été résolu en ajoutant les **définitions** des fonctions qu'on veut utiliser au début de *main.c* et en linkant les 2 fichiers **.o**, ou simplement en les compilant ensemble.

On va continuer sur l'exemple du cours précédent, donc petit rappel :

- Un fichier *pretty_printing.c* qui contient des fonctions pour afficher des array
- Un fichier *main.c* qui contient des exemples avec ces fonctions

On avait rencontré le problème d'utiliser ces fonctions la dans *main.c*, et cela a été résolu en ajoutant les **définitions** des fonctions qu'on veut utiliser au début de *main.c* et en linkant les 2 fichiers **.o**, ou simplement en les compilant ensemble.

Passons de cette structure à notre structure désirée.

```
~/Desktop/Universite/demo main*
```

```
> tree
```

```
.  
├── main.c  
└── pretty_printing.c
```

```
~/Desktop/Universite/demo main*
```

```
> tree
```

```
.  
├── main.c  
└── pretty_printing.c
```

```
~/Desktop/Universite/demo main*
```

```
> gcc main.c pretty_printing.c
```

```
~/Desktop/Universite/demo main*
```

```
> ls
```


```
a.out  main.c  pretty_printing.c
```

Ajoutons notre header

- Nous aimerions ajouter un header afin de ne pas écrire les définitions à chaque fois que nous voulons utiliser une fonction définie dans un autre fichier (**Translation Unit** à la compilation)


Ajoutons notre header

- Nous aimerions ajouter un header afin de ne pas écrire les définitions à chaque fois que nous voulons utiliser une fonction définie dans un autre fichier (**Translation Unit** à la compilation)
- Voilà à quoi cela peut ressembler :

 pretty_printing.c


```
1  #include "pretty_printing.h"
2
3  void pretty_printing_tab(int *tab, size_t size)
4  {
5      size_t i;
6
7      i = 0;
8      printf("[");
9      while (i < size)
10     {
11         printf("%d, ", tab[i]);
12         i++;
13     }
14     printf("]\n");
15 }
```



 main.c

```
1 #include "pretty_printing.h"
2
3 int main(void)
4 {
5     int tab[3] = {1, 123, 1231312};
6     pretty_printing_tab(tab, sizeof(tab) / sizeof(tab[0]));
7
8 }
```



 pretty_printing.c

```
1 #include "pretty_printing.h"
2
3 void pretty_printing_tab(int *tab, size_t size)
4 {
5     size_t i;
6
7     i = 0;
8     printf("[");
9     while (i < size)
10     {
11         printf("%d, ", tab[i]);
12         i++;
13     }
14     printf("]\n");
15 }
```

main.c

```
1 #include "pretty_printing.h"
2
3 int main(void)
4 {
5     int tab[3] = {1, 123, 1231312};
6     pretty_printing_tab(tab, sizeof(tab) / sizeof(tab[0]));
7
8 }
```


pretty_printing.c

```
1 #include "pretty_printing.h"
2
3 void pretty_printing_tab(int *tab, size_t size)
4 {
5     size_t i;
6
7     i = 0;
8     printf("[");
9     while (i < size)
10     {
11         printf("%d, ", tab[i]);
12         i++;
13     }
14     printf("]\n");
15 }
```

pretty_printing.h

```
1 #include <stdio.h>
2
3 void pretty_printing_tab(int *tab, size_t size);
```



 main.c


```
1 #include "pretty_printing.h"
2
3 int main(void)
4 {
5     int tab[3] = {1, 123, 1231312};
6     pretty_printing_tab(tab, sizeof(tab) / sizeof(tab[0]));
7
8 }
```



 pretty_printing.h

```
1 #include <stdio.h>
2
3 void pretty_printing_tab(int *tab, size_t size);
```



 pretty_printing.c

```
1 #include "pretty_printing.h"
2
3 void pretty_printing_tab(int *tab, size_t size)
4 {
5     size_t i;
6
7     i = 0;
8     printf("[");
9     while (i < size)
10     {
11         printf("%d, ", tab[i]);
12         i++;
13     }
14     printf("]\n");
15 }
```



main.c

```
1 #include "pretty_printing.h"
2
3 int main(void)
4 {
5     int tab[3] = {1, 123, 1231312};
6     pretty_printing_tab(tab, sizeof(tab) / sizeof(tab[0]));
7
8 }
```



pretty_printing.h

```
1 #include <stdio.h>
2
3 void pretty_printing_tab(int *tab, size_t size);
```

Voilà nos fichiers.



pretty_printing.c

```
1 #include "pretty_printing.h"
2
3 void pretty_printing_tab(int *tab, size_t size)
4 {
5     size_t i;
6
7     i = 0;
8     printf("[");
9     while (i < size)
10     {
11         printf("%d, ", tab[i]);
12         i++;
13     }
14     printf("]\n");
15 }
```

```
main.c
1 #include "pretty_printing.h"
2
3 int main(void)
4 {
5     int tab[3] = {1, 123, 1231312};
6     pretty_printing_tab(tab, sizeof(tab) / sizeof(tab[0]));
7
8 }
```

```
pretty_printing.h
1 #include <stdio.h>
2
3 void pretty_printing_tab(int *tab, size_t size);
```

```
pretty_printing.c
1 #include "pretty_printing.h"
2
3 void pretty_printing_tab(int *tab, size_t size)
4 {
5     size_t i;
6
7     i = 0;
8     printf("[");
9     while (i < size)
10     {
11         printf("%d, ", tab[i]);
12         i++;
13     }
14     printf("]\n");
15 }
```

Voilà nos fichiers.

le "pretty_printing.h" n'est pas `<pretty_printing.h>` car il n'est pas dans un endroit spéciale où quoi que ce soit, il est dans notre projet actuel, et il faut donc **le préciser** [son chemin relatif] à la compilation.

```
main.c
1 #include "pretty_printing.h"
2
3 int main(void)
4 {
5     int tab[3] = {1, 123, 1231312};
6     pretty_printing_tab(tab, sizeof(tab) / sizeof(tab[0]));
7
8 }
```

```
pretty_printing.h
1 #include <stdio.h>
2
3 void pretty_printing_tab(int *tab, size_t size);
```

```
pretty_printing.c
1 #include "pretty_printing.h"
2
3 void pretty_printing_tab(int *tab, size_t size)
4 {
5     size_t i;
6
7     i = 0;
8     printf("[");
9     while (i < size)
10     {
11         printf("%d, ", tab[i]);
12         i++;
13     }
14     printf("]\n");
15 }
```

Voilà nos fichiers.

le "pretty_printing.h" n'est pas `<pretty_printing.h>` car il n'est pas dans un endroit spéciale où quoi que ce soit, il est dans notre projet actuel, et il faut donc **le préciser** [son chemin relatif] à la compilation.

