

Langage C – Mise à niveau

Yves Legrandgérard
ylg@irif.fr

Laboratoire IRIF - Université Paris Diderot

Novembre 2018

Table des Matières I

1 Principes de programmation

- Structure d'un programme C
- Compilation – Édition de liens – Débogage
- Le langage C en pratique

2 Pointeurs et Tableaux

- Pointeurs
- Tableaux

3 Pointeurs et structures

- Un peu de syntaxe
- Structures pseudo-récurrentes

4 Quelques compléments

- Les opérateurs logiques
- La classe de variables `static`

5 Pointeurs de fonctions

- Introduction

Table des Matières II

- Hachage cryptographique

6 Listes

- Introduction
- Fonctions de base
- Listes d'entiers
- Optimisation
- Extension de la bibliothèque

7 Outils de développement

- Le logiciel GNU make
- Une bibliothèque de gestion d'erreurs
- Le logiciel GNU autoconf

Généralités

- Un programme C est un **ensemble de fonctions** parmi lesquelles il y a une fonction qui est le **point d'entrée principal** du programme : l'exécution du programme commence par l'exécution de cette fonction qui appelle successivement les éventuelles autres fonctions du programme.
- Cette fonction a pour nom : `main`. Elle peut avoir pour prototype `int main(int, char **)` ou `int main(void)` suivant que le programme prend ou pas des arguments en entrée.
- Le programme C minimal, et ne prenant pas d'arguments, a ainsi pour code :

```
int main(void) {return 0;}
```

- L'entier retourné par la fonction `main` est par convention 0 si le programme s'est déroulé sans erreur et $\neq 0$ sinon. Cette valeur peut être récupérée au niveau du *shell* avec la commande : `echo $?`.

Généralités

- Un programme C est un **ensemble de fonctions** parmi lesquelles il y a une fonction qui est le **point d'entrée principal** du programme : l'exécution du programme commence par l'exécution de cette fonction qui appelle successivement les éventuelles autres fonctions du programme.
- Cette fonction a pour nom : `main`. Elle peut avoir pour prototype `int main(int, char **)` ou `int main(void)` suivant que le programme prend ou pas des arguments en entrée.
- Le programme C minimal, et ne prenant pas d'arguments, a ainsi pour code :

```
int main(void) {return 0;}
```

- L'entier retourné par la fonction `main` est par convention 0 si le programme s'est déroulé sans erreur et $\neq 0$ sinon. Cette valeur peut être récupérée au niveau du *shell* avec la commande : `echo $?`.

Généralités

- Un programme C est un **ensemble de fonctions** parmi lesquelles il y a une fonction qui est le **point d'entrée principal** du programme : l'exécution du programme commence par l'exécution de cette fonction qui appelle successivement les éventuelles autres fonctions du programme.
- Cette fonction a pour nom : `main`. Elle peut avoir pour prototype `int main(int, char **)` ou `int main(void)` suivant que le programme prend ou pas des arguments en entrée.
- Le programme C minimal, et ne prenant pas d'arguments, a ainsi pour code :

```
int main(void) {return 0;}
```

- L'entier retourné par la fonction `main` est par convention 0 si le programme s'est déroulé sans erreur et $\neq 0$ sinon. Cette valeur peut être récupérée au niveau du *shell* avec la commande : `echo $?`.

Généralités

- Un programme C est un **ensemble de fonctions** parmi lesquelles il y a une fonction qui est le **point d'entrée principal** du programme : l'exécution du programme commence par l'exécution de cette fonction qui appelle successivement les éventuelles autres fonctions du programme.
- Cette fonction a pour nom : `main`. Elle peut avoir pour prototype `int main(int, char **)` ou `int main(void)` suivant que le programme prend ou pas des arguments en entrée.
- Le programme C minimal, et ne prenant pas d'arguments, a ainsi pour code :

```
int main(void) {return 0;}
```

- L'entier retourné par la fonction `main` est par convention 0 si le programme s'est déroulé sans erreur et $\neq 0$ sinon. Cette valeur peut être récupérée au niveau du *shell* avec la commande : `echo $?`.

Programmation modulaire

- Un programme C, dès qu'il devient un peu conséquent (disons au delà de 100 lignes pour fixer les idées) devra **impérativement** être scindé en plusieurs fichiers sources appelés **modules**.
- Bien entendu, un (et un seul) de ces fichiers sources contiendra la fonction `main`.
- Chaque module sera composé d'un fichier en-tête `module.h` (ou fichier **interface**) et d'un fichier source `module.c` (ou fichier **implémentation**).
- Un principe **fondamental** : toujours commencer par coder l'interface car c'est la partie **publique** de votre module (pouvant être utilisé par d'autres programmes). Elle doit être soigneusement pensée car elle ne pourra pas être aisément modifiée ultérieurement (songez aux primitives système comme `printf` dont le prototype ne pourrait être modifié sans casser la plupart des programmes C).
- En revanche, l'implémentation est modifiable sans problème (on a un algorithme plus performant par exemple).

Programmation modulaire

- Un programme C, dès qu'il devient un peu conséquent (disons au delà de 100 lignes pour fixer les idées) devra **impérativement** être scindé en plusieurs fichiers sources appelés **modules**.
- Bien entendu, un (et un seul) de ces fichiers sources contiendra la fonction `main`.
- Chaque module sera composé d'un fichier en-tête `module.h` (ou fichier **interface**) et d'un fichier source `module.c` (ou fichier **implémentation**).
- Un principe **fondamental** : toujours commencer par coder l'interface car c'est la partie **publique** de votre module (pouvant être utilisé par d'autres programmes). Elle doit être soigneusement pensée car elle ne pourra pas être aisément modifiée ultérieurement (songez aux primitives système comme `printf` dont le prototype ne pourrait être modifié sans casser la plupart des programmes C).
- En revanche, l'implémentation est modifiable sans problème (on a un algorithme plus performant par exemple).

Programmation modulaire

- Un programme C, dès qu'il devient un peu conséquent (disons au delà de 100 lignes pour fixer les idées) devra **impérativement** être scindé en plusieurs fichiers sources appelés **modules**.
- Bien entendu, un (et un seul) de ces fichiers sources contiendra la fonction `main`.
- Chaque module sera composé d'un fichier en-tête `module.h` (ou fichier **interface**) et d'un fichier source `module.c` (ou fichier **implémentation**).
- Un principe **fondamental** : toujours commencer par coder l'interface car c'est la partie **publique** de votre module (pouvant être utilisé par d'autres programmes). Elle doit être soigneusement pensée car elle ne pourra pas être aisément modifiée ultérieurement (songez aux primitives système comme `printf` dont le prototype ne pourrait être modifié sans casser la plupart des programmes C).
- En revanche, l'implémentation est modifiable sans problème (on a un algorithme plus performant par exemple).

Programmation modulaire

- Un programme C, dès qu'il devient un peu conséquent (disons au delà de 100 lignes pour fixer les idées) devra **impérativement** être scindé en plusieurs fichiers sources appelés **modules**.
- Bien entendu, un (et un seul) de ces fichiers sources contiendra la fonction `main`.
- Chaque module sera composé d'un fichier en-tête `module.h` (ou fichier **interface**) et d'un fichier source `module.c` (ou fichier **implémentation**).
- Un principe **fondamental** : toujours commencer par coder l'interface car c'est la partie **publique** de votre module (pouvant être utilisé par d'autres programmes). Elle doit être soigneusement pensée car elle ne pourra pas être aisément modifiée ultérieurement (songez aux primitives système comme `printf` dont le prototype ne pourrait être modifié sans casser la plupart des programmes C).
- En revanche, l'implémentation est modifiable sans problème (on a un algorithme plus performant par exemple).

Programmation modulaire

- Un programme C, dès qu'il devient un peu conséquent (disons au delà de 100 lignes pour fixer les idées) devra **impérativement** être scindé en plusieurs fichiers sources appelés **modules**.
- Bien entendu, un (et un seul) de ces fichiers sources contiendra la fonction `main`.
- Chaque module sera composé d'un fichier en-tête `module.h` (ou fichier **interface**) et d'un fichier source `module.c` (ou fichier **implémentation**).
- Un principe **fondamental** : toujours commencer par coder l'interface car c'est la partie **publique** de votre module (pouvant être utilisé par d'autres programmes). Elle doit être soigneusement pensée car elle ne pourra pas être aisément modifiée ultérieurement (songez aux primitives système comme `printf` dont le prototype ne pourrait être modifié sans casser la plupart des programmes C).
- En revanche, l'implémentation est modifiable sans problème (on a un algorithme plus performant par exemple).

