Cours PRG2

Programmation C

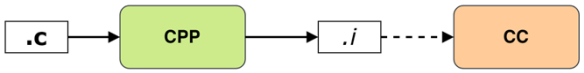
C8 *(2024_1)*

Contenu

- Préprocesseur C et directives
- Macros en C
- Programmation modulaire

Préprocesseur C et directives (1/6)

- Le préprocesseur est une application de la *toolchain*




```
gcc --save-temps input.c -DCONFIG_WAYLAND
```

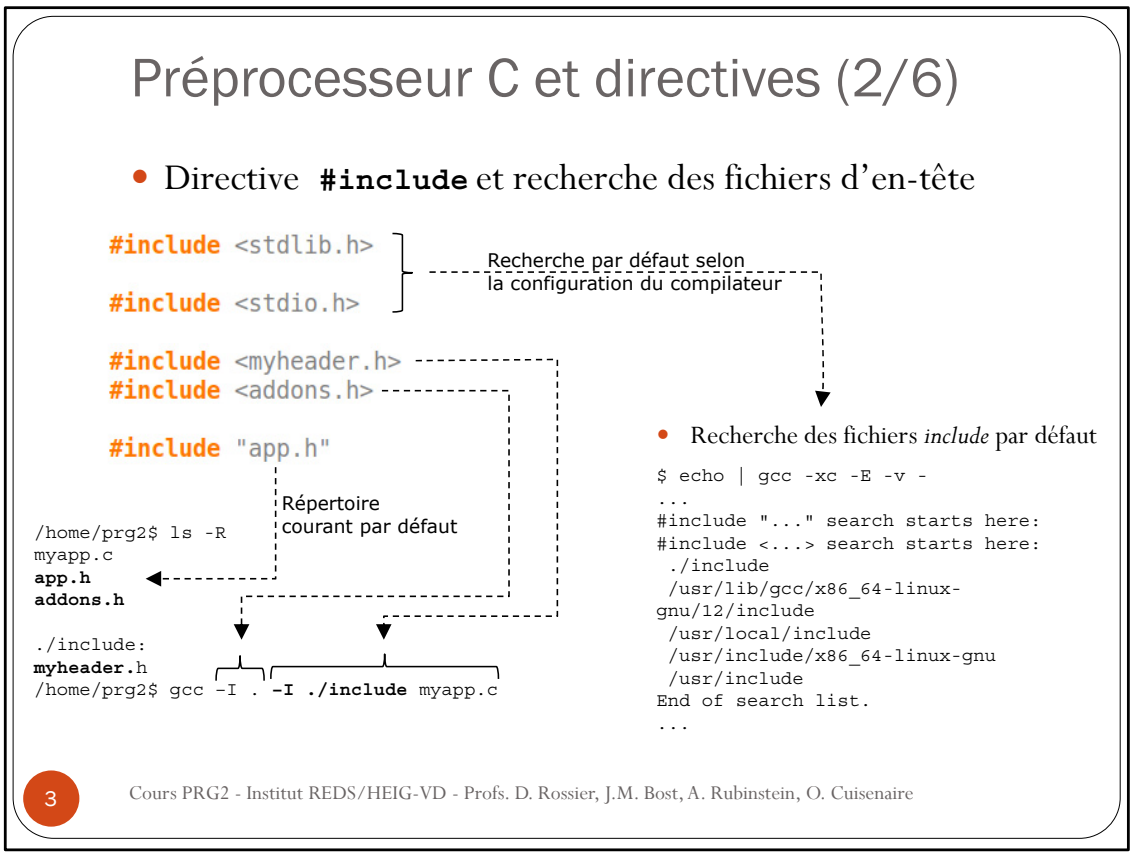
```c
#include <stddef.h>

#define NMAX    100

#ifdef CONFIG_WAYLAND

int wayland_init(void) {


    /* ... */
}
#endif

int main(void) {
    char array[NMAX];

  /* ... */
}
```

```c
# 1 "/usr/lib/gcc/x86_64-linux-gnu/12/include/stddef.h" 1 3 4
# 145 "/usr/lib/gcc/x86_64-linux-gnu/12/include/stddef.h" 3 4

# 145 "/usr/lib/gcc/x86_64-linux-gnu/12/include/stddef.h" 3 4
typedef long int ptrdiff_t;
# 214 "/usr/lib/gcc/x86_64-linux-gnu/12/include/stddef.h" 3 4
typedef long unsigned int size_t;
# 329 "/usr/lib/gcc/x86_64-linux-gnu/12/include/stddef.h" 3 4
typedef int wchar_t;
# 424 "/usr/lib/gcc/x86_64-linux-gnu/12/include/stddef.h" 3 4
typedef struct {
  long long __max_align_ll __attribute__((__aligned__(__alignof__(long long))));
  long double __max_align_ld __attribute__((__aligned__(__alignof__(long double))));
# 435 "/usr/lib/gcc/x86_64-linux-gnu/12/include/stddef.h" 3 4
} max_align_t;
# 4 "input.c" 2

# 9 "input.c"
int wayland_init(void) {

}

int main(void) {
  char array[100];

}
```


Le préprocesseur C dispose de plusieurs directives qui agissent sur le code source et le transforme en une nouvelle version qui sera transmise au compilateur.

Avec *gcc*, la production des fichiers intermédiaires peut être préservée à l'aide de l'option `-save-temps`. On remarque ainsi que l'effet de l'utilisation de la directive #define par exemple, qui remplace chaque occurrence du terme défini par le contenu tel que spécifié. **sans autre vérification**.

L'exemple ci-dessus montre qu'il est possible de définir un terme (avec ou sans valeur) au moment de l'invocation de la commande *gcc* grâce à l'option `-D` .



Le directive #include permet l'inclusion de fichiers d'en-tête contenant des définitions et/ou des déclarations.