Pour cela, on peut utiliser l'option `-I` `[/chemin/relatif]`, en remplaçant avec le vrai chemin bien sûr.


```
main.c
1 #include "pretty_printing.h"
2
3 int main(void)
4 {
5     int tab[3] = {1, 123, 1231312};
6     pretty_printing_tab(tab, sizeof(tab) / sizeof(tab[0]));
7
8 }
```

```
pretty_printing.h
1 #include <stdio.h>
2
3 void pretty_printing_tab(int *tab, size_t size);
```

```
pretty_printing.c
1 #include "pretty_printing.h"
2
3 void pretty_printing_tab(int *tab, size_t size)
4 {
5     size_t i;
6
7     i = 0;
8     printf("[");
9     while (i < size)
10     {
11         printf("%d, ", tab[i]);
12         i++;
13     }
14     printf("]\n");
15 }
```

Voilà nos fichiers.

le "pretty_printing.h" n'est pas `<pretty_printing.h>` car il n'est pas dans un endroit spéciale où quoi que ce soit, il est dans notre projet actuel, et il faut donc **le préciser** [son chemin relatif] à la compilation.

Pour cela, on peut utiliser l'option `-I` `[/chemin/relatif]`, en remplaçant avec le vrai chemin bien sûr.

Voyons cela :

Rappelons notre structure idéale de fichiers :

```
•  
├─ README.md <- pour des infos  
├─ src/  
│   └─ les fichiers .c  
└─ includes/  
    └─ les fichiers .h
```

Rappelons notre structure idéale de fichiers :

```
.  
├── README.md <- pour des infos  
├── src/  
│   ├── les fichiers .c  
└── includes/  
    ├── les fichiers .h
```

On veut quelque chose comme ça, donc on a :

Rappelons notre structure idéale de fichiers :

```
.
├── README.md <- pour des infos
├── src/
│   └── les fichiers .c
└── includes/
    └── les fichiers .h
```

On veut quelque chose comme ça, donc on a :

```
.
├── README.md <- pour des infos [optionel]
├── src/
│   ├── main.c
│   └── pretty_printing.c
└── includes/
    └── pretty_printing.h
```

Rappelons notre structure idéale de fichiers :

```
.  
├── README.md <- pour des infos  
├── src/  
│   ├── les fichiers .c  
└── includes/  
    └── les fichiers .h
```

On veut quelque chose comme ça, donc on a :

```
.  
├── README.md <- pour des infos [optionell]  
├── src/  
│   ├── main.c  
│   └── pretty_printing.c  
└── includes/  
    └── pretty_printing.h
```

```
~/Desktop/Universite/test/demo main*  
> gcc src/main.c src/pretty_printing.c
```

Rappelons notre structure idéale de fichiers :

```
.
├── README.md <- pour des infos
├── src/
│   ├── les fichiers .c
└── includes/
    └── les fichiers .h
```

On veut quelque chose comme ça, donc on a :

```
.
├── README.md <- pour des infos [optionell]
├── src/
│   ├── main.c
│   └── pretty_printing.c
└── includes/
    └── pretty_printing.h
```

```
~/Desktop/Universite/test/demo main*
> gcc src/main.c src/pretty_printing.c
src/main.c:1:10: fatal error: 'pretty_printing.h' file not found
  1 | #include "pretty_printing.h"
    |           ^~~~~~
1 error generated.
src/pretty_printing.c:1:10: fatal error: 'pretty_printing.h' file not found
  1 | #include "pretty_printing.h"
    |           ^~~~~~
1 error generated.
```

Rappelons notre structure idéale de fichiers :

```
.
├── README.md <- pour des infos
├── src/
│   └── les fichiers .c
└── includes/
    └── les fichiers .h
```

On veut quelque chose comme ça, donc on a :

```
.
├── README.md <- pour des infos [optionell]
├── src/
│   ├── main.c
│   └── pretty_printing.c
└── includes/
    └── pretty_printing.h
```

```
~/Desktop/Universite/test/demo main*
> gcc src/main.c src/pretty_printing.c
```

Pourquoi ça ne compile pas ?

Rappelons notre structure idéale de fichiers :

```
.  
├── README.md <- pour des infos  
├── src/  
│   ├── les fichiers .c  
└── includes/  
    └── les fichiers .h
```

On veut quelque chose comme ça, donc on a :

```
.  
├── README.md <- pour des infos [optionell]  
├── src/  
│   ├── main.c  
│   └── pretty_printing.c  
└── includes/  
    └── pretty_printing.h
```

```
~/Desktop/Universite/test/demo main*  
> gcc src/main.c src/pretty_printing.c
```

Pourquoi ça ne compile pas ?
Il ne trouve pas le fichier .h ! (comme prévu)

Rappelons notre structure idéale de fichiers :

```
.  
├── README.md <- pour des infos  
├── src/  
│   ├── les fichiers .c  
└── includes/  
    └── les fichiers .h
```

On veut quelque chose comme ça, donc on a :

```
.  
├── README.md <- pour des infos [optionell]  
├── src/  
│   ├── main.c  
│   └── pretty_printing.c  
└── includes/  
    └── pretty_printing.h
```

```
~/Desktop/Universite/test/demo main*  
> gcc src/main.c src/pretty_printing.c
```

Pourquoi ça ne compile pas ?
Il ne trouve pas le fichier .h ! (comme prévu)

```
~/Desktop/Universite/test/demo main*  
> gcc src/main.c src/pretty_printing.c -I ./includes
```

Rappelons notre structure idéale de fichiers :

```
.
├── README.md <- pour des infos
├── src/
│   └── les fichiers .c
└── includes/
    └── les fichiers .h
```

On veut quelque chose comme ça, donc on a :

```
.
├── README.md <- pour des infos [optionell]
├── src/
│   ├── main.c
│   └── pretty_printing.c
└── includes/
    └── pretty_printing.h
```

```
~/Desktop/Universite/test/demo main*
> gcc src/main.c src/pretty_printing.c
```