Programmation modulaire – Un exemple (1/5)

- On veut créer une bibliothèque de gestion de polynômes à coefficients dans le corps fini $\mathbb{F}_2 = \{0, 1\}$.
- On va, bien entendu, commencer par l'interface à savoir le fichier `poly.h`. La première chose à faire est de définir la **structure de données** qui va représenter un tel polynôme.
- Les coefficients d'un polynôme sur \mathbb{F}_2 peuvent être codés sur un bit puisqu'ils valent 0 ou 1. Donc si, pour simplifier, on se limite aux polynômes P tels que $\deg(P) \leq 31$, on peut coder un polynôme avec un entier non signé de 32 bits :

fichier en-tête poly.h

```
#include <inttypes.h>

typedef uint32_t poly_t;
```

Programmation modulaire – Un exemple (1/5)

- On veut créer une bibliothèque de gestion de polynômes à coefficients dans le corps fini $\mathbb{F}_2 = \{0, 1\}$.
- On va, bien entendu, commencer par l'interface à savoir le fichier `poly.h`. La première chose à faire est de définir la **structure de données** qui va représenter un tel polynôme.
- Les coefficients d'un polynôme sur \mathbb{F}_2 peuvent être codés sur un bit puisqu'ils valent 0 ou 1. Donc si, pour simplifier, on se limite aux polynômes P tels que $\deg(P) \leq 31$, on peut coder un polynôme avec un entier non signé de 32 bits :

fichier en-tête poly.h

```
#include <inttypes.h>

typedef uint32_t poly_t;
```

Programmation modulaire – Un exemple (1/5)

- On veut créer une bibliothèque de gestion de polynômes à coefficients dans le corps fini $\mathbb{F}_2 = \{0, 1\}$.
- On va, bien entendu, commencer par l'interface à savoir le fichier `poly.h`. La première chose à faire est de définir la **structure de données** qui va représenter un tel polynôme.
- Les coefficients d'un polynôme sur \mathbb{F}_2 peuvent être codés sur un bit puisqu'ils valent 0 ou 1. Donc si, pour simplifier, on se limite aux polynômes P tels que $\deg(P) \leq 31$, on peut coder un polynôme avec un entier non signé de 32 bits :

fichier en-tête `poly.h`

```
#include <inttypes.h>

typedef uint32_t poly_t;
```

Programmation modulaire – Un exemple (2/5)

- **Attention** : il faut éviter d'**inclure plusieurs fois** un même fichier en-tête dans un programme car cela risque de générer des erreurs à la compilation. A priori cela paraît difficile à réaliser si notre programme est composé de beaucoup de modules imbriqués.
- Fort heureusement, il existe une solution simple au problème d'inclusions multiples :

fichier en-tête poly.h

```
#ifndef POLY_H /* on remplace les . par des _ */
#define POLY_H
#include <inttypes.h>

typedef uint32_t poly_t;

#endif /* POLY_H */
```

Il suffit donc d'encadrer le source de **chaque fichier en-tête** par ces trois directives *préprocesseur*.

Programmation modulaire – Un exemple (2/5)

- **Attention** : il faut éviter d'**inclure plusieurs fois** un même fichier en-tête dans un programme car cela risque de générer des erreurs à la compilation. A priori cela paraît difficile à réaliser si notre programme est composé de beaucoup de modules imbriqués.
- Fort heureusement, il existe une solution simple au problème d'inclusions multiples :

fichier en-tête poly.h

```
#ifndef POLY_H /* on remplace les . par des _ */
#define POLY_H
#include <inttypes.h>

typedef uint32_t poly_t;

#endif /* POLY_H */
```

Il suffit donc d'encadrer le source de **chaque fichier en-tête** par ces trois directives *préprocesseur*.

Programmation modulaire – Un exemple (3/5)

- Notre bibliothèque va offrir les opérations algébriques de base comme l'addition de polynômes :

fichier en-tête poly.h

```
#ifndef POLY_H
#define POLY_H
#include <inttypes.h>

typedef uint32_t poly_t;

poly_t poly_add(const poly_t, const poly_t);
#endif /* POLY_H */
```

- Ainsi, dans le fichier interface `poly.h`, on trouvera toutes les déclarations de prototype des fonctions de la bibliothèque.
- On trouvera également des définitions de constantes symboliques comme par exemple :

```
#define POLY_MAX_DEGREE 31
```

Programmation modulaire – Un exemple (3/5)

- Notre bibliothèque va offrir les opérations algébriques de base comme l'addition de polynômes :

fichier en-tête poly.h

```
#ifndef POLY_H
#define POLY_H
#include <inttypes.h>

typedef uint32_t poly_t;

poly_t poly_add(const poly_t, const poly_t);
#endif /* POLY_H */
```

- Ainsi, dans le fichier interface `poly.h`, on trouvera toutes les déclarations de prototype des fonctions de la bibliothèque.
- On trouvera également des définitions de constantes symboliques comme par exemple :

```
#define POLY_MAX_DEGREE 31
```

Programmation modulaire – Un exemple (3/5)

- Notre bibliothèque va offrir les opérations algébriques de base comme l'addition de polynômes :

fichier en-tête poly.h

```
#ifndef POLY_H
#define POLY_H
#include <inttypes.h>

typedef uint32_t poly_t;

poly_t poly_add(const poly_t, const poly_t);
#endif /* POLY_H */
```

- Ainsi, dans le fichier interface `poly.h`, on trouvera toutes les déclarations de prototype des fonctions de la bibliothèque.
- On trouvera également des définitions de **constantes symboliques** comme par exemple :

```
#define POLY_MAX_DEGREE 31
```

Programmation modulaire – Un exemple (4/5)

- Le fichier implémentation `poly.c` contiendra évidemment la ligne :

```
#include "poly.h"
```

afin d'inclure les déclarations/définitions du fichier en-tête `poly.h`.

- Il contiendra également la définition des fonctions déclarées dans l'interface :

fichier source poly.c

```
#include "poly.h"
```

```
poly_t poly_add(const poly_t p, const poly_t q)
{
    return p ^ q; /* ^  $\Rightarrow$  opérateur ou exclusif  $\oplus$  */
}
```

Programmation modulaire – Un exemple (4/5)

- Le fichier implémentation `poly.c` contiendra évidemment la ligne :

```
#include "poly.h"
```

afin d'inclure les déclarations/définitions du fichier en-tête `poly.h`.

- Il contiendra également la définition des fonctions déclarées dans l'interface :

fichier source `poly.c`

```
#include "poly.h"
```

```
poly_t poly_add(const poly_t p, const poly_t q)
{
    return p ^ q; /* ^  $\Rightarrow$  opérateur ou exclusif  $\oplus$  */
}
```

Programmation modulaire – Un exemple (5/5)

- Si on a besoin d'une fonction auxiliaire dans l'implémentation `poly.c`, il faudra la définir dans la classe `static`.
- Cela aura pour effet que cette fonction ne sera connue que du module dans lequel elle a été définie. Cela permet, outre l'amélioration de la lisibilité, de simplifier grandement le **nommage** qui devient un problème épineux lorsque le programme atteint une taille conséquente (un système d'exploitation par exemple).
- Par exemple, si on a besoin d'une fonction auxiliaire qui calcule le maximum de 2 entiers :

```
static int max(int, int); /* prototype au début */

poly_t poly_add(const poly_t p, const poly_t q) {
    return p ^ q; }

static int max(int a, int b) { /* définition à la fin */
    return a > b ? a : b; }
```

Programmation modulaire – Un exemple (5/5)

- Si on a besoin d'une fonction auxiliaire dans l'implémentation `poly.c`, il faudra la définir dans la classe `static`.
- Cela aura pour effet que cette **fonction ne sera connue que du module dans lequel elle a été définie**. Cela permet, outre l'amélioration de la lisibilité, de simplifier grandement le **nommage** qui devient un problème épineux lorsque le programme atteint une taille conséquente (un système d'exploitation par exemple).
- Par exemple, si on a besoin d'une fonction auxiliaire qui calcule le maximum de 2 entiers :

```
static int max(int, int); /* prototype au début */

poly_t poly_add(const poly_t p, const poly_t q) {
    return p ^ q; }

static int max(int a, int b) { /* définition à la fin */
    return a > b ? a : b; }
```

Programmation modulaire – Un exemple (5/5)

- Si on a besoin d'une fonction auxiliaire dans l'implémentation `poly.c`, il faudra la définir dans la classe `static`.
- Cela aura pour effet que cette **fonction ne sera connue que du module dans lequel elle a été définie**. Cela permet, outre l'amélioration de la lisibilité, de simplifier grandement le **nommage** qui devient un problème épineux lorsque le programme atteint une taille conséquente (un système d'exploitation par exemple).
- Par exemple, si on a besoin d'une fonction auxiliaire qui calcule le maximum de 2 entiers :

```
static int max(int, int); /* prototype au début */

poly_t poly_add(const poly_t p, const poly_t q) {
    return p ^ q; }

static int max(int a, int b) { /* définition à la fin */
    return a > b ? a : b; }
```

Le compilateur GNU CC (1/3)

- GCC, développé à l'origine par *Richard Stallmann* à la fin des années 1980, est le compilateur C le plus utilisé dans la communauté du logiciel libre. On le trouve dans la plupart des OS dérivés d'*UNIX* comme *Linux*, **BSD*, *Mac OS X*, etc.
- Pour appeler ce compilateur, c'est la commande `gcc` qui possède un nombre impressionnant d'options (plusieurs centaines).
- Pour compiler le fichier source `source.c` puis effectuer l'édition de liens et générer l'exécutable `source` :

```
$ gcc source.c -o source
```

- On peut scinder l'opération précédente en deux étapes :
 - on effectue tout d'abord la **compilation** du fichier `source.c` :

```
$ gcc -c source.c
```

commande qui génère le fichier objet `source.o`.