La différence entre l'utilisation de `< >` et `" "` réside dans la manière dont le préprocesseur va rechercher les fichiers dans les différents sous-répertoires. La version avec les guillemets indique au préprocesseur de rechercher **d'abord** le fichier dans le **répertoire courant**. S'il ne trouve pas le fichier, il devra parcourir alors les répertoires indiqués avec l'option `-I`, comme le montre l'exemple ci-dessus, ainsi que les répertoires définis *par défaut* dans la *toolchain*.

L'utilisation des `< >` se réfèrent quant à eux toujours aux répertoires définis dans la ligne de commande et ceux de la *toolchain.* Bien sûr, il est possible d'indiquer le répertoire courant comme premier répertoire de l'option `-I` pour avoir un comportement similaire avec les guillemets.

Préprocesseur C et directives (3/6)

- Directive **`#define`** (macro)
- Directive **`#ifdef`** pour la compilation conditionnelle

Constantes (traité par le préprocesseur)

Existence de la macro DEBUG
Non-existence de la macro VERBOSE

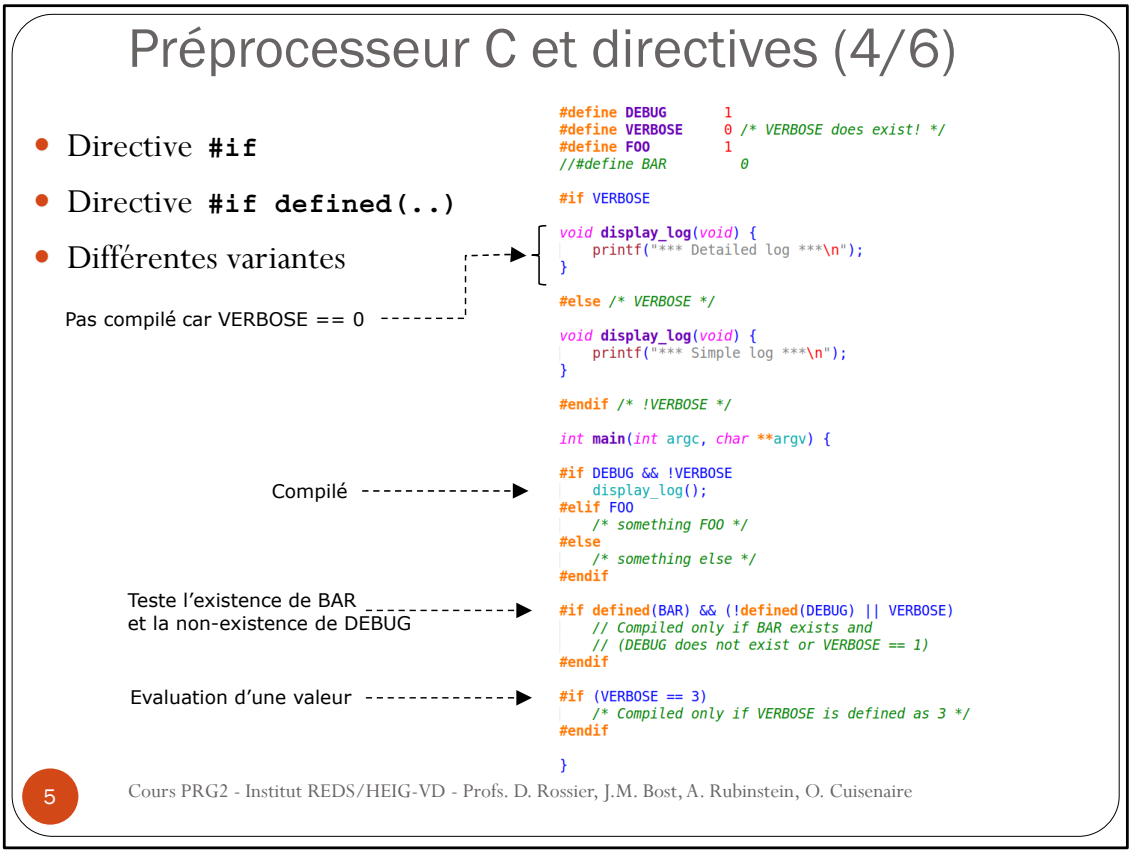
Compilation conditionnelle

```
#ifdef symbole
    partie-du-programme-1
#else condition-2
    partie-du-programme-2
#endif

#ifndef symbole
    partie-du-programme-1
#else condition-2
    partie-du-programme-2
#endif
```

```c
#define NR_ITEMS    100

#define DEBUG
// #define VERBOSE

#ifdef VERBOSE

void display_log(void) {

    printf("*** Detailed log ***\n");
    /* ... */
}

#else /* VERBOSE */

void display_log(void) {

    printf("*** Simple log ***\n");
    /* ... */
}

#endif /* !VERBOSE */

int main(int argc, char **argv) {

    int tab[NR_ITEMS];

    /* ... */

#ifdef DEBUG

    display_log();

#endif /* DEBUG */

}
```


La compilation conditionnelle permet de définir quelle portion de code doit être compilée sous certaines conditions qui peuvent être traitées avec des #define et la directive `#ifdef ... #endif`. Cette directive est utilisée pour tester si un terme existe ou non, quelque soit sa définition.