Pourquoi ça ne compile pas ?
Il ne trouve pas le fichier .h ! (comme prévu)

```
~/Desktop/Universite/test/demo main*
> gcc src/main.c src/pretty_printing.c -I ./includes

~/Desktop/Universite/test/demo main*
> ls
a.out  includes  src
```

Rappelons notre structure idéale de fichiers :

```
.
├── README.md <- pour des infos
├── src/
│   └── les fichiers .c
└── includes/
    └── les fichiers .h
```

On veut quelque chose comme ça, donc on a :

```
.
├── README.md <- pour des infos [optionell]
├── src/
│   ├── main.c
│   └── pretty_printing.c
└── includes/
    └── pretty_printing.h
```

```
~/Desktop/Universite/test/demo main*
> gcc src/main.c src/pretty_printing.c
```

Pourquoi ça ne compile pas ?
Il ne trouve pas le fichier .h ! (comme prévu)

```
~/Desktop/Universite/test/demo main*
> gcc src/main.c src/pretty_printing.c -I ./includes

~/Desktop/Universite/test/demo main*
> ls
a.out  includes  src
```

Rappelons notre structure idéale de fichiers :

```
.  
├── README.md <- pour des infos  
├── src/  
│   ├── les fichiers .c  
└── includes/  
    └── les fichiers .h
```

On veut quelque chose comme ça, donc on a :

```
.  
├── README.md <- pour des infos [optionell]  
├── src/  
│   ├── main.c  
│   └── pretty_printing.c  
└── includes/  
    └── pretty_printing.h
```

```
~/Desktop/Universite/test/demo main*  
> gcc src/main.c src/pretty_printing.c
```

Pourquoi ça ne compile pas ?
Il ne trouve pas le fichier .h ! (comme prévu)

```
~/Desktop/Universite/test/demo main*  
> gcc src/main.c src/pretty_printing.c -I ./includes  
  
~/Desktop/Universite/test/demo main*  
> ls  
a.out includes src
```

-I [chemin] sert à préciser les chemins relatifs vers tous les dossiers qui contiennent des .h

Problèmes potentiels

- Si nous changeons uniquement les contenus de *main.c*, devons-nous forcément recompiler tous nos fichiers ?

Problèmes potentiels

- Si nous changeons uniquement les contenus de *main.c*, devons-nous forcément recompiler tous nos fichiers ?
- Non ! Grâce à l'intermédiaire des *.o* , strictement parlant, si nous **compilons** uniquement en *.o* alors nous pouvons simplement recompiler les fichiers *.c* **dont la date de modification est différente de la date de création de leurs *.o* respectifs**, et relinker le tout

Problèmes potentiels

- Si nous changeons uniquement les contenus de *main.c*, devons-nous forcément recompiler tous nos fichiers ?
- Non ! Grâce à l'intermédiaire des *.o* , strictement parlant, si nous **compilons** uniquement en *.o* alors nous pouvons simplement recompiler les fichiers *.c* **dont la date de modification est différente de la date de création de leurs *.o* respectifs**, et relinker le tout
- Voyons comment cela fonctionne

```
~/Desktop/Universite/test/demo main*
```

```
> gcc src/main.c src/pretty_printing.c -I ./includes -c
```



```
~/Desktop/Universite/test/demo main*  
> gcc src/main.c src/pretty_printing.c -I ./includes -c
```

-c pour dire “donne moi des .o”

```
~/Desktop/Universite/test/demo main*  
> gcc src/main.c src/pretty_printing.c -I ./includes -c
```

-c pour dire “donne-moi des .o”

```
~/Desktop/Universite/test/demo main*  
> ls  
includes  main.o  pretty_printing.o  src
```

```
~/Desktop/Universite/test/demo main*  
> gcc src/main.c src/pretty_printing.c -I ./includes -c
```

-c pour dire “donne-moi des .o”

```
~/Desktop/Universite/test/demo main*  
> ls  
includes  main.o  pretty_printing.o  src
```

Voilà nos fichiers .o , on remarque qu’ils sont au milieu de nos fichiers et ce n’est pas très propre..

```
~/Desktop/Universite/test/demo main*  
> gcc src/main.c src/pretty_printing.c -I ./includes -c
```

-c pour dire “donne-moi des .o”

```
~/Desktop/Universite/test/demo main*  
> ls  
includes  main.o  pretty_printing.o  src
```

Voilà nos fichiers .o , on remarque qu’ils sont au milieu de nos fichiers et ce n’est pas très propre..

```
~/Desktop/Universite/test/demo main*  
> gcc main.o pretty_printing.o -o exec  
  
~/Desktop/Universite/test/demo main*  
> ls  
exec  includes  main.o  pretty_printing.o  src
```

```
~/Desktop/Universite/test/demo main*  
> gcc src/main.c src/pretty_printing.c -I ./includes -c
```

-c pour dire “donne-moi des .o”

```
~/Desktop/Universite/test/demo main*  
> ls  
includes  main.o  pretty_printing.o  src
```

Voilà nos fichiers .o , on remarque qu’ils sont au milieu de nos fichiers et ce n’est pas très propre..

```
~/Desktop/Universite/test/demo main*  
> gcc main.o pretty_printing.o -o exec  
  
~/Desktop/Universite/test/demo main*  
> ls  
exec  includes  main.o  pretty_printing.o  src
```

On ajoute -o pour renommer le a.out

```
~/Desktop/Universite/test/demo main*  
> gcc src/main.c src/pretty_printing.c -I ./includes -c
```

-c pour dire “donne-moi des .o”

```
~/Desktop/Universite/test/demo main*  
> ls  
includes  main.o  pretty_printing.o  src
```

Voilà nos fichiers .o , on remarque qu'ils sont au milieu de nos fichiers et ce n'est pas très propre..

```
~/Desktop/Universite/test/demo main*  
> gcc main.o pretty_printing.o -o exec  
  
~/Desktop/Universite/test/demo main*  
> ls  
exec  includes  main.o  pretty_printing.o  src
```

On ajoute -o pour renommer le a.out

```
~/Desktop/Universite/test/demo main*  
> gcc src/pretty_printing.c -I ./includes -c
```

```
~/Desktop/Universite/test/demo main*  
> gcc src/main.c src/pretty_printing.c -I ./includes -c
```