- puis c'est ensuite au tour de l'**édition de liens** :

```
$ gcc source.o -o source
```

Le compilateur GNU CC (1/3)

- GCC, développé à l'origine par *Richard Stallmann* à la fin des années 1980, est le compilateur C le plus utilisé dans la communauté du logiciel libre. On le trouve dans la plupart des OS dérivés d'*UNIX* comme *Linux*, **BSD*, *Mac OS X*, etc.
- Pour appeler ce compilateur, c'est la commande `gcc` qui possède un nombre impressionnant d'options (plusieurs centaines).
- Pour compiler le fichier source `source.c` puis effectuer l'édition de liens et générer l'exécutable `source` :

```
$ gcc source.c -o source
```

- On peut scinder l'opération précédente en deux étapes :
 - on effectue tout d'abord la **compilation** du fichier `source.c` :

```
$ gcc -c source.c
```

commande qui génère le fichier objet `source.o`.

- puis c'est ensuite au tour de l'**édition de liens** :

```
$ gcc source.o -o source
```

Le compilateur GNU CC (1/3)

- GCC, développé à l'origine par *Richard Stallmann* à la fin des années 1980, est le compilateur C le plus utilisé dans la communauté du logiciel libre. On le trouve dans la plupart des OS dérivés d'*UNIX* comme *Linux*, **BSD*, *Mac OS X*, etc.
- Pour appeler ce compilateur, c'est la commande `gcc` qui possède un nombre impressionnant d'options (plusieurs centaines).
- Pour compiler le fichier source `source.c` puis effectuer l'édition de liens et générer l'exécutable `source` :

```
$ gcc source.c -o source
```

- On peut scinder l'opération précédente en deux étapes :
 - on effectue tout d'abord la compilation du fichier `source.c` :

```
$ gcc -c source.c
```

commande qui génère le fichier objet `source.o`.

- puis c'est ensuite au tour de l'édition de liens :

```
$ gcc source.o -o source
```

Le compilateur GNU CC (1/3)

- GCC, développé à l'origine par *Richard Stallmann* à la fin des années 1980, est le compilateur C le plus utilisé dans la communauté du logiciel libre. On le trouve dans la plupart des OS dérivés d'*UNIX* comme *Linux*, **BSD*, *Mac OS X*, etc.
- Pour appeler ce compilateur, c'est la commande `gcc` qui possède un nombre impressionnant d'options (plusieurs centaines).
- Pour compiler le fichier source `source.c` puis effectuer l'édition de liens et générer l'exécutable `source` :

```
$ gcc source.c -o source
```

- On peut scinder l'opération précédente en deux étapes :
 - on effectue tout d'abord la **compilation** du fichier `source.c` :

```
$ gcc -c source.c
```

commande qui génère le fichier objet `source.o`.

- puis c'est ensuite au tour de l'**édition de liens** :

```
$ gcc source.o -o source
```

Le compilateur GNU CC (1/3)

- GCC, développé à l'origine par *Richard Stallmann* à la fin des années 1980, est le compilateur C le plus utilisé dans la communauté du logiciel libre. On le trouve dans la plupart des OS dérivés d'*UNIX* comme *Linux*, **BSD*, *Mac OS X*, etc.
- Pour appeler ce compilateur, c'est la commande `gcc` qui possède un nombre impressionnant d'options (plusieurs centaines).
- Pour compiler le fichier source `source.c` puis effectuer l'édition de liens et générer l'exécutable `source` :

```
$ gcc source.c -o source
```

- On peut scinder l'opération précédente en deux étapes :
 - on effectue tout d'abord la **compilation** du fichier `source.c` :

```
$ gcc -c source.c
```

commande qui génère le fichier objet `source.o`.

- puis c'est ensuite au tour de l'**édition de liens** :

```
$ gcc source.o -o source
```

Le compilateur GNU CC (1/3)

- GCC, développé à l'origine par *Richard Stallmann* à la fin des années 1980, est le compilateur C le plus utilisé dans la communauté du logiciel libre. On le trouve dans la plupart des OS dérivés d'*UNIX* comme *Linux*, **BSD*, *Mac OS X*, etc.
- Pour appeler ce compilateur, c'est la commande `gcc` qui possède un nombre impressionnant d'options (plusieurs centaines).
- Pour compiler le fichier source `source.c` puis effectuer l'édition de liens et générer l'exécutable `source` :

```
$ gcc source.c -o source
```

- On peut scinder l'opération précédente en deux étapes :
 - on effectue tout d'abord la **compilation** du fichier `source.c` :

```
$ gcc -c source.c
```

commande qui génère le fichier objet `source.o`.

- puis c'est ensuite au tour de l'**édition de liens** :

```
$ gcc source.o -o source
```

Le compilateur GNU CC (2/3)

- Si on utilise une fonction de la bibliothèque mathématique (par exemple la fonction exponentielle : `double exp(double)`), il faudra charger cette bibliothèque au moment de l'édition de liens avec l'option `-l` :

```
$ gcc source.c -o source -lm
```

- Parmi les nombreuses options de compilation proposées par GCC, il y a une qui devrait toujours être présente : `-Wall` (*Warning all*). Si la compilation de votre code avec cette option ne génère **aucun avertissement** (plusieurs dizaines potentiellement), ce sera un gage de **qualité** de celui-ci.
- Pour encore **plus de contrôle**, on peut ajouter l'option `-Wextra` qui gère d'autres avertissements qui ne sont pas pris en compte par l'option `-Wall` :

```
$ gcc -Wall -Wextra -c source.c
```


Le compilateur GNU CC (2/3)

- Si on utilise une fonction de la bibliothèque mathématique (par exemple la fonction exponentielle : `double exp(double)`), il faudra charger cette bibliothèque au moment de l'édition de liens avec l'option `-l` :

```
$ gcc source.c -o source -lm
```

- Parmi les nombreuses options de compilation proposées par GCC, il y a une qui devrait toujours être présente : `-Wall` (*Warning all*). Si la compilation de votre code avec cette option ne génère **aucun avertissement** (plusieurs dizaines potentiellement), ce sera un gage de **qualité** de celui-ci.
- Pour encore **plus de contrôle**, on peut ajouter l'option `-Wextra` qui gère d'autres avertissements qui ne sont pas pris en compte par l'option `-Wall` :

```
$ gcc -Wall -Wextra -c source.c
```

Le compilateur GNU CC (2/3)

- Si on utilise une fonction de la bibliothèque mathématique (par exemple la fonction exponentielle : `double exp(double)`), il faudra charger cette bibliothèque au moment de l'édition de liens avec l'option `-l` :

```
$ gcc source.c -o source -lm
```

- Parmi les nombreuses options de compilation proposées par GCC, il y a une qui devrait toujours être présente : `-Wall` (*Warning all*). Si la compilation de votre code avec cette option ne génère **aucun avertissement** (plusieurs dizaines potentiellement), ce sera un gage de **qualité** de celui-ci.
- Pour encore **plus de contrôle**, on peut ajouter l'option `-Wextra` qui gère d'autres avertissements qui ne sont pas pris en compte par l'option `-Wall` :

```
$ gcc -Wall -Wextra -c source.c
```

Le compilateur GNU CC (3/3)

- Une autre option de compilation dont nous ferons un usage systématique et dont nous verrons la signification ultérieurement, l'option `-Wpointer-arith`. Cette option n'est prise en compte ni par `-Wall` ni par `-Wextra`.
- Les options d'optimisation de code : `-O` (`-O1`), `-O2` et `-O3`. Ces options, notamment `-O3`, sont d'une **efficacité remarquable** quant à l'**optimisation du code** généré. Cela étant, elles sont consommatrices de ressources et ralentissent la compilation. A n'utiliser donc qu'en fin de cycle de développement.
- En résumé :

– pour compiler :

```
$ gcc -Wall -Wpointer-arith [-Wextra] -c source.c
```

et en fin cycle de développement, on rajoute l'option `-O3`.