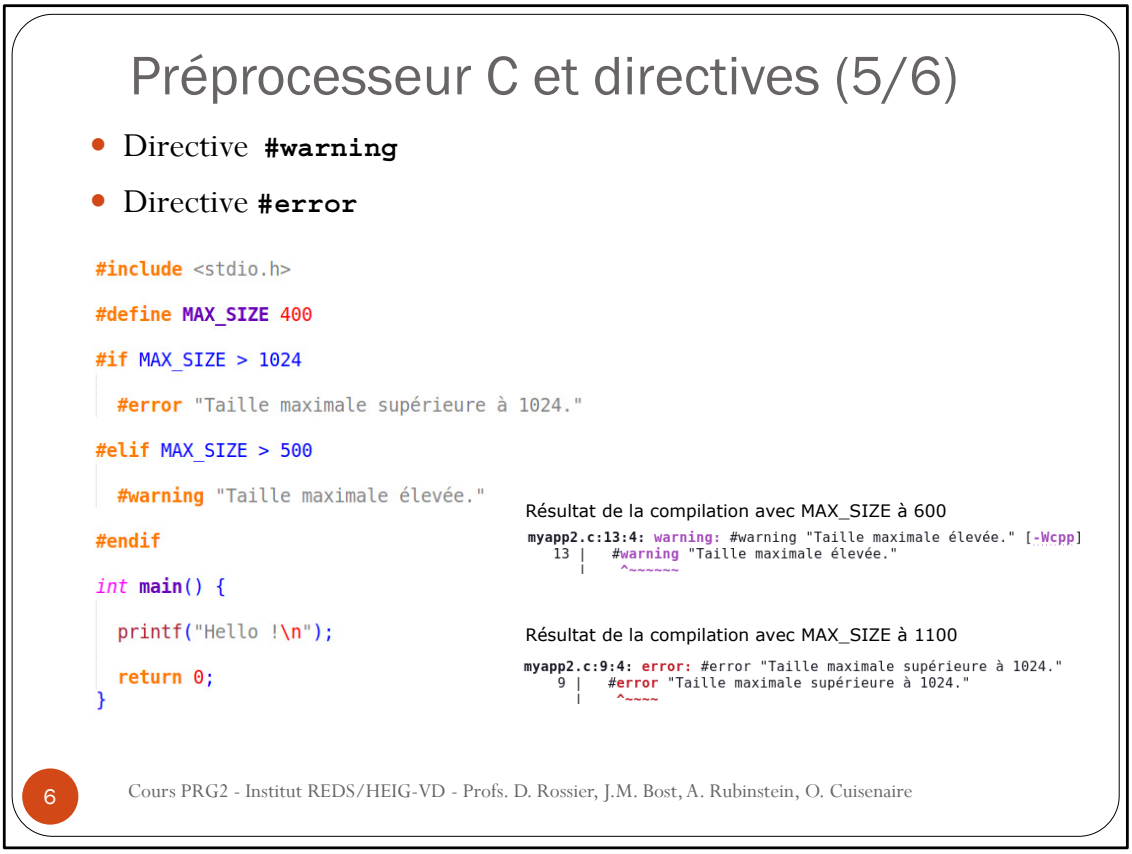
Tout comme un `if` traditionnel, la directive `#else` peut être utilisé pour gérer la condition.



La directive `#if` est plus élaboré que `#ifdef` car elle permet de tester des valeurs et d'être utilisée également pour tester l'existence d'une définition.

Le test d'existence s'effectue à l'aide de `defined(..)` comme le montre l'exemple ci-dessus.

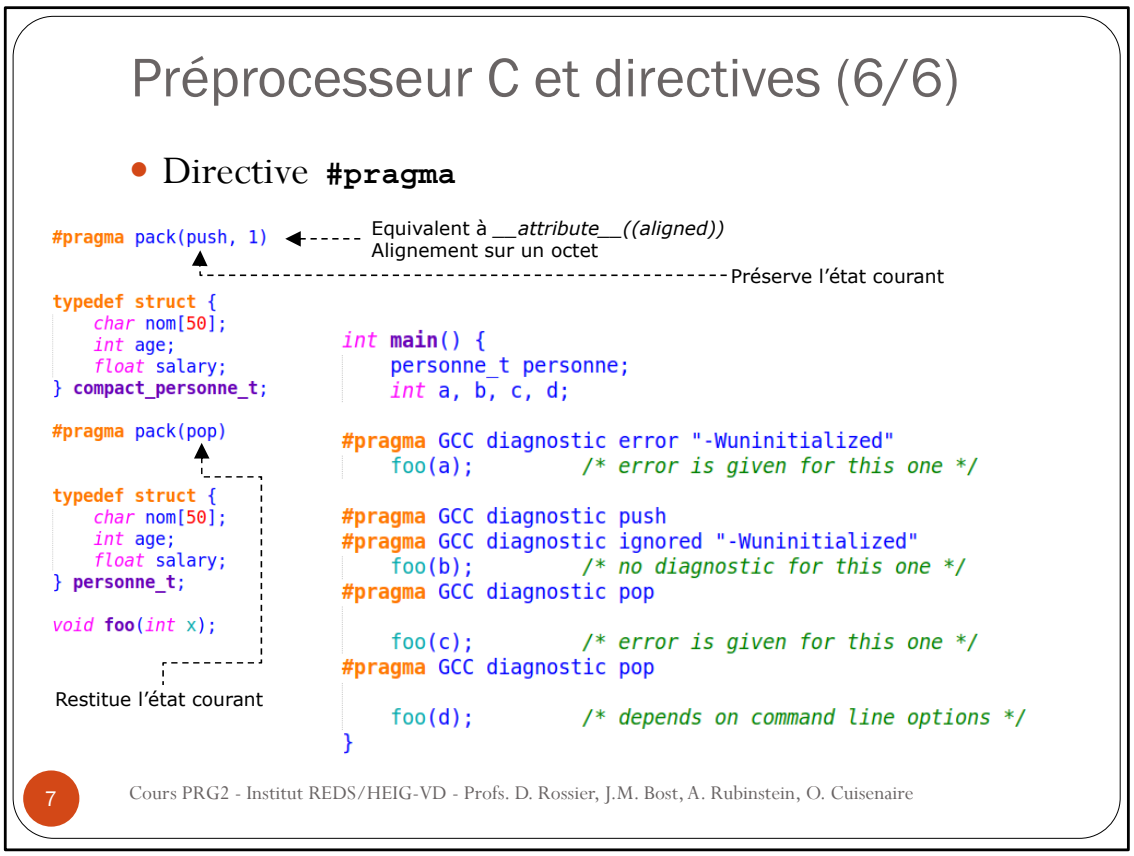
Par ailleurs, il est possible d'enchaîner une cascade de if grâce à `elif`.