-c pour dire “donne-moi des .o”

```
~/Desktop/Universite/test/demo main*  
> ls  
includes  main.o  pretty_printing.o  src
```

Voilà nos fichiers .o , on remarque qu'ils sont au milieu de nos fichiers et ce n'est pas très propre..

```
~/Desktop/Universite/test/demo main*  
> gcc main.o pretty_printing.o -o exec  
  
~/Desktop/Universite/test/demo main*  
> ls  
exec  includes  main.o  pretty_printing.o  src
```

On ajoute -o pour renommer le a.out

```
~/Desktop/Universite/test/demo main*  
> gcc src/pretty_printing.c -I ./includes -c  
  
~/Desktop/Universite/test/demo main*  
> gcc main.o pretty_printing.o -o exec
```

```
~/Desktop/Universite/test/demo main*  
> gcc src/main.c src/pretty_printing.c -I ./includes -c
```

-c pour dire “donne-moi des .o”

```
~/Desktop/Universite/test/demo main*  
> ls  
includes  main.o  pretty_printing.o  src
```

Voilà nos fichiers .o , on remarque qu'ils sont au milieu de nos fichiers et ce n'est pas très propre..

```
~/Desktop/Universite/test/demo main*  
> gcc main.o pretty_printing.o -o exec  
  
~/Desktop/Universite/test/demo main*  
> ls  
exec  includes  main.o  pretty_printing.o  src
```

On ajoute -o pour renommer le a.out

```
~/Desktop/Universite/test/demo main*  
> gcc src/pretty_printing.c -I ./includes -c  
  
~/Desktop/Universite/test/demo main*  
> gcc main.o pretty_printing.o -o exec  
  
~/Desktop/Universite/test/demo main*  
> ls  
exec  includes  main.o  pretty_printing.o  src
```



```
~/Desktop/Universite/test/demo main*  
> gcc src/main.c src/pretty_printing.c -I ./includes -c
```

-c pour dire “donne-moi des .o”

```
~/Desktop/Universite/test/demo main*  
> ls  
includes  main.o  pretty_printing.o  src
```

Voilà nos fichiers .o , on remarque qu'ils sont au milieu de nos fichiers et ce n'est pas très propre..

```
~/Desktop/Universite/test/demo main*  
> gcc main.o pretty_printing.o -o exec  
  
~/Desktop/Universite/test/demo main*  
> ls  
exec  includes  main.o  pretty_printing.o  src
```

On ajoute -o pour renommer le a.out

```
~/Desktop/Universite/test/demo main*  
> gcc src/pretty_printing.c -I ./includes -c  
  
~/Desktop/Universite/test/demo main*  
> gcc main.o pretty_printing.o -o exec  
  
~/Desktop/Universite/test/demo main*  
> ls  
exec  includes  main.o  pretty_printing.o  src
```

Donc on peut simplement recompiler *pretty_printing.c* en .o et dire au compilateur de relinker tout ça en lui donnant les .o

Conclusion

- Pour 2-3 fichiers, cela peut être gérable, mais pour plus de fichiers, cela devient très compliqué.

Conclusion : Build System

- Pour 2-3 fichiers, cela peut être gérable, mais pour plus de fichiers, cela devient très compliqué.
- On passe donc par un **build system**, qui se chargera de cela

Conclusion : Build System

- Pour 2-3 fichiers, cela peut être gérable, mais pour plus de fichiers, cela devient très compliqué.
- On passe donc par un **build system**, qui se chargera de cela
- Vous avez peut-être vu Gradle ou Maven pour Java, mais il en existe bien plus, en fonction de vos besoins.

Conclusion : Build System

- Pour 2-3 fichiers, cela peut être gérable, mais pour plus de fichiers, cela devient très compliqué.
- On passe donc par un **build system**, qui se chargera de cela
- Vous avez peut-être vu Gradle ou Maven pour Java, mais il en existe bien plus, en fonction de vos besoins.
- Voyons en quelque uns.

Build systems

- **make** : Vétéran de l'industrie, works for small to medium-sized projects

Build systems

- **make** : Vétéran de l'industrie, works for small to medium-sized projects
- **CMake** : Courbe d'apprentissage plus abrupte que celle de la physique quantique, mais très utile pour la compilation de projets sur plusieurs architectures.

Build systems

- **make** : Vétéran de l'industrie, works for small to medium-sized projects
- **CMake** : Courbe d'apprentissage plus abrupte que celle de la physique quantique, mais très utile pour la compilation de projets sur plusieurs architectures.
- **Meson** : Relativement nouveau, mix entre make et CMake, très prometteur.

Build systems

- **make** : Vétéran de l'industrie, works for small to medium-sized projects
- **CMake** : Courbe d'apprentissage plus abrupte que celle de la physique quantique, mais très utile pour la compilation de projets sur plusieurs architectures.
- **Meson** : Relativement nouveau, mix entre make et CMake, très prometteur.
- Et bien d'autres encore

make

- Dans ce cours, nous allons voir **make** et le fameux **Makefile**

make

- Dans ce cours, nous allons voir **make** et le fameux **Makefile**
- Créé en 1976 par Stuart Feldman dans Bell Labs

make

- Dans ce cours, nous allons voir **make** et le fameux **Makefile**
- Créé en 1976 par Stuart Feldman dans Bell Labs
- Utilise la programmation déclarative

make

- Dans ce cours, nous allons voir **make** et le fameux **Makefile**
- Créé en 1976 par Stuart Feldman dans Bell Labs
- Utilise la programmation déclarative
- Observons les features de ce langage :

Les MACROS

- Permet de déclarer des macros (variable) :

Les MACROS

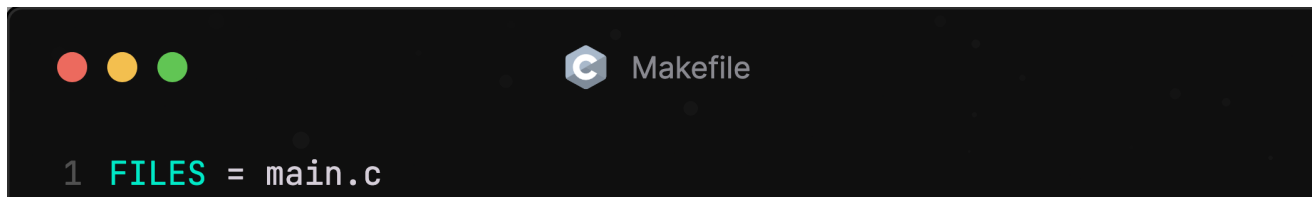
- Permet de déclarer des macros (variable) :
- Certaines MACROS sont déjà prédéfinies : [Implicit variables](#)