– pour l'édition de liens :

```
$ gcc source.o -o source [-l<lib1> [-l<lib2>]...]
```

Le compilateur GNU CC (3/3)

- Une autre option de compilation dont nous ferons un usage systématique et dont nous verrons la signification ultérieurement, l'option `-Wpointer-arith`. Cette option n'est prise en compte ni par `-Wall` ni par `-Wextra`.
- Les options d'optimisation de code : `-O` (`-O1`), `-O2` et `-O3`. Ces options, notamment `-O3`, sont d'une **efficacité remarquable** quant à l'**optimisation du code** généré. Cela étant, elles sont consommatrices de ressources et ralentissent la compilation. A n'utiliser donc qu'en fin de cycle de développement.
- En résumé :

– pour compiler :

```
$ gcc -Wall -Wpointer-arith [-Wextra] -c source.c
```

et en fin cycle de développement, on rajoute l'option `-O3`.

– pour l'édition de liens :

```
$ gcc source.o -o source [-l<lib1> [-l<lib2>]...]
```

Le compilateur GNU CC (3/3)

- Une autre option de compilation dont nous ferons un usage systématique et dont nous verrons la signification ultérieurement, l'option `-Wpointer-arith`. Cette option n'est prise en compte ni par `-Wall` ni par `-Wextra`.
- Les options d'optimisation de code : `-O` (`-O1`), `-O2` et `-O3`. Ces options, notamment `-O3`, sont d'une **efficacité remarquable** quant à l'**optimisation du code** généré. Cela étant, elles sont consommatrices de ressources et ralentissent la compilation. A n'utiliser donc qu'en fin de cycle de développement.
- En résumé :

– pour compiler :

```
$ gcc -Wall -Wpointer-arith [-Wextra] -c source.c
```

et en fin cycle de développement, on rajoute l'option `-O3`.

– pour l'édition de liens :

```
$ gcc source.o -o source [-l<lib1> [-l<lib2>]...]
```

Le compilateur GNU CC (3/3)

- Une autre option de compilation dont nous ferons un usage systématique et dont nous verrons la signification ultérieurement, l'option `-Wpointer-arith`. Cette option n'est prise en compte ni par `-Wall` ni par `-Wextra`.
- Les options d'optimisation de code : `-O` (`-O1`), `-O2` et `-O3`. Ces options, notamment `-O3`, sont d'une **efficacité remarquable** quant à l'**optimisation du code** généré. Cela étant, elles sont consommatrices de ressources et ralentissent la compilation. A n'utiliser donc qu'en fin de cycle de développement.
- En résumé :

- pour compiler :

```
$ gcc -Wall -Wpointer-arith [-Wextra] -c source.c
```

et en fin cycle de développement, on rajoute l'option `-O3`.

- pour l'édition de liens :

```
$ gcc source.o -o source [-l<lib1> [-l<lib2>]...]
```

Le compilateur GNU CC (3/3)

- Une autre option de compilation dont nous ferons un usage systématique et dont nous verrons la signification ultérieurement, l'option `-Wpointer-arith`. Cette option n'est prise en compte ni par `-Wall` ni par `-Wextra`.
- Les options d'optimisation de code : `-O` (`-O1`), `-O2` et `-O3`. Ces options, notamment `-O3`, sont d'une **efficacité remarquable** quant à l'**optimisation du code** généré. Cela étant, elles sont consommatrices de ressources et ralentissent la compilation. A n'utiliser donc qu'en fin de cycle de développement.
- En résumé :
 - pour compiler :

```
$ gcc -Wall -Wpointer-arith [-Wextra] -c source.c
```

et en fin cycle de développement, on rajoute l'option `-O3`.
 - pour l'édition de liens :

```
$ gcc source.o -o source [-l<lib1> [-l<lib2>]...]
```

Le débogage

- Une remarque préliminaire : lorsque l'on rencontre un problème lors de l'exécution d'un programme, comme par exemple une erreur fatale débouchant sur le message bien connu « Segmentation fault », il faut se garder de commencer par lancer un débogueur pour voir comment corriger le programme fautif.
- En effet, il s'agit bien souvent d'un pointeur non initialisé, d'un débordement de mémoire ou d'autres erreurs fréquentes qu'il est bien plus facile de détecter en **relisant le source** du programme.
- Cela étant, la suite de logiciels GNU propose un débogueur : `gdb`.
- Nous lui préférons le logiciel `valgrind` d'un usage plus commode et qui ne fait pas partie de la suite de logiciels GNU.
- Un programme peut s'exécuter normalement sans être pour autant correct. Il peut par exemple y avoir des **fuites mémoire** que `valgrind` saura détecter. Le cycle de développement d'un programme devrait donc comprendre l'**analyse du binaire** par `valgrind`.

Le débogage

- Une remarque préliminaire : lorsque l'on rencontre un problème lors de l'exécution d'un programme, comme par exemple une erreur fatale débouchant sur le message bien connu « Segmentation fault », il faut se garder de commencer par lancer un débogueur pour voir comment corriger le programme fautif.
- En effet, il s'agit bien souvent d'un pointeur non initialisé, d'un débordement de mémoire ou d'autres erreurs fréquentes qu'il est bien plus facile de détecter en **relisant le source** du programme.
- Cela étant, la suite de logiciels GNU propose un débogueur : `gdb`.
- Nous lui préférons le logiciel `valgrind` d'un usage plus commode et qui ne fait pas partie de la suite de logiciels GNU.
- Un programme peut s'exécuter normalement sans être pour autant correct. Il peut par exemple y avoir des **fuites mémoire** que `valgrind` saura détecter. Le cycle de développement d'un programme devrait donc comprendre l'**analyse du binaire** par `valgrind`.

Le débogage

- Une remarque préliminaire : lorsque l'on rencontre un problème lors de l'exécution d'un programme, comme par exemple une erreur fatale débouchant sur le message bien connu « Segmentation fault », il faut se garder de commencer par lancer un débogueur pour voir comment corriger le programme fautif.
- En effet, il s'agit bien souvent d'un pointeur non initialisé, d'un débordement de mémoire ou d'autres erreurs fréquentes qu'il est bien plus facile de détecter en **relisant le source** du programme.
- Cela étant, la suite de logiciels GNU propose un débogueur : `gdb`.
- Nous lui préférons le logiciel `valgrind` d'un usage plus commode et qui ne fait pas partie de la suite de logiciels GNU.
- Un programme peut s'exécuter normalement sans être pour autant correct. Il peut par exemple y avoir des **fuites mémoire** que `valgrind` saura détecter. Le cycle de développement d'un programme devrait donc comprendre l'**analyse du binaire** par `valgrind`.

Le débogage

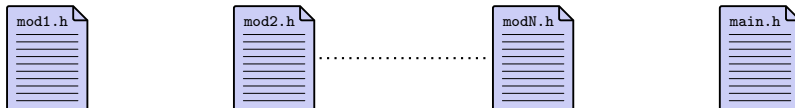
- Une remarque préliminaire : lorsque l'on rencontre un problème lors de l'exécution d'un programme, comme par exemple une erreur fatale débouchant sur le message bien connu « Segmentation fault », il faut se garder de commencer par lancer un débogueur pour voir comment corriger le programme fautif.
- En effet, il s'agit bien souvent d'un pointeur non initialisé, d'un débordement de mémoire ou d'autres erreurs fréquentes qu'il est bien plus facile de détecter en **relisant le source** du programme.
- Cela étant, la suite de logiciels GNU propose un débogueur : `gdb`.
- Nous lui préférons le logiciel `valgrind` d'un usage plus commode et qui ne fait pas partie de la suite de logiciels GNU.
- Un programme peut s'exécuter normalement sans être pour autant correct. Il peut par exemple y avoir des **fuites mémoire** que `valgrind` saura détecter. Le cycle de développement d'un programme devrait donc comprendre l'**analyse du binaire** par `valgrind`.

Le débogage

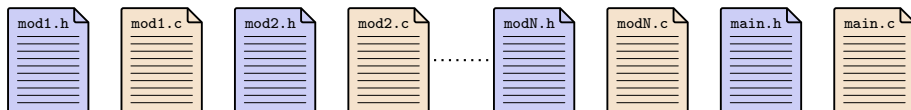
- Une remarque préliminaire : lorsque l'on rencontre un problème lors de l'exécution d'un programme, comme par exemple une erreur fatale débouchant sur le message bien connu « Segmentation fault », il faut se garder de commencer par lancer un débogueur pour voir comment corriger le programme fautif.
- En effet, il s'agit bien souvent d'un pointeur non initialisé, d'un débordement de mémoire ou d'autres erreurs fréquentes qu'il est bien plus facile de détecter en **relisant le source** du programme.
- Cela étant, la suite de logiciels GNU propose un débogueur : `gdb`.
- Nous lui préférons le logiciel `valgrind` d'un usage plus commode et qui ne fait pas partie de la suite de logiciels GNU.
- Un programme peut s'exécuter normalement sans être pour autant correct. Il peut par exemple y avoir des **fuites mémoire** que `valgrind` saura détecter. Le cycle de développement d'un programme devrait donc comprendre l'**analyse du binaire** par `valgrind`.