La directive `#warning` est une fonctionnalité utile du préprocesseur C qui permet de générer un avertissement au moment de la compilation. Son objectif principal est d'attirer l'attention du développeur sur des informations importantes ou potentiellement problématiques concernant le code en cours de compilation.

Dans le contexte de configurations multiples avec des variables de configuration définies à l'aide de `#define`, la directive `#warning` est souvent utilisée pour informer le développeur sur les conditions de compilation et sur les choix de configuration.

Comparé à `#warning`, la directive `#error` est également une fonctionnalité utile du préprocesseur C, mais avec une fonctionnalité légèrement différente. Alors que `#warning` génère un avertissement lors de la compilation, `#error` génère une erreur qui interrompt la compilation et empêche la génération du fichier exécutable.

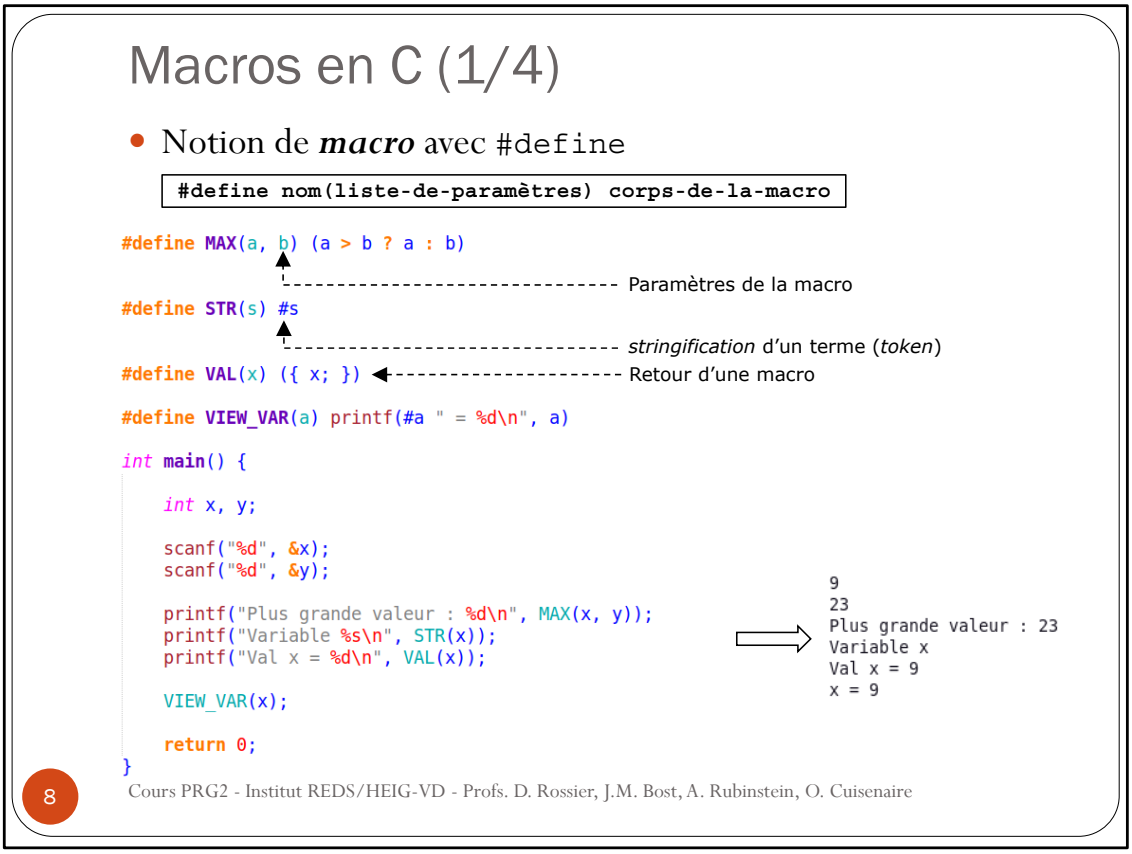


Les directives `#pragma` sont des instructions spéciales utilisées dans le langage C (et dans d'autres langages) pour contrôler le comportement du compilateur sur des parties spécifiques du code source. Elles sont conçues pour fournir des instructions au compilateur sur la manière de traiter certaines sections de code, tout en restant indépendantes du langage C lui-même.

Elles ne font pas partie du langage C standard, mais sont plutôt des extensions spécifiques à chaque compilateur. Par conséquent, leur comportement peut varier entre les différents compilateurs C.

L'exemple ci-dessus montre deux exemples d'utilisation dans le contexte de l'alignement des données et des options du compilateurs.

Il existe un mécanisme de `push/pop` permettant de préserver un état courant et de le restituer à tout moment dans le code.