Les MACROS

- Permet de déclarer des macros (variable) :
- Certaines MACROS sont déjà prédéfinies : [Implicit variables](#)
- Voilà un exemple :

Les MACROS

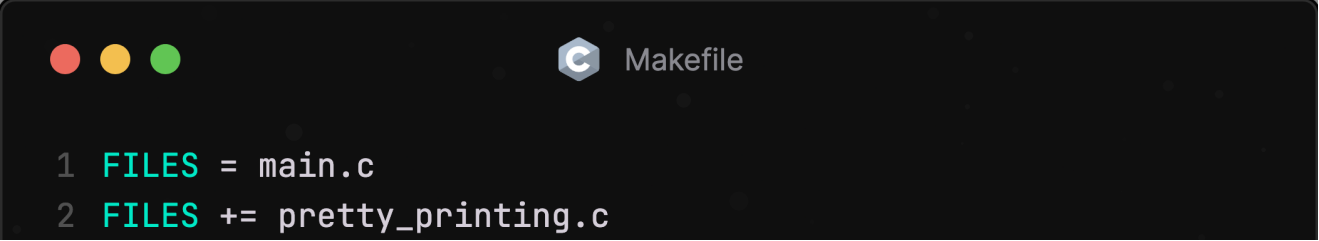
- Permet de déclarer des macros (variable) :
- Certaines MACROS sont déjà prédéfinies : [Implicit variables](#)
- Voilà un exemple :

A screenshot of a code editor window. The title bar shows three colored circles (red, yellow, green) on the left and a hexagonal icon with a 'C' followed by the text 'Makefile' on the right. The code area has a dark background and shows a single line of text: '1 FILES = main.c'. The word 'FILES' is highlighted in a light blue color.

```
1 FILES = main.c
```

Les MACROS

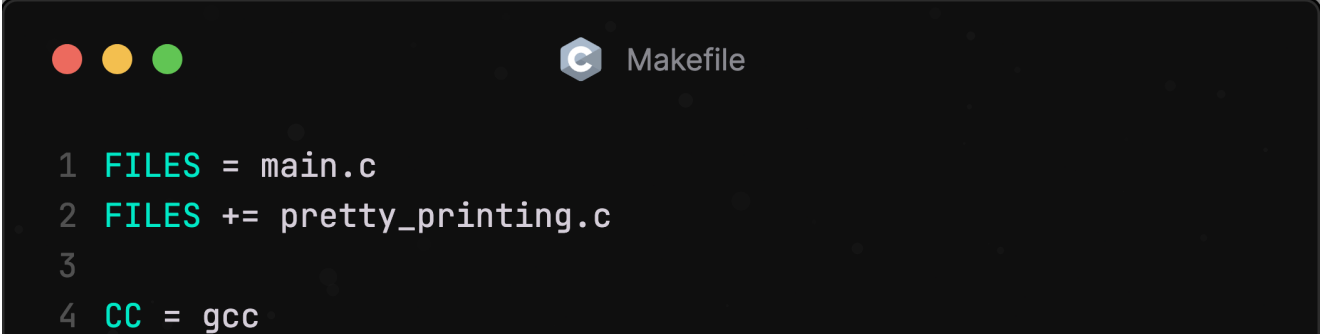
- Permet de déclarer des macros (variable) :
- Certaines MACROS sont déjà prédéfinies : [Implicit variables](#)
- Voilà un exemple :

A terminal window with a dark background. At the top left are three colored circles (red, yellow, green). At the top center is a hexagonal icon with a 'C' and the text 'Makefile'. The terminal contains two lines of text: '1 FILES = main.c' and '2 FILES += pretty_printing.c'.

```
1 FILES = main.c
2 FILES += pretty_printing.c
```

Les MACROS

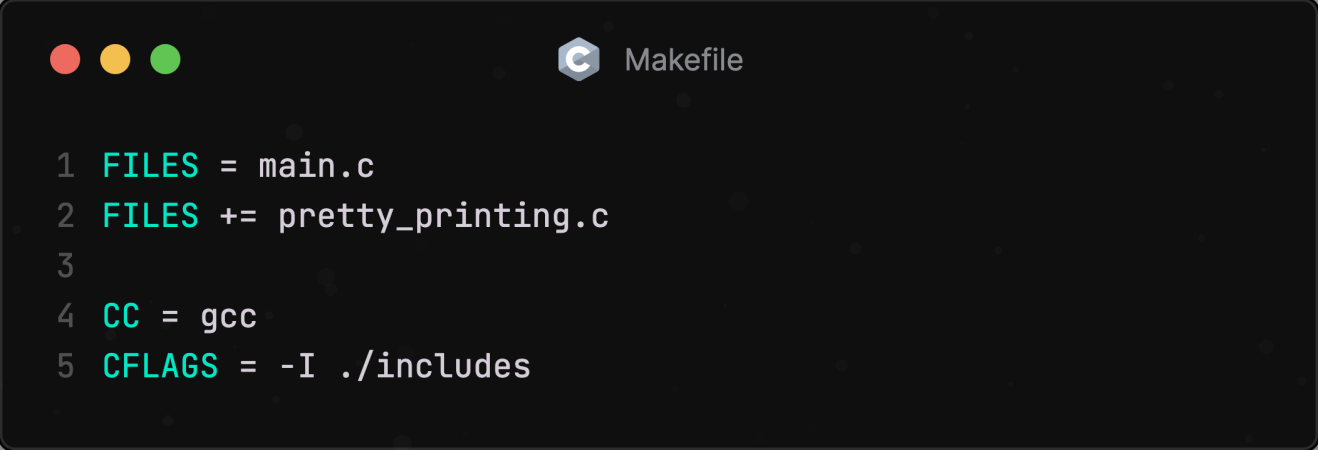
- Permet de déclarer des macros (variable) :
- Certaines MACROS sont déjà prédéfinies : [Implicit variables](#)
- Voilà un exemple :