Le cycle de développement

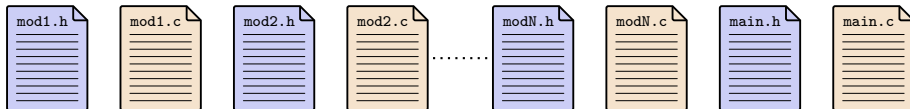
On commence par les fichiers en-têtes : `mod1.h`, ..., `modN.h`, `main.h`



Puis les fichiers sources : `mod1.c`, ..., `modN.c`, `main.c`

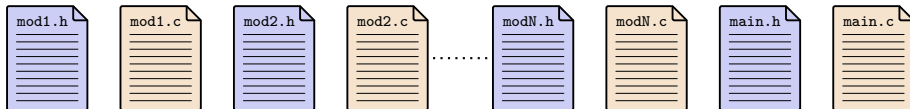


Compilation des fichiers sources



```
$ gcc -Wall -Wpointer-arith [-Wextra] [-O3] -c mod1.c  
$ gcc -Wall -Wpointer-arith [-Wextra] [-O3] -c main.c
```

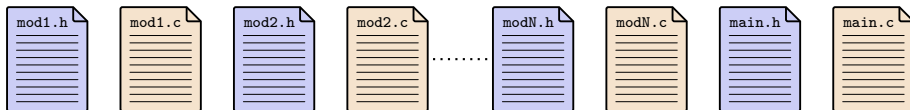

Création des fichiers objets : mod1.o,...,modN.o, main.o



```
$ gcc -Wall -Wpointer-arith [-Wextra] [-O3] -c modI.c  
$ gcc -Wall -Wpointer-arith [-Wextra] [-O3] -c main.c
```



Édition de liens

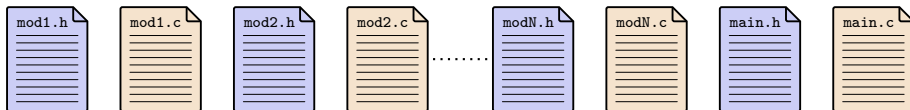


```
$ gcc -Wall -Wpointer-arith [-Wextra] [-O3] -c mod1.c  
$ gcc -Wall -Wpointer-arith [-Wextra] [-O3] -c main.c
```



```
$ gcc mod1.o ... modN.o main.o -o main [-llib1 [-llib2 ...]]
```

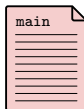
Création de l'exécutable main



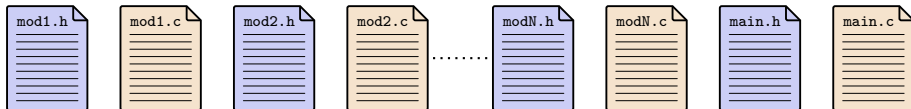
```
$ gcc -Wall -Wpointer-arith [-Wextra] [-O3] -c mod1.c  
$ gcc -Wall -Wpointer-arith [-Wextra] [-O3] -c main.c
```



```
$ gcc mod1.o ... modN.o main.o -o main [-llib1 [-llib2 ...]]
```



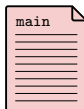
Vérification de l'exécutable avec valgrind



```
$ gcc -Wall -Wpointer-arith [-Wextra] [-O3] -c mod1.c  
$ gcc -Wall -Wpointer-arith [-Wextra] [-O3] -c main.c
```



```
$ gcc mod1.o ... modN.o main.o -o main [-llib1 [-llib2 ...]]
```



```
$ valgrind ./main [arg1 [arg2 ...]]
```

Quelques règles de base (1/3)

- Programmer de manière **modulaire**. Éviter les fichiers sources trop volumineux.
- Pour chaque module, commencer toujours par écrire l'**interface** et seulement ensuite l'**implémentation**.
- Concernant la syntaxe, **une instruction par ligne** et surtout une **indentation du code** sans laquelle un programme de plus de quelques lignes est illisible.
- Ne jamais utiliser de constantes « en dur » mais des **constantes symboliques** :

```
#define PHI 1.61803
```

```
double x = PHI;
```

et non pas :

```
double x = 1.61803;
```

Quelques règles de base (1/3)

- Programmer de manière **modulaire**. Éviter les fichiers sources trop volumineux.
- Pour chaque module, commencer toujours par écrire l'**interface** et seulement ensuite l'**implémentation**.
- Concernant la syntaxe, **une instruction par ligne** et surtout une **indentation du code** sans laquelle un programme de plus de quelques lignes est illisible.
- Ne jamais utiliser de constantes « en dur » mais des **constantes symboliques** :

```
#define PHI 1.61803
```

```
double x = PHI;
```

et non pas :

```
double x = 1.61803;
```

Quelques règles de base (1/3)

- Programmer de manière **modulaire**. Éviter les fichiers sources trop volumineux.
- Pour chaque module, commencer toujours par écrire l'**interface** et seulement ensuite l'**implémentation**.
- Concernant la syntaxe, **une instruction par ligne** et surtout une **indentation du code** sans laquelle un programme de plus de quelques lignes est illisible.
- Ne jamais utiliser de constantes « en dur » mais des **constantes symboliques** :

```
#define PHI 1.61803  
  
double x = PHI;
```

et non pas :

```
double x = 1.61803;
```

Quelques règles de base (1/3)

- Programmer de manière **modulaire**. Éviter les fichiers sources trop volumineux.
- Pour chaque module, commencer toujours par écrire l'**interface** et seulement ensuite l'**implémentation**.
- Concernant la syntaxe, **une instruction par ligne** et surtout une **indentation du code** sans laquelle un programme de plus de quelques lignes est illisible.
- Ne jamais utiliser de constantes « en dur » mais des **constantes symboliques** :

```
#define PHI 1.61803
```

```
double x = PHI;
```

et non pas :

```
double x = 1.61803;
```


Quelques règles de base (2/3)

- Compiler chaque source avec, au minimum, les options `-Wall` et `-Wpointer-arith`.
- Ne pas oublier de tester les parties de code traitant des erreurs en déclenchant ces erreurs. Par exemple :

```
if (expression) {  
    /* traitement de l'erreur */  
}
```

Pour déclencher l'erreur :

```
if (!expression) {  
    /* traitement de l'erreur */  
}
```

puis recompiler et exécuter. Si on ne s'astreint pas à cette discipline, le code traitant les erreurs est **rarement ou jamais testé**.

- Analyser le binaire avec le logiciel `valgrind` (détection de fuites mémoire, de pointeurs non initialisés, etc.).

Quelques règles de base (2/3)

- Compiler chaque source avec, au minimum, les options `-Wall` et `-Wpointer-arith`.
- Ne pas oublier de tester les parties de code traitant des erreurs en **déclenchant ces erreurs**. Par exemple :

```
if (expression) {  
    /* traitement de l'erreur */  
}
```

Pour déclencher l'erreur :

```
if (!expression) {  
    /* traitement de l'erreur */  
}
```

puis recompiler et exécuter. Si on ne s'astreint pas à cette discipline, le code traitant les erreurs est **rarement ou jamais testé**.

- Analyser le binaire avec le logiciel `valgrind` (détection de fuites mémoire, de pointeurs non initialisés, etc.).

Quelques règles de base (2/3)

- Compiler chaque source avec, au minimum, les options `-Wall` et `-Wpointer-arith`.
- Ne pas oublier de tester les parties de code traitant des erreurs en **déclenchant ces erreurs**. Par exemple :

```
if (expression) {  
    /* traitement de l'erreur */  
}
```

Pour déclencher l'erreur :

```
if (!expression) {  
    /* traitement de l'erreur */  
}
```

puis recompiler et exécuter. Si on ne s'astreint pas à cette discipline, le code traitant les erreurs est **rarement ou jamais testé**.

- Analyser le binaire avec le logiciel `valgrind` (détection de fuites mémoire, de pointeurs non initialisés, etc.).