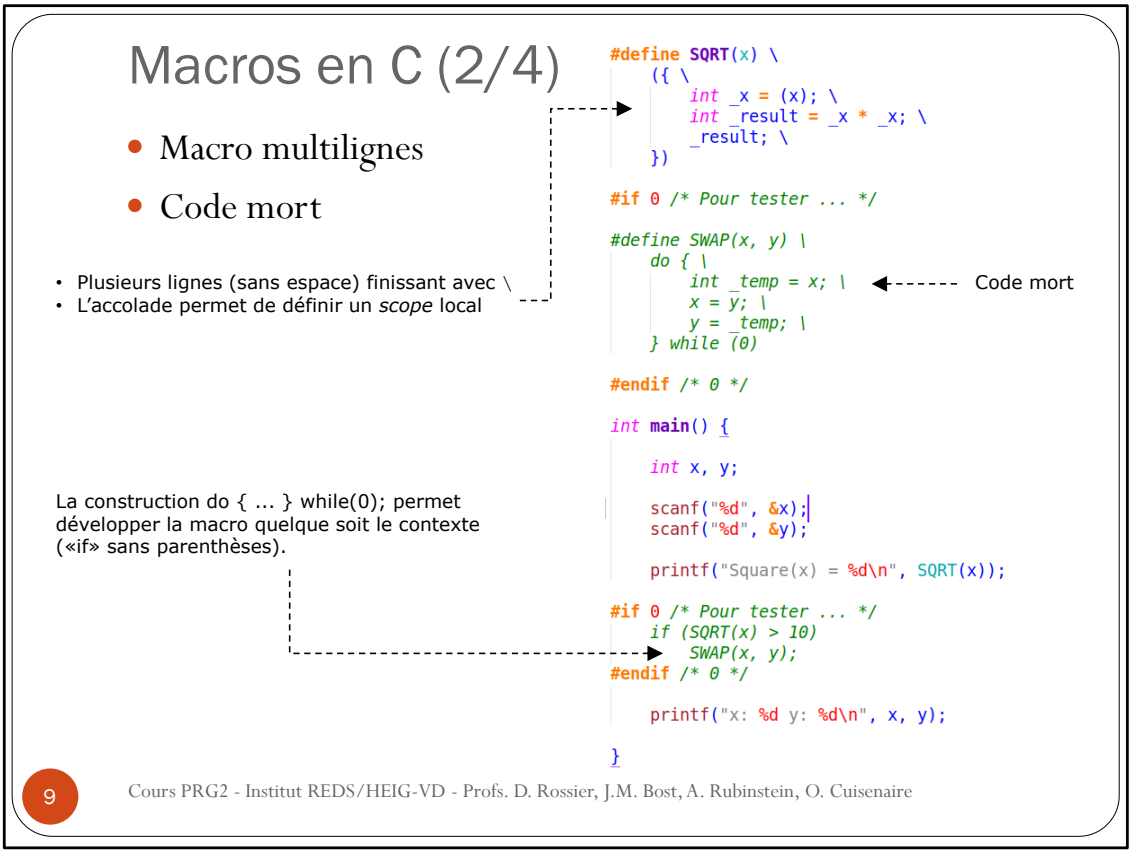


Une macro est définie à l'aide de la directive préprocesseur `#define`. Cette directive permet de définir un nom symbolique (appelé identificateur de macro) associé à une valeur ou à un fragment de code.

Comme nous l'avons déjà évoqué précédemment, lorsque le préprocesseur rencontre une utilisation de la macro définie (`#define`), il effectue une substitution textuelle. Cela signifie que l'identificateur de la macro est remplacé par son contenu dans tout le code source avant la compilation effective.

Les macros peuvent également être définies avec des paramètres. Cela permet de créer des macros paramétrées qui peuvent générer différents morceaux de code en fonction des valeurs passées.

Lorsqu'une macro avec # est utilisée, le paramètre qui suit # est converti en **une chaîne de caractères entre guillemets**. Ceci est utile lorsque l'on souhaite transformer le nom d'une variable ou d'un autre identificateur en une chaîne de texte pour une utilisation ultérieure dans le code, comme le montre l'exemple ci-dessus.



Une macro peut s'étendre sur plusieurs lignes. Dans ce cas, chaque ligne (à l'exception de la dernière) doit se terminer par `\`. Il n'est pas permis d'avoir des espaces entre les lignes.

L'utilisation de `({ .. })` permet de définir des variables locales visiblement uniquement à l'intérieur des accolades. Cette construction permet également à la macro de retourner une valeur en la mettant sous forme d'une instruction terminée par un point-virgule.

L'exemple ci-dessus montre également la technique préférée pour désactiver du code à la compilation (code mort) en utilisation `#if 0 ... #endif /* 0 */`. C'est une pratique courante permettant de faciliter l'utilisation de code de *debug* ou de validation. Il suffit de passer la valeur à 1 afin de ré-activer le code.
Cette technique est de loin préférable à l'utilisation de `/* ... */` ou pire encore de //.

Macros en C (3/4)

- Concaténation de termes (*token*) dans une macro

Concatenation de deux *tokens* avec ## (dont le second *token* est passé en paramètre

Développement de 3 fonctions de nom différent

Généré par le préprocesseur

```c
#include <stdio.h>

// Macro pour déclarer une fonction d'impression générique
#define PRINT_FUNCTION(type, format) \
    void print_##type(type value) { \
        printf("Value: " format "\n", value); \
    }

// Mise ne place des fonctions pour différents types
PRINT_FUNCTION(int, "%d")
PRINT_FUNCTION(float, "%f")
PRINT_FUNCTION(char, "%c")

int main() {
    int intValue = 10;
    float floatValue = 3.14;
    char charValue = 'A';

    // Appels aux fonctions générées par la macro
    print_int(intValue);
    print_float(floatValue);
    print_char(charValue);

    return 0;
}
```

```c
void print_int(int value) {
    printf("Value: " "%d" "\n", value);
}
void print_float(float value) {
    printf("Value: " "%f" "\n", value);
}
void print_char(char value) {
    printf("Value: " "%c" "\n", value);
}
```


le symbole ## dans une macro en langage C permet la concaténation de termes *(tokens*) permettant de combiner dynamiquement des fragments de code pour former des identificateurs ou des expressions valides pendant la phase de pré-compilation. Cette fonctionnalité peut être utile dans certains cas, mais elle doit être utilisée avec précaution pour maintenir la lisibilité et la clarté du code.