A terminal window with a dark background. At the top, there are three colored circles (red, yellow, green) and a logo with the letter 'C' followed by the word 'Makefile'. Below this, there is a list of four lines of code, each preceded by a number from 1 to 4. The code defines the 'FILES' variable and the compiler 'CC'.

```
1 FILES = main.c
2 FILES += pretty_printing.c
3
4 CC = gcc
```

Les MACROS

- Permet de déclarer des macros (variable) :
- Certaines MACROS sont déjà prédéfinies : [Implicit variables](#)
- Voilà un exemple :



```
1 FILES = main.c
2 FILES += pretty_printing.c
3
4 CC = gcc
5 CFLAGS = -I ./includes
```

Les Rules

- Le nom d'une règle est un **target**

Les Rules

- Le nom d'une règle est un **target**
- Une règle peut dépendre d'autres règles ou de variables (qui généralement contiennent des fichiers) : des **prerequisites**

Les Rules

- Le nom d'une règle est un **target**
- Une règle peut dépendre d'autres règles ou de variables (qui généralement contiennent des fichiers) : des **prerequisites**
- Le contenu d'une règle est appelé la **recipe**

Les Rules

- Le nom d'une règle est un **target**
- Une règle peut dépendre d'autres règles ou de variables (qui généralement contiennent des fichiers) : des **prerequisites**
- Le contenu d'une règle est appelé la **recipe**
- Le *target* est soit un fichier à générer (l'exécutable à générer par exemple) ou un concept défini par l'utilisateur (par exemple une règle "help" qui affiche les commandes disponibles), dans ce cas-là, on appelle ça une **phony target**

Les Rules

- Le nom d'une règle est un **target**
- Une règle peut dépendre d'autres règles ou de variables (qui généralement contiennent des fichiers) : des **prerequisites**
- Le contenu d'une règle est appelé la **recipe**
- Le *target* est soit un fichier à générer (l'exécutable à générer par exemple) ou un concept défini par l'utilisateur (par exemple une règle "help" qui affiche les commandes disponibles), dans ce cas-là, on appelle ça une **phony target**
- Essayons donc de faire un makefile pour notre projet



```
1 FICHIERS_SRC = src/main.c
2 FICHIERS_SRC += pretty_printing.c
3
```



```
1 FICHIERS_SRC = src/main.c
2 FICHIERS_SRC += pretty_printing.c
3
4 FICHIERS_OBJ = $(patsubst %.c, %.o, $(FICHIERS_SRC))
```

```
FICHIERS_OBJ = $(patsubst %.c, %.o, $(FICHIERS_SRC))
```

On utilise la *fonction texte* **patsubst** qui va garder tout le contenu qui est “matcher” par les ‘%’, et changer .c en .o

```
FICHIERS_OBJ = $(patsubst %.c, %.o, $(FICHIERS_SRC))
```

On utilise la *fonction texte* **patsubst** qui va garder tout le contenu qui est “matcher” par les ‘%’, et changer **.c** en **.o**

- Le contenu de FICHIERS_OBJ sera nos fichiers **.c**, mais à la place du **.c** on aura des **.o**

```
FICHIERS_OBJ = $(patsubst %.c, %.o, $(FICHIERS_SRC))
```

On utilise la *fonction texte* **patsubst** qui va garder tout le contenu qui est “matcher” par les ‘%’, et changer .c en .o

- Le contenu de FICHIERS_OBJ sera nos fichiers .c, mais à la place du .c on aura des .o
- Donc si :
 - FICHIERS_SRC vaut "src/main.c src/pretty_printing.c"

```
FICHIERS_OBJ = $(patsubst %.c, %.o, $(FICHIERS_SRC))
```

On utilise la *fonction texte* **patsubst** qui va garder tout le contenu qui est “matcher” par les ‘%’, et changer .c en .o

- Le contenu de FICHIERS_OBJ sera nos fichiers .c, mais à la place du .c on aura des .o
- Donc si :
 - FICHIERS_SRC vaut "src/main.c src/pretty_printing.c"
 - FICHIERS_OBJ vaut "src/main.o src/pretty_printing.o"



```
1 FICHIERS_SRC = src/main.c
2 FICHIERS_SRC += pretty_printing.c
3
4 FICHIERS_OBJ = $(patsubst %.c, %.o, $(FICHIERS_SRC))
5
6 CC = gcc
7 CFLAGS = -I ./includes
```




```
1 FICHIERS_SRC = src/main.c
2 FICHIERS_SRC += pretty_printing.c
3
4 FICHIERS_OBJ = $(patsubst %.c, %.o, $(FICHIERS_SRC))
5
6 CC = gcc
7 CFLAGS = -I ./includes
8
9 %.o: %.c
10     $(CC) $(CFLAGS) -o $@ -c $<
```

%.o: %.c

gcc \$(CFLAGS) \$< -o \$@ -c

```
%.o: %.c  
gcc $(CFLAGS) $< -o $@ -c
```

Cette ligne permet de compiler chaque **.c** en son **.o** de manière spécifique, c'est une **pattern rule**

```
%.o: %.c  
gcc $(CFLAGS) $< -o $@ -c
```

Cette ligne permet de compiler chaque **.c** en son **.o** de manière spécifique, c'est une **pattern rule**

Elle est très utile vu qu'avant de compiler bêtement un fichier, disons *main.c*, elle va vérifier si il avait été modifié depuis la création de son **.o** (*main.o*).

```
%.o: %.c  
gcc $(CFLAGS) $< -o $@ -c
```

Cette ligne permet de compiler chaque **.c** en son **.o** de manière spécifique, c'est une **pattern rule**

Elle est très utile vu qu'avant de compiler bêtement un fichier, disons **main.c**, elle va vérifier si il avait été modifié depuis la création de son **.o** (**main.o**).