Quelques règles de base (3/3)

- Le **préprocesseur** est un outil puissant qui permet, entre autres, de rendre **plus lisible** le code, de faire de la **compilation conditionnelle** (et donc, en particulier, d'avoir plusieurs versions d'un logiciel dans un même source).
- Utiliser systématiquement **typedef** lorsque vous définissez un **nouveau type** (composé ou pas). Nous verrons même, qu'avec les pointeurs de fonction, c'est une quasi nécessité.
- Lorsque le nombre de modules devient conséquent, employer les outils **make** (*BSD) ou **gmake** (GNU).
- Pour assurer la **portabilité** de votre code, la suite GNU offre un outil remarquable : **autoconf**. Cet outil permet d'automatiquement configurer votre logiciel en fonction de l'environnement (OS, bibliothèques disponibles, etc.)
- Et enfin, tout ce j'ai oublié de mentionner 😊

Quelques règles de base (3/3)

- Le **préprocesseur** est un outil puissant qui permet, entre autres, de rendre **plus lisible** le code, de faire de la **compilation conditionnelle** (et donc, en particulier, d'avoir plusieurs versions d'un logiciel dans un même source).
- Utiliser systématiquement **typedef** lorsque vous définissez un **nouveau type** (composé ou pas). Nous verrons même, qu'avec les pointeurs de fonction, c'est une quasi nécessité.
- Lorsque le nombre de modules devient conséquent, employer les outils **make** (*BSD) ou **gmake** (GNU).
- Pour assurer la **portabilité** de votre code, la suite GNU offre un outil remarquable : **autoconf**. Cet outil permet d'automatiquement configurer votre logiciel en fonction de l'environnement (OS, bibliothèques disponibles, etc.)
- Et enfin, tout ce j'ai oublié de mentionner 😊

Quelques règles de base (3/3)

- Le **préprocesseur** est un outil puissant qui permet, entre autres, de rendre **plus lisible** le code, de faire de la **compilation conditionnelle** (et donc, en particulier, d'avoir plusieurs versions d'un logiciel dans un même source).
- Utiliser systématiquement **typedef** lorsque vous définissez un **nouveau type** (composé ou pas). Nous verrons même, qu'avec les pointeurs de fonction, c'est une quasi nécessité.
- Lorsque le nombre de modules devient conséquent, employer les outils **make** (*BSD) ou **gmake** (GNU).
- Pour assurer la **portabilité** de votre code, la suite GNU offre un outil remarquable : **autoconf**. Cet outil permet d'automatiquement configurer votre logiciel en fonction de l'environnement (OS, bibliothèques disponibles, etc.)
- Et enfin, tout ce j'ai oublié de mentionner 😊

Quelques règles de base (3/3)

- Le **préprocesseur** est un outil puissant qui permet, entre autres, de rendre **plus lisible** le code, de faire de la **compilation conditionnelle** (et donc, en particulier, d'avoir plusieurs versions d'un logiciel dans un même source).
- Utiliser systématiquement **typedef** lorsque vous définissez un **nouveau type** (composé ou pas). Nous verrons même, qu'avec les pointeurs de fonction, c'est une quasi nécessité.
- Lorsque le nombre de modules devient conséquent, employer les outils **make** (*BSD) ou **gmake** (GNU).
- Pour assurer la **portabilité** de votre code, la suite GNU offre un outil remarquable : **autoconf**. Cet outil permet d'automatiquement configurer votre logiciel en fonction de l'environnement (OS, bibliothèques disponibles, etc.)
- Et enfin, tout ce j'ai oublié de mentionner 😊

Quelques règles de base (3/3)

- Le **préprocesseur** est un outil puissant qui permet, entre autres, de rendre **plus lisible** le code, de faire de la **compilation conditionnelle** (et donc, en particulier, d'avoir plusieurs versions d'un logiciel dans un même source).
- Utiliser systématiquement **typedef** lorsque vous définissez un **nouveau type** (composé ou pas). Nous verrons même, qu'avec les pointeurs de fonction, c'est une quasi nécessité.
- Lorsque le nombre de modules devient conséquent, employer les outils **make** (*BSD) ou **gmake** (GNU).
- Pour assurer la **portabilité** de votre code, la suite GNU offre un outil remarquable : **autoconf**. Cet outil permet d'automatiquement configurer votre logiciel en fonction de l'environnement (OS, bibliothèques disponibles, etc.)
- Et enfin, tout ce j'ai oublié de mentionner 😊

Une définition quelque peu formelle

Définition

*Un pointeur est une **variable** destinée à contenir l'adresse d'un autre objet C. Cet autre objet C peut être absolument **quelconque**, par exemple un pointeur également (on parle alors de « double pointeur »).*

Un pointeur se déclare de la manière suivante :

```
type *nom_du_pointeur;
```

Par exemple, pour déclarer un pointeur sur un entier :

```
int *i;
```

ou un double pointeur sur un flottant simple précision :

```
float **f;
```

Une définition quelque peu formelle

Définition

*Un pointeur est une **variable** destinée à contenir l'adresse d'un autre objet C. Cet autre objet C peut être absolument **quelconque**, par exemple un pointeur également (on parle alors de « double pointeur »).*

Un pointeur se déclare de la manière suivante :

```
type *nom_du_pointeur;
```

Par exemple, pour déclarer un pointeur sur un entier :

```
int *i;
```

ou un double pointeur sur un flottant simple précision :

```
float **f;
```

Une définition quelque peu formelle

Définition

*Un pointeur est une **variable** destinée à contenir l'adresse d'un autre objet C. Cet autre objet C peut être absolument **quelconque**, par exemple un pointeur également (on parle alors de « double pointeur »).*

Un pointeur se déclare de la manière suivante :

```
type *nom_du_pointeur;
```

Par exemple, pour déclarer un pointeur sur un entier :

```
int *i;
```

ou un double pointeur sur un flottant simple précision :

```
float **f;
```

Initialisation des pointeurs (1/2)

- Une fois déclaré, un pointeur doit être **initialisé**, avant toute utilisation. La manipulation de pointeurs non initialisés est une erreur fréquente engendrant une sortie brutale du programme ou un comportement erratique de ce dernier.
- Voyons, sur un exemple, une première manière d'initialiser un pointeur :

```
int n;      /* déclaration de l'entier n */  
int *p;     /* déclaration du pointeur p */  
  
n = 3;      /* initialisation de l'entier n */  
p = &n;     /* initialisation du pointeur p */
```

L'objet `&n` est un pointeur **constant** (vers un entier) dont la valeur est l'adresse de l'objet `n`.

Initialisation des pointeurs (1/2)

- Une fois déclaré, un pointeur doit être **initialisé**, avant toute utilisation. La manipulation de pointeurs non initialisés est une erreur fréquente engendrant une sortie brutale du programme ou un comportement erratique de ce dernier.
- Voyons, sur un exemple, une première manière d'initialiser un pointeur :

```
int n;      /* déclaration de l'entier n */
int *p;     /* déclaration du pointeur p */

n = 3;      /* initialisation de l'entier n */
p = &n;     /* initialisation du pointeur p */
```

L'objet `&n` est un pointeur **constant** (vers un entier) dont la valeur est l'adresse de l'objet `n`.

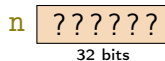
Initialisation des pointeurs (2/2)

Schéma mémoire (sur une machine 64 bits)

Initialisation des pointeurs (2/2)

Schéma mémoire (sur une machine 64 bits)

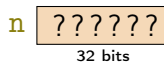
```
int n;
```



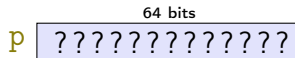
Initialisation des pointeurs (2/2)

Schéma mémoire (sur une machine 64 bits)

```
int n;
```



```
int *p;
```



Initialisation des pointeurs (2/2)

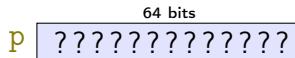
Schéma mémoire (sur une machine 64 bits)

```
int n;
```



```
int *p;
```

```
n = 3;
```



Initialisation des pointeurs (2/2)

Schéma mémoire (sur une machine 64 bits)

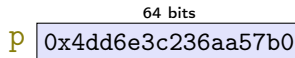
```
int n;
```



```
int *p;
```

```
n = 3;
```

```
p = &n;
```



Initialisation des pointeurs (2/2)

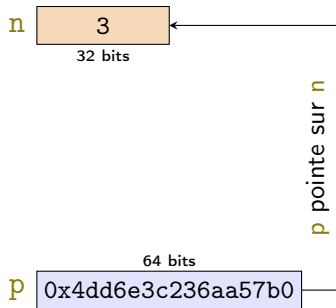
Schéma mémoire (sur une machine 64 bits)

```
int n;
```

```
int *p;
```

```
n = 3;
```

```
p = &n;
```



L'opérateur d'indirection * (1/2)

- L'opérateur d'indirection *, appliqué à un pointeur, permet d'accéder directement à la valeur de l'objet pointé.
- Si l'on reprend l'exemple précédent :

```
int i = 3;  
int *p;
```

```
p = &i;
```

alors `*p` est l'entier 3, valeur de `i`. On peut en particulier modifier la valeur de l'entier `i` via le pointeur `p` :

```
*p = 5; /* i vaut maintenant 5 */
```

- Attention : il est interdit d'appliquer l'opérateur d'indirection à un pointeur de type `void *` :

```
void *p; /* l'expression *p est illégale */
```

L'opérateur d'indirection * (1/2)

- L'opérateur d'indirection *, appliqué à un pointeur, permet d'accéder directement à la valeur de l'objet pointé.
- Si l'on reprend l'exemple précédent :

```
int i = 3;  
int *p;
```

```
p = &n;
```

alors `*p` est l'entier 3, valeur de `i`. On peut en particulier modifier la valeur de l'entier `i` via le pointeur `p` :

```
*p = 5; /* i vaut maintenant 5 */
```

- Attention : il est interdit d'appliquer l'opérateur d'indirection à un pointeur de type `void *` :

```
void *p; /* l'expression *p est illégale */
```