Macros en C (4/4)

- Macros prédéfinies
  - `__func__` (ou `__FUNC__`), `__LINE__`, `__FILE__`

```c
#include <stdio.h>

void debugPrint(const char *message, const char *file, const char *func, int line) {
    printf("DEBUG: %s\n", message);
    printf("File: %s\n", file);
    printf("Function: %s\n", func);
    printf("Line: %d\n", line);
}

#define DEBUG_PRINT(msg) debugPrint(msg, __FILE__, __func__, __LINE__)

int main() {
    int x = 10;
    DEBUG_PRINT("Value of x is being printed");

    for (int i = 0; i < 5; i++) {
        DEBUG_PRINT("Inside loop");
        printf("Iteration: %d\n", i);
    }

    return 0;
}
```

```
DEBUG: Value of x is being printed
File: macro_3.c
Function: main
Line: 15
DEBUG: Inside loop
File: macro_3.c
Function: main
Line: 18
Iteration: 0
DEBUG: Inside loop
File: macro_3.c
Function: main
Line: 18
Iteration: 1
DEBUG: Inside loop
File: macro_3.c
Function: main
Line: 18
Iteration: 2
DEBUG: Inside loop
File: macro_3.c
Function: main
Line: 18
Iteration: 3
DEBUG: Inside loop
File: macro_3.c
Function: main
Line: 18
Iteration: 4
```


Les macros ci-dessus sont prédéfinis par le préprocesseur. Elles sont très pratiques lors du *debug* de code.

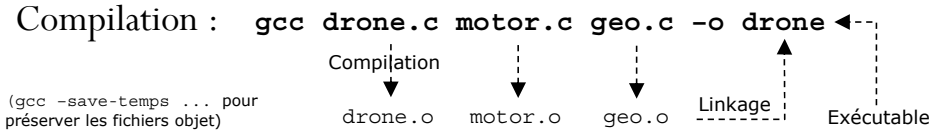
Programmation modulaire (1/6)

- Une application est généralement composée de plusieurs fichiers `.c` et fichiers `.h` (*include)*

- **Pas d'implémentation de fonction** dans les fichiers d'en-tête (*include*)
- Exemple :

```
motor.h   geo.h   drone.h
motor.c   geo.c   drone.c
```

- Compilation : `gcc drone.c motor.c geo.c -o drone`



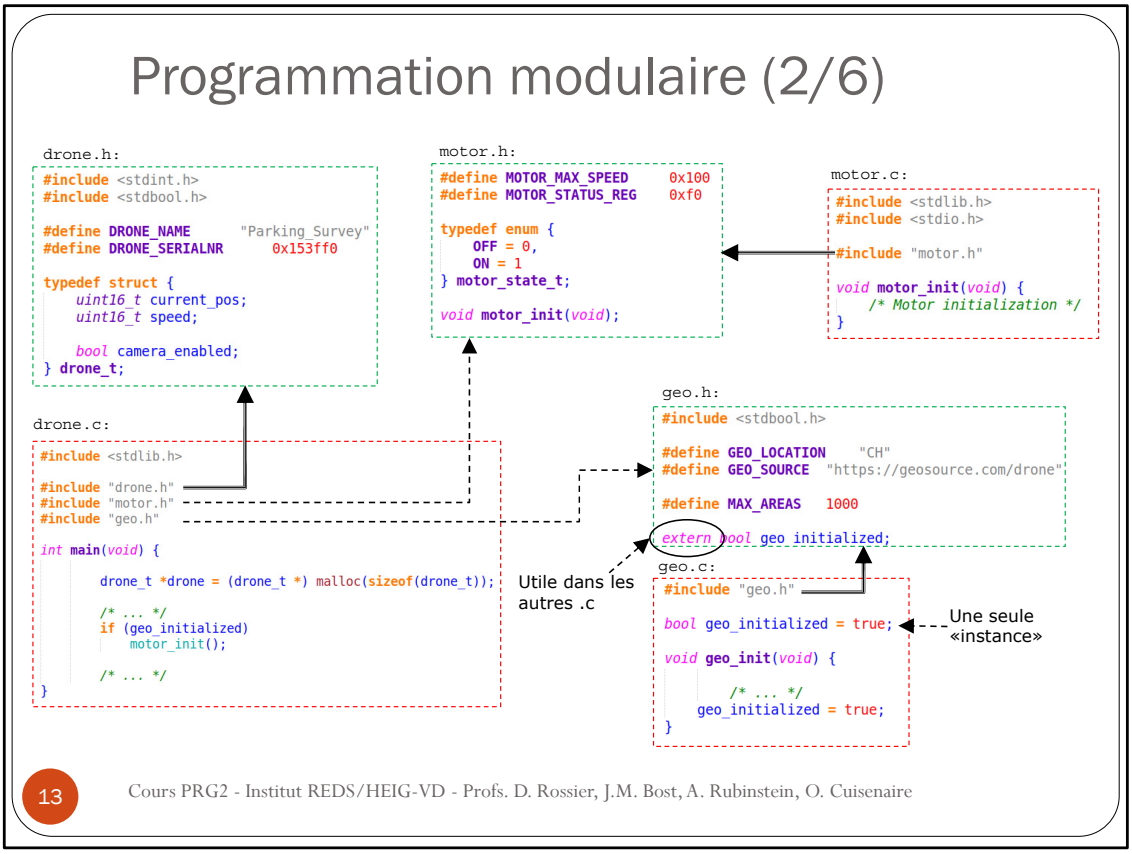
(gcc -save-temps ... pour préserver les fichiers objet)

La programmation modulaire en langage C est une approche de conception qui consiste à diviser un programme en **modules distincts et autonomes**, chacun ayant une fonctionnalité ou une responsabilité spécifique. Chaque module est représenté par un fichier source (`.c`) associé à un fichier d'en-tête (`.h`) qui déclare les interfaces publiques du module. Cette approche favorise la réutilisabilité du code, la facilité de maintenance et la gestion de la complexité des grands systèmes logiciels.

La **gestion des dépendances** est également essentielle, car elle permet de minimiser le couplage entre les modules. Une conception modulaire bien réalisée vise à réduire le couplage tout en favorisant la cohésion, où chaque module a une tâche spécifique et bien définie.