Reprenons notre makefile



```
1 FICHIERS_SRC = src/main.c
2 FICHIERS_SRC += pretty_printing.c
3
4 FICHIERS_OBJ = $(patsubst %.c, %.o, $(FICHIERS_SRC))
5
6 CC = gcc
7 CFLAGS = -I ./includes
8
9 %.o: %.c
10     $(CC) $(CFLAGS) -o $@ -c $<
11
```



```
1 FICHIERS_SRC = src/main.c
2 FICHIERS_SRC += pretty_printing.c
3
4 FICHIERS_OBJ = $(patsubst %.c, %.o, $(FICHIERS_SRC))
5
6 CC = gcc
7 CFLAGS = -I ./includes
8
9 %.o: %.c
10     $(CC) $(CFLAGS) -o $@ -c $<
11
12 a.out: $(FICHIERS_OBJ)
```



```
1 FICHIERS_SRC = src/main.c
2 FICHIERS_SRC += pretty_printing.c
3
4 FICHIERS_OBJ = $(patsubst %.c, %.o, $(FICHIERS_SRC))
5
6 CC = gcc
7 CFLAGS = -I ./includes
8
9 %.o: %.c
10     $(CC) $(CFLAGS) -o $@ -c $<
11
12 a.out: $(FICHIERS_OBJ)
13     @echo "Linking les fichiers objects !"
```




```
1 FICHIERS_SRC = src/main.c
2 FICHIERS_SRC += pretty_printing.c
3
4 FICHIERS_OBJ = $(patsubst %.c, %.o, $(FICHIERS_SRC))
5
6 CC = gcc
7 CFLAGS = -I ./includes
8
9 %.o: %.c
10     $(CC) $(CFLAGS) -o $@ -c $<
11
12 a.out: $(FICHIERS_OBJ)
13     @echo "Linking les fichiers objects !"
14     $(CC) $(FICHIERS_OBJ)
```




```
1 FICHIERS_SRC = src/main.c
2 FICHIERS_SRC += pretty_printing.c
3
4 FICHIERS_OBJ = $(patsubst %.c, %.o, $(FICHIERS_SRC))
5
6 CC = gcc
7 CFLAGS = -I ./includes
8
9 %.o: %.c
10     $(CC) $(CFLAGS) -o $@ -c $<
11
12 a.out: $(FICHIERS_OBJ)
13     @echo "Linking les fichiers objects !"
14     $(CC) $(FICHIERS_OBJ)
```

Automatic Variables

Petite parenthèse

```
9  %.o: %.c
```

```
10 TAB $(CC) $(CFLAGS) -o $@ -c $<
```

```
11
```

```
12 a.out: $(FICHIERS_OBJ)
```

```
13 TAB @echo "Linking les fichiers objects !"
```

```
14 TAB $(CC) $(FICHIERS_OBJ)
```

espace ≠ TAB

```
a.out: $(FICHIERS_OBJ)
```

```
@echo "Linking les objects files !"
```

```
gcc $(CFLAGS) $(FICHIERS_OBJ)
```

```
a.out: $(FICHIERS_OBJ)
    @echo "Linking les objects files !"
    gcc $(CFLAGS) $(FICHIERS_OBJ)
```

Toute la magie du makefile se joue ici, “a.out” (le **target name**), est le nom de la règle, et comme **prerequisites** on a les fichiers .o , et donc on va vérifier si chaque .o est "en retard" par rapport à son .c respectif

```
a.out: $(FICHIERS_OBJ)
    @echo "Linking les objects files !"
    gcc $(CFLAGS) $(FICHIERS_OBJ)
```

Toute la magie du makefile se joue ici, “a.out” (le **target name**), est le nom de la règle, et comme **prerequisites** on a les fichiers .o , et donc on va vérifier si chaque .o est "en retard" par rapport à son .c respectif

Tout d'abord, on vérifie si au moins un .o (un élément de FICHIERS_OBJ) a une date de création différente de celle de modification de son .c respectif, *(si il n'existe pas, on va le créer grâce en utilisant notre propre conversion, ligne qui contient %.o : %.c)*. Si c'est le cas, on le(s) recompile, et on retourne exécuter la recette (le contenu de la règle).

```
a.out: $(FICHIERS_OBJ)
    @echo "Linking les objects files !"
    gcc $(CFLAGS) $(FICHIERS_OBJ)
```

Toute la magie du makefile se joue ici, “a.out” (le **target name**), est le nom de la règle, et comme **prerequisites** on a les fichiers .o , et donc on va vérifier si chaque .o est "en retard" par rapport à son .c respectif

Tout d'abord, on vérifie si au moins un .o (un élément de FICHIERS_OBJ) a une date de création différente de celle de modification de son .c respectif, *(si il n'existe pas, on va le créer grâce en utilisant notre propre conversion, ligne qui contient %.o : %.c)*. Si c'est le cas, on le(s) recompile, et on retourne exécuter la recette (le contenu de la règle).

Si a.out n'existe pas, c'est une raison de rentrer exécuter la règle, même si tout les .o sont à jours


```
a.out: $(FICHIERS_OBJ)
@echo "Linking les objects files !"
gcc $(CFLAGS) $(FICHIERS_OBJ)
```

Toute la magie du makefile se joue ici, “a.out” (le **target name**), est le nom de la règle, et comme **prerequisites** on a les fichiers .o , et donc on va vérifier si chaque .o est "en retard" par rapport à son .c respectif

Tout d'abord, on vérifie si au moins un .o (un élément de FICHIERS_OBJ) a une date de création différente de celle de modification de son .c respectif, *(si il n'existe pas, on va le créer grâce en utilisant notre propre conversion, ligne qui contient %.o : %.c)*. Si c'est le cas, on le(s) recompile, et on retourne exécuter la recette (le contenu de la règle).