L'opérateur d'indirection * (1/2)

- L'opérateur d'indirection *, appliqué à un pointeur, permet d'accéder directement à la valeur de l'objet pointé.
- Si l'on reprend l'exemple précédent :

```
int i = 3;  
int *p;
```

```
p = &i;
```

alors `*p` est l'entier 3, valeur de `i`. On peut en particulier modifier la valeur de l'entier `i` via le pointeur `p` :

```
*p = 5; /* i vaut maintenant 5 */
```

- **Attention** : il est interdit d'appliquer l'opérateur d'indirection à un pointeur de type `void *` :

```
void *p; /* l'expression *p est illégale */
```

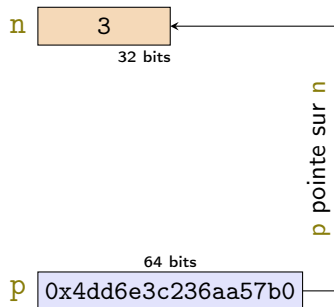
L'opérateur d'indirection * (2/2)

Schéma mémoire (sur une machine 64 bits)

L'opérateur d'indirection * (2/2)

Schéma mémoire (sur une machine 64 bits)

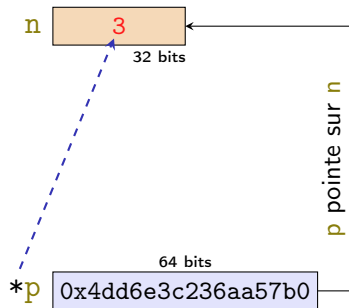
```
int n;  
  
int *p;  
  
n = 3;  
  
p = &n;
```



L'opérateur d'indirection * (2/2)

Schéma mémoire (sur une machine 64 bits)

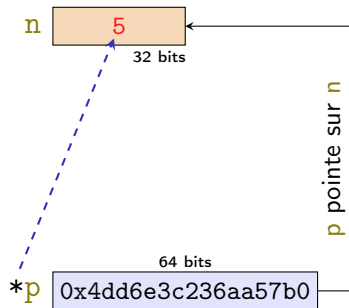
```
int n;  
  
int *p;  
  
n = 3;  
  
p = &n;
```



L'opérateur d'indirection * (2/2)

Schéma mémoire (sur une machine 64 bits)

```
int n;  
  
int *p;  
  
n = 3;  
  
p = &n;  
  
*p = 5;
```



Arithmétique des pointeurs

Il y a trois opérations arithmétiques définies sur les pointeurs (excepté les pointeurs de fonctions et les pointeurs de type `void *`) :

- **addition d'un entier** : soit `type *p` un pointeur et `i` un entier, alors `p + i` est un pointeur du même type dont la valeur est celle de `p` augmentée de `i * sizeof(type)`,
- **soustraction d'un entier** : définition analogue à la précédente, `p - i` valant dans ce cas `p - (i * sizeof(type))`
- **différence de deux pointeurs** : soit `p` et `q` deux pointeurs de même type, alors `p - q` est un entier égal à `p - q / sizeof(type)`.
Attention : en pratique, `p - q` n'a de sens que si les pointeurs `p` et `q` pointent sur la même zone mémoire (un tableau ou une zone allouée dynamiquement).

Arithmétique des pointeurs

Il y a trois opérations arithmétiques définies sur les pointeurs (excepté les pointeurs de fonctions et les pointeurs de type `void *`) :

- **addition d'un entier** : soit `type *p` un pointeur et `i` un entier, alors `p + i` est un pointeur du même type dont la valeur est celle de `p` augmentée de `i * sizeof(type)`,
- **soustraction d'un entier** : définition analogue à la précédente, `p - i` valant dans ce cas `p - (i * sizeof(type))`
- **différence de deux pointeurs** : soit `p` et `q` deux pointeurs de même type, alors `p - q` est un entier égal à `(p - q) / sizeof(type)`.
Attention : en pratique, `p - q` n'a de sens que si les pointeurs `p` et `q` pointent sur la même zone mémoire (un tableau ou une zone allouée dynamiquement).

Arithmétique des pointeurs

Il y a trois opérations arithmétiques définies sur les pointeurs (excepté les pointeurs de fonctions et les pointeurs de type `void *`) :

- **addition d'un entier** : soit `type *p` un pointeur et `i` un entier, alors `p + i` est un pointeur du même type dont la valeur est celle de `p` augmentée de `i * sizeof(type)`,
- **soustraction d'un entier** : définition analogue à la précédente, `p - i` valant dans ce cas `p - (i * sizeof(type))`
- **différence de deux pointeurs** : soit `p` et `q` deux pointeurs de même type, alors `p - q` est un entier égal à `(p - q) / sizeof(type)`.
Attention : en pratique, `p - q` n'a de sens que si les pointeurs `p` et `q` pointent sur la même zone mémoire (un tableau ou une zone allouée dynamiquement).

Arithmétique des pointeurs

Il y a trois opérations arithmétiques définies sur les pointeurs (excepté les pointeurs de fonctions et les pointeurs de type `void *`) :

- **addition d'un entier** : soit `type *p` un pointeur et `i` un entier, alors `p + i` est un pointeur du même type dont la valeur est celle de `p` augmentée de `i * sizeof(type)`,
- **soustraction d'un entier** : définition analogue à la précédente, `p - i` valant dans ce cas `p - (i * sizeof(type))`
- **différence de deux pointeurs** : soit `p` et `q` deux pointeurs de même type, alors `p - q` est un entier égal à `p - q / sizeof(type)`.
Attention : en pratique, `p - q` n'a de sens que si les pointeurs `p` et `q` pointent sur la même zone mémoire (un tableau ou une zone allouée dynamiquement).

Arithmétique des pointeurs - Compléments

- L'addition et la soustraction d'un entier étant définies, les opérateurs d'incrémentation et de décrémentation `++` et `--` le sont aussi. Par exemple :

```
int *p, *q;  
p++ = q++; /* équivaut à p = q; p += 1; q += 1; */  
--p = --q; /* équivaut à p -= 1; q -= 1; p = q; */
```

Ces opérateurs ne s'appliquent pas aux pointeurs constants, `(&i)++` est illégal.

- Une abréviation utile : `*(p + i)` peut se noter `p[i]`.
- Outre les opérations arithmétiques, les **opérateurs de comparaison** sont également définis pour deux pointeurs de même type `p` et `q`. Plus précisément les expressions `p < q`, `p <= q`, `p == q`,... sont tout simplement définies comme comparaison des valeurs des pointeurs `p` et `q`. À noter qu'il n'y a pas de restrictions sur le type de pointeur (pointeurs de fonctions et pointeurs de type `void *`).

Arithmétique des pointeurs - Compléments

- L'addition et la soustraction d'un entier étant définies, les opérateurs d'incrément et de décrémentation `++` et `--` le sont aussi. Par exemple :

```
int *p, *q;  
p++ = q++; /* équivaut à p = q; p += 1; q += 1; */  
--p = --q; /* équivaut à p -= 1; q -= 1; p = q; */
```

Ces opérateurs ne s'appliquent pas aux pointeurs constants, `(&i)++` est illégal.

- Une abréviation utile : `*(p + i)` peut se noter `p[i]`.
- Outre les opérations arithmétiques, les **opérateurs de comparaison** sont également définis pour deux pointeurs de même type `p` et `q`. Plus précisément les expressions `p < q`, `p <= q`, `p == q`,... sont tout simplement définies comme comparaison des valeurs des pointeurs `p` et `q`. À noter qu'il n'y a pas de restrictions sur le type de pointeur (pointeurs de fonctions et pointeurs de type `void *`).

Arithmétique des pointeurs - Compléments

- L'addition et la soustraction d'un entier étant définies, les opérateurs d'incrément et de décrémentation `++` et `--` le sont aussi. Par exemple :

```
int *p, *q;  
p++ = q++; /* équivaut à p = q; p += 1; q += 1; */  
--p = --q; /* équivaut à p -= 1; q -= 1; p = q; */
```

Ces opérateurs ne s'appliquent pas aux pointeurs constants, `(&i)++` est illégal.

- Une abréviation utile : `*(p + i)` peut se noter `p[i]`.
- Outre les opérations arithmétiques, les **opérateurs de comparaison** sont également définis pour deux pointeurs de même type `p` et `q`. Plus précisément les expressions `p < q`, `p <= q`, `p == q`,... sont tout simplement définies comme comparaison des valeurs des pointeurs `p` et `q`. À noter qu'il n'y a pas de restrictions sur le type de pointeur (pointeurs de fonctions et pointeurs de type `void *`).