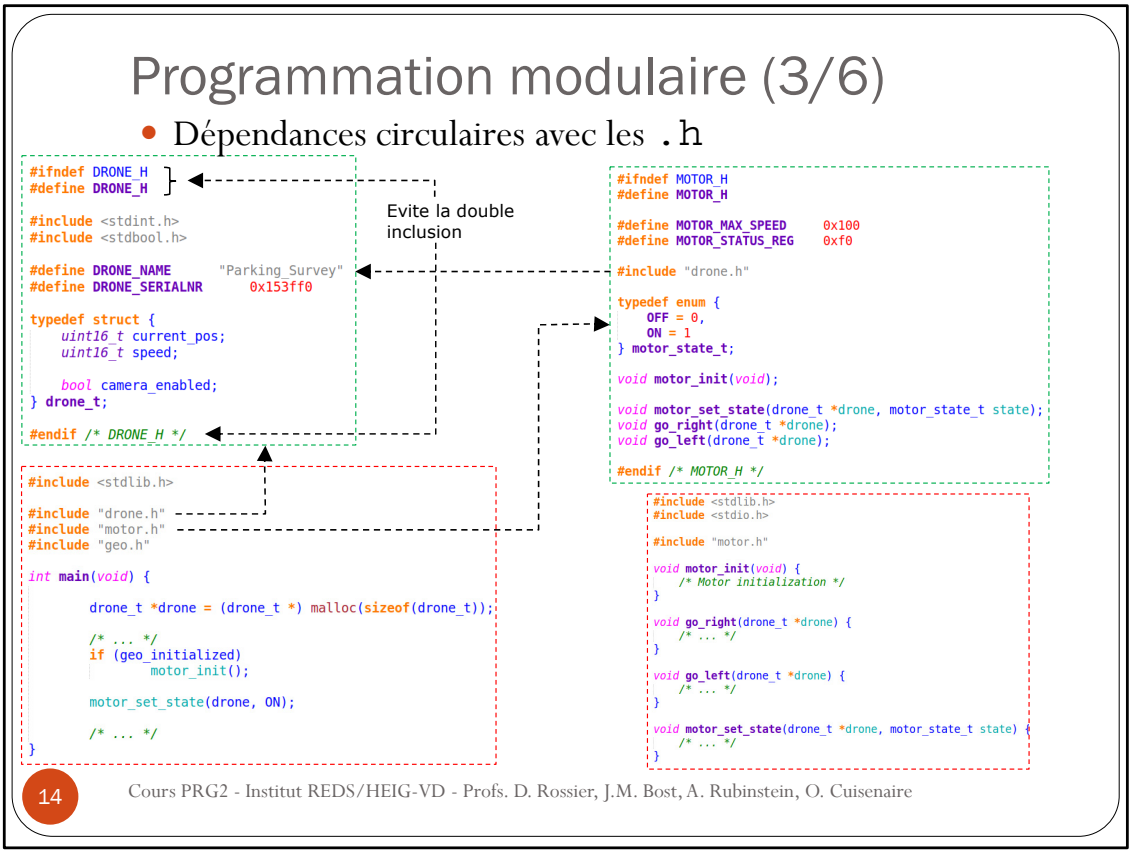
La programmation modulaire encourage également la **réutilisabilité** du code, car les modules bien conçus peuvent être utilisés dans d'autres projets sans modification. Cela améliore la productivité et facilite la maintenance, car les modifications apportées à un module **n'affectent pas les autres modules, tant que les interfaces publiques restent inchangées**.

La programmation modulaire est une approche puissante pour développer des logiciels **robustes** et **évolutifs**. Elle favorise la séparation des préoccupations, la réutilisabilité du code et la gestion efficace de la complexité, ce qui en fait une pratique courante dans le développement de systèmes logiciels de qualité.



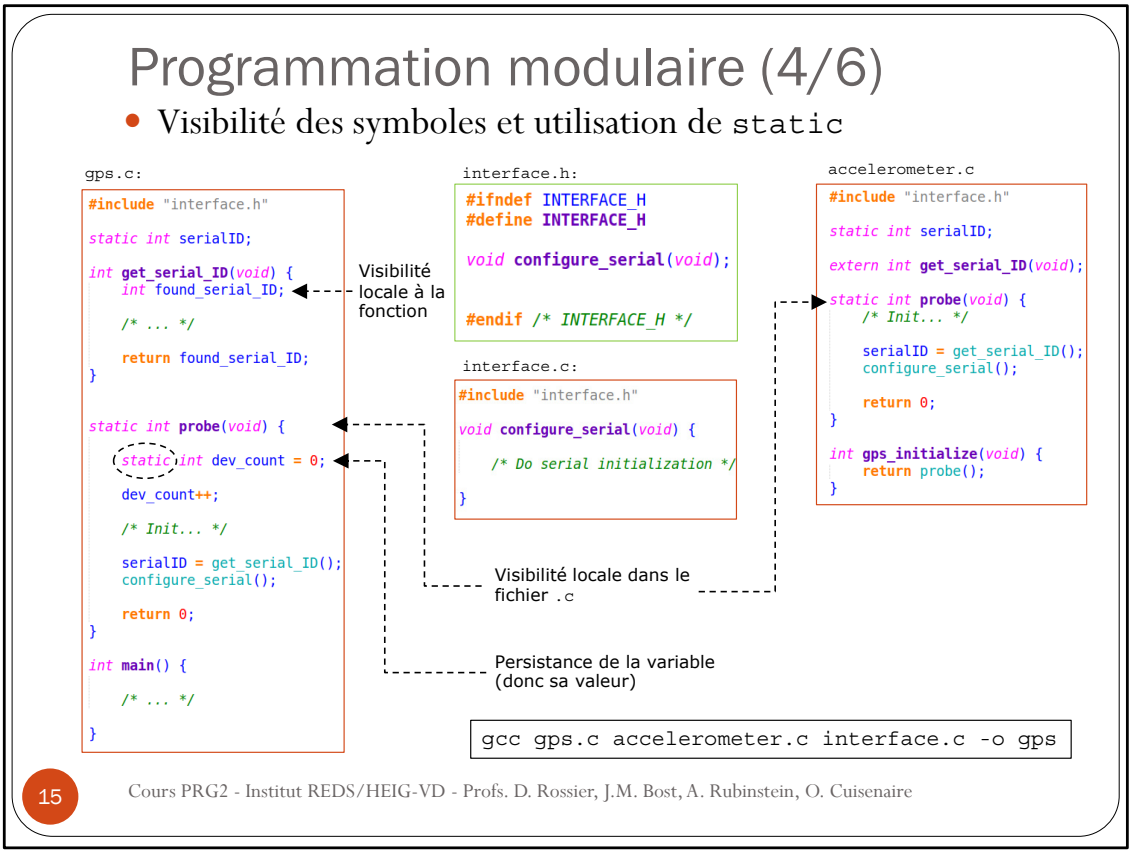
L'exemple ci-dessus illustre l'utilisation de plusieurs fichiers sources et d'en-tête permettant de bien séparer les fonctionnalités de l'application. Le contenu des fichiers d'en-tête se limite à des définitions de constantes et des prototypes de fonctions. Le corps de la fonction doit impérativement se trouver dans un fichier `.c`.