Si a.out n'existe pas, c'est une raison de retourner exécuter la règle, même si tous les .o sont à jour

Donc on exécute le linking dans notre cas si et seulement si a.out n'existe pas, ou un des fichiers .o a dû être recompilé

```
a.out: $(FICHIERS_OBJ)
@echo "Linking les objects files !"
gcc $(CFLAGS) $(FICHIERS_OBJ)
```

Petite parenthèse sur le nom de notre target : ici c'est le nom par default de l'exécutable, mais on peut change sa a par exemple dans notre cas : **pretty_print**, sauf qu'on devra aussi ajoute **-o pretty_print** comme options a la ligne de linking

```
a.out: $(FICHIERS_OBJ)
@echo "Linking les objects files !"
gcc $(CFLAGS) $(FICHIERS_OBJ)
```

Petite parenthèse sur le nom de notre target : ici c'est le nom par default de l'exécutable, mais on peut change sa a par exemple dans notre cas : **pretty_print**, sauf qu'on devra aussi ajoute **-o pretty_print** comme options a la ligne de linking

Généralement vous trouverez une variable NAME = ... avec le nom de l'exécutable
Et la règle de compilation qui ressemble a sa :

```
a.out: $(FICHIERS_OBJ)
@echo "Linking les objects files !"
gcc $(CFLAGS) $(FICHIERS_OBJ)
```

Petite parenthèse sur le nom de notre target : ici c'est le nom par default de l'exécutable, mais on peut change sa a par exemple dans notre cas : **pretty_print**, sauf qu'on devra aussi ajoute **-o pretty_print** comme options a la ligne de linking

Généralement vous trouverez une variable NAME = ... avec le nom de l'exécutable
Et la règle de compilation qui ressemble a sa :

```
$(NAME): $(FICHIERS_OBJ)
@echo "Linking les fichiers objects !"
$(CC) $(FICHIERS_OBJ) -o $(NAME)
```

Testons cela

- Si on essaye d'appeler “make”, il va d'abord chercher le makefile (Makefile ou makefile ou GNUmakefile), si on appelle make sans argument, il va exécuter la 1ère règle du fichier

Testons cela

- Si on essaye d'appeler "make", il va d'abord chercher le makefile (Makefile ou makefile ou GNUmakefile), si on appelle make sans argument, il va exécuter la 1ère règle du fichier
- Donc généralement on va faire une règle 'all' qui appelle dans notre cas a.out, et on utilise la variable spécial **.DEFAULT_GOAL** en la mettant à **all** afin d'exécuter la règle all quand on dit « make »

Testons cela

- Si on essaye d'appeler "make", il va d'abord chercher le makefile (Makefile ou makefile ou GNUmakefile), si on appelle make sans argument, il va exécuter la 1ère règle du fichier
- Donc généralement on va faire une règle 'all' qui appelle dans notre cas a.out, et on utilise la variable spécial **.DEFAULT_GOAL** en la mettant à **all** afin d'exécuter la règle all quand on dit « make »
- On peut par contre faire bien mieux !

Et voilà

- On a donc un makefile qui compile avec notre project actuel, après il y a beaucoup de chose à améliorer, comme :

Et voilà

- On a donc un makefile qui compile avec notre project actuel, après il y a beaucoup de chose à améliorer, comme :
- Déplacer tous les fichiers .o dans leur propre dossier au moment de la compilation

Et voilà

- On a donc un makefile qui compile avec notre project actuel, après il y a beaucoup de chose à améliorer, comme :
- Déplacer tous les fichiers .o dans leur propre dossier au moment de la compilation
- Avoir des variable pour les chemins, le nom de l'exécutable, etc..

Et voilà

- On a donc un makefile qui compile avec notre project actuel, après il y a beaucoup de chose à améliorer, comme :
- Déplacer tous les fichiers .o dans leur propre dossier au moment de la compilation
- Avoir des variable pour les chemins, le nom de l'exécutable, etc..
- Des règles qui permettent de “nettoyer” les .o, lancer en mode debug, etc...

Et voilà

- On a donc un makefile qui compile avec notre project actuel, après il y a beaucoup de chose à améliorer, comme :
- Déplacer tous les fichiers .o dans leur propre dossier au moment de la compilation
- Avoir des variable pour les chemins, le nom de l'exécutable, etc..
- Des règles qui permettent de “nettoyer” les .o, lancer en mode debug, etc...
- *Plus de couleurs :)*

Conclusion

- Les headers permettent de transporter des definitions de fonctions dans des fichiers .c, pour permettre leur compilation. Il faut quand même donner les implémentations de ceux-ci au moment du linking

Conclusion

- Les headers permettent de transporter des definitions de fonctions dans des fichiers .c, pour permettre leur compilation. Il faut quand même donner les implémentations de ceux-ci au moment du linking
- On a vu 2 manières de include un header : “ <..> ” vs “ ".." ”

Conclusion

- Les headers permettent de transporter des definitions de fonctions dans des fichiers .c, pour permettre leur compilation. Il faut quand même donner les implémentations de ceux-ci au moment du linking
- On a vu 2 manières de include un header : “ <..> ” vs “ ".." ”
- <..> pour les headers contenu dans un dossier spécial du système

Conclusion

- Les headers permettent de transporter des definitions de fonctions dans des fichiers .c, pour permettre leur compilation. Il faut quand même donner les implémentations de ceux-ci au moment du linking
- On a vu 2 manières de include un header : “ <..> ” vs “ ".." ”
 - <..> pour les headers contenu dans un dossier spécial du système
 - ".." pour nos propres headers

Conclusion

- Les headers permettent de transporter des définitions de fonctions dans des fichiers .c, pour permettre leur compilation. Il faut quand même donner les implémentations de ceux-ci au moment du linking
- On a vu 2 manières de include un header : “ <..> ” vs “ ".." ”
 - <..> pour les headers contenu dans un dossier spécial du système
 - ".." pour nos propres headers
- Par défaut, la libc est **linker** au moment du linking par la majorité des compilateurs, il faut carrément ajouter -nolibc (à ne pas confondre avec -nostdlib)

Conclusion

- Dès qu'on dépasse les quelques fichiers, il serait mieux d'utiliser un build system, **make** est une solution.

Conclusion

- Dès qu'on dépasse les quelques fichiers, il serait mieux d'utiliser un build system, **make** est une solution.
- Se base sur un Makefile, on utilise des macros (variables) pour stocker les files, et utilise de règles ainsi que ses prérequis pour compiler efficacement un projet.

Conclusion

- Dès qu'on dépasse les quelques fichiers, il serait mieux d'utiliser un build system, **make** est une solution.
 - Se base sur un Makefile, on utilise des macros (variables) pour stocker les files, et utilise de règles ainsi que ses prérequis pour compiler efficacement un projet.
- « Il faut passer le moins de temps possible sur le build system, et le plus de temps possible à coder »

Merci de votre écoute

- Contacts
 - Matej Stehlik : <https://www.irif.fr/users/matej/index>
 - Iyan Nazarian : <https://github.com/Fxmouskid>
- Ressources
 - Code snippets : <https://chalk.ist/>
 - Slides : <https://www.libreoffice.org/>
 - Dessins : <https://notability.com/>