L'accès à un symbole (variable, fonction) défini dans une autre fichier `.c` nécessite l'utilisation de la particule `extern`. Comme le montre le fichier `motor.h`, la déclaration d'une fonction sans le corps de celle-ci est équivalent à une déclaration avec `extern`.
Cela peut être également le cas pour une déclaration de `struct`. En revanche, la déclaration d'une variable globale ne peut se faire que dans un seul module. Par conséquent, la variable est déclarée `extern` dans le fichier d'en-tête et sa déclaration effective apparaît dans l'un des fichiers `.c`.



Il arrive fréquemment qu'un fichier `.c` inclut plusieurs fichiers d'en-tête qui eux-mêmes incluent d'autres fichiers d'en-tête. Par conséquent, des définitions peuvent apparaître plusieurs fois et conduisent, bien qu'identiques, à des erreurs de compilation.

L'utilisation de la construction présentée ci-dessus permet d'éviter une référence circulaire entre fichiers d'en-tête.



Le mot-clé `static` peut être utilisé pour deux raisons différentes : soit pour donner la visibilité à un symbole (variable, fonction) uniquement dans le fichier courant (proche de la notion de symbole ***privé***), soit pour stipuler qu'une variable locale est persistante, i,e. son allocation ne disparaît pas au terme de l'exécution de la fonction. Une telle variable n'est alors pas allouée sur la pile, mais dans une autre région mémoire réservée aux variables persistantes (section *data*). De telles variables existent jusqu'à la fin du programme.

Programmation modulaire (5/6)

- Fonction **`inline`**
- Amélioration des performances

```c
#include <stdio.h>

int add(int a, int b);

// Définition d'une fonction inline
inline int add(int a, int b) {
    return a + b;
}

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

    // Appel de la fonction inline
    result = add(x, y);

    printf("Addition: %d\n", result);

    return 0;
}
```

Pas forcément nécessaire (dépend du compilateur)

Remplacement lors de la compilation

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

    // Appel de la fonction inline
    result = 5 + 3;

    printf("Addition: %d\n", result);

    return 0;
}
```


Une **fonction `inline`** est une fonction dont le corps est **inséré directement à l'endroit de son appel**. En d'autres termes, lorsqu'elle est utilisée, l'exécution se poursuit **sans qu'il y ait un appel de fonction distinct**. Cela permet de réduire le temps d'exécution, car il n'y a pas de préparation à l'entrée et à la sortie de la fonction. En somme, les fonctions inline sont optimisées pour minimiser les frais généraux liés aux appels de fonction.

Il est cependant plus difficile de *debugger* de tel programme car la référence à la fonction `inline` n'existe plus à l'exécution.

Programmation modulaire (6/6)

- Exemple avec plusieurs unités de compilation

gps.c:

```c
#include "interface.h"

static int serialID;

int get_serial_ID(void) {
    int found_serial_ID;

    /* ... */

    return found_serial_ID;
}


static int probe(void) {
    static int dev_count = 0;

    dev_count++;

    /* Init... */

    serialID = get_serial_ID();
    configure_serial();

    return 0;
}

int main() {

    /* ... */

}
```

interface.h:

```c
#ifndef INTERFACE_H
#define INTERFACE_H

static inline void configure_serial(void) {

    /* Do serial initialization */

}


#endif /* INTERFACE_H */
```

accelerometer.c

```c
#include "interface.h"

static int serialID;

extern int get_serial_ID(void);

static int probe(void) {
    /* Init... */

    serialID = get_serial_ID();
    configure_serial();

    return 0;
}

int gps_initialize(void) {
    return probe();
}
```

```
gcc gps.c accelerometer.c -o gps
```

Sans la particule `static`, on obtient une erreur de références multiples

```
/usr/bin/ld: /tmp/cccP5RQK.o: in function `configure_serial':
accelerometer.c:(.text+0x0): multiple definition of `configure_serial'; /tmp/ccOz3bcz.o:gps.c:(.text+0x0): first defined here
collect2: error: ld returned 1 exit status
```


Malgré que la fonction ne soit pas appelée en tant que telle, et qu'on peut imaginer une copie de la fonction partout où celle-ci est appelé (avant la compilation), le compilateur préserve toujours la cohérence des prototypes entre les unités de compilation différentes. C'est pourquoi une déclaration de fonction reste présente et **impose l'utilisation de `static`** si la fonction `inline` est définie dans un `.h` et utilisée dans plusieurs fichiers `.c` différents.