Arithmétique des pointeurs – Quelques exemples

```
int i;  
int *p = &i;  
  
printf("p=%p, p+2=%p\n", p, p + 2);
```

Sortie du programme :

```
p = 0x7ffec05a6b24, p + 2 = 0x7ffec05a6b2c
```

```
int i;  
int *p = &i, *q = p + 2;  
  
printf("p=%p, q=%p, q-p=%zd\n", p, q, q - p);
```

Sortie du programme :

```
p = 0x7ffc57cb17ac, q = 0x7ffc57cb17b4, q - p = 2
```

Arithmétique des pointeurs – Quelques exemples

```
int i;  
int *p = &i;  
  
printf("p=%p, p+2=%p\n", p, p + 2);
```

Sortie du programme :

```
p = 0x7ffec05a6b24, p + 2 = 0x7ffec05a6b2c
```

```
int i;  
int *p = &i, *q = p + 2;  
  
printf("p=%p, q=%p, q-p=%zd\n", p, q, q - p);
```

Sortie du programme :

```
p = 0x7ffc57cb17ac, q = 0x7ffc57cb17b4, q - p = 2
```

Arithmétique des pointeurs – Quelques exemples

```
int i;  
int *p = &i;  
  
printf("p=%p, p+2=%p\n", p, p + 2);
```

Sortie du programme :

```
p = 0x7ffec05a6b24, p + 2 = 0x7ffec05a6b2c
```

```
int i;  
int *p = &i, *q = p + 2;  
  
printf("p=%p, q=%p, q-p=%zd\n", p, q, q - p);
```

Sortie du programme :

```
p = 0x7ffc57cb17ac, q = 0x7ffc57cb17b4, q - p = 2
```

Arithmétique des pointeurs – Quelques exemples

```
int i;  
int *p = &i;  
  
printf("p=%p, p+2=%p\n", p, p + 2);
```

Sortie du programme :

```
p = 0x7ffec05a6b24, p + 2 = 0x7ffec05a6b2c
```

```
int i;  
int *p = &i, *q = p + 2;  
  
printf("p=%p, q=%p, q-p=%zd\n", p, q, q - p);
```

Sortie du programme :

```
p = 0x7ffc57cb17ac, q = 0x7ffc57cb17b4, q - p = 2
```

Arithmétique des pointeurs – Pointeurs de type `void *`

- A priori, l'expression `sizeof(void)` est dénuée de sens et donc également une opération arithmétique comme l'addition.
- Cela étant, considérons le programme *test.c* suivant :

```
int main(void)
{ printf("sizeof(void): %zu\n", sizeof(void)); }
```

- Si on le compile avec GNU CC et qu'on l'exécute :

```
$ gcc -Wall test.c -o test # pas d'avertissement
$ ./test
sizeof(void): 1
```

- En revanche si on le compile comme suit :

```
$ gcc -Wall -Wpointer-arith test.c -o test
```

on a l'avertissement :

```
warning: invalid application of 'sizeof' to a void type
```

Arithmétique des pointeurs – Pointeurs de type `void *`

- A priori, l'expression `sizeof(void)` est dénuée de sens et donc également une opération arithmétique comme l'addition.
- Cela étant, considérons le programme *test.c* suivant :

```
int main(void)
{ printf("sizeof(void): %zu\n", sizeof(void)); }
```

- Si on le compile avec GNU CC et qu'on l'exécute :

```
$ gcc -Wall test.c -o test # pas d'avertissement
$ ./test
sizeof(void): 1
```

- En revanche si on le compile comme suit :

```
$ gcc -Wall -Wpointer-arith test.c -o test
```

on a l'avertissement :

```
warning: invalid application of 'sizeof' to a void type
```

Arithmétique des pointeurs – Pointeurs de type `void *`

- A priori, l'expression `sizeof(void)` est dénuée de sens et donc également une opération arithmétique comme l'addition.
- Cela étant, considérons le programme *test.c* suivant :

```
int main(void)
{ printf("sizeof(void): %zu\n", sizeof(void)); }
```

- Si on le compile avec GNU CC et qu'on l'exécute :

```
$ gcc -Wall test.c -o test # pas d'avertissement
$ ./test
sizeof(void): 1
```

- En revanche si on le compile comme suit :

```
$ gcc -Wall -Wpointer-arith test.c -o test
```

on a l'avertissement :

```
warning: invalid application of 'sizeof' to a void type
```


Arithmétique des pointeurs – Pointeurs de type `void *`

- A priori, l'expression `sizeof(void)` est dénuée de sens et donc également une opération arithmétique comme l'addition.
- Cela étant, considérons le programme *test.c* suivant :

```
int main(void)
{ printf("sizeof(void): %zu\n", sizeof(void)); }
```

- Si on le compile avec GNU CC et qu'on l'exécute :

```
$ gcc -Wall test.c -o test # pas d'avertissement
$ ./test
sizeof(void): 1
```

- En revanche si on le compile comme suit :

```
$ gcc -Wall -Wpointer-arith test.c -o test
```

on a l'avertissement :

```
warning: invalid application of 'sizeof' to a void type
```

Coercition de pointeurs

- L'affectation entre deux pointeurs de types différents est incorrecte sauf si l'un des pointeurs est de type `void *`. Par exemple :

```
int *p;  
char *q;  
void *r;  
  
r = q; /* correct */  
p = q; /* incorrect */
```

La dernière instruction engendre, avec le compilateur GNU CC, l'avertissement :

```
warning: assignment from incompatible pointer type
```

- Il faut, si l'on veut effectuer une affectation entre pointeurs de types différents, avoir recours à la **coercition** (*cast* en anglais) de type. Si l'on reprend, l'exemple précédent :

```
p = (int *) q; /* correct */
```

Coercition de pointeurs

- L'affectation entre deux pointeurs de types différents est incorrecte sauf si l'un des pointeurs est de type `void *`. Par exemple :

```
int *p;  
char *q;  
void *r;  
  
r = q; /* correct */  
p = q; /* incorrect */
```

La dernière instruction engendre, avec le compilateur GNU CC, l'avertissement :

```
warning: assignment from incompatible pointer type
```

- Il faut, si l'on veut effectuer une affectation entre pointeurs de types différents, avoir recours à la **coercition** (*cast* en anglais) de type. Si l'on reprend, l'exemple précédent :

```
p = (int *) q; /* correct */
```

Le pointeur nul

- Le pointeur nul est un pointeur ayant une valeur réservée indiquant que ce pointeur n'est pas **valide**.
- La norme C99 dit que cette valeur réservée est l'entier 0, converti implicitement ou explicitement en `void *`.
- Dans le fichier `stdio.h`, on trouve la ligne :

```
#define NULL 0
```

ou la ligne :

```
#define NULL ((void *) 0)
```

- On utilise la constante `NULL`, par exemple ainsi :

```
#include <stdio.h>
char *p = NULL; /* pointeur invalide */
.....
if (p != NULL) { /* teste si p est valide */
    .....
}
```

Le pointeur nul

- Le pointeur nul est un pointeur ayant une valeur réservée indiquant que ce pointeur n'est pas **valide**.
- La norme C99 dit que cette valeur réservée est l'entier 0, converti implicitement ou explicitement en `void *`.
- Dans le fichier `stdio.h`, on trouve la ligne :

```
#define NULL 0
```

ou la ligne :

```
#define NULL ((void *) 0)
```

- On utilise la constante `NULL`, par exemple ainsi :

```
#include <stdio.h>
char *p = NULL; /* pointeur invalide */
.....
if (p != NULL) { /* teste si p est valide */
    .....
}
```

Le pointeur nul

- Le pointeur nul est un pointeur ayant une valeur réservée indiquant que ce pointeur n'est pas **valide**.
- La norme C99 dit que cette valeur réservée est l'entier 0, converti implicitement ou explicitement en `void *`.
- Dans le fichier `stdio.h`, on trouve la ligne :

```
#define NULL 0
```

ou la ligne :

```
#define NULL ((void *) 0)
```

- On utilise la constante `NULL`, par exemple ainsi :

```
#include <stdio.h>
char *p = NULL; /* pointeur invalide */
.....
if (p != NULL) { /* teste si p est valide */
    .....
}
```

Le pointeur nul

- Le pointeur nul est un pointeur ayant une valeur réservée indiquant que ce pointeur n'est pas **valide**.
- La norme C99 dit que cette valeur réservée est l'entier 0, converti implicitement ou explicitement en `void *`.
- Dans le fichier `stdio.h`, on trouve la ligne :

```
#define NULL 0
```

ou la ligne :

```
#define NULL ((void *) 0)
```

- On utilise la constante `NULL`, par exemple ainsi :

```
#include <stdio.h>
char *p = NULL; /* pointeur invalide */
.....
if (p != NULL) { /* teste si p est valide */
    .....
}
```

Allocation dynamique (1/3)

- La primitive système `malloc`, de prototype `void *malloc(size_t)`, permet d'allouer *dynamiquement* une zone mémoire de taille en octets le paramètre de `malloc`. Lorsque cette zone n'est plus utilisée au sein du programme, il faut la désallouer à l'aide de la primitive `free` de prototype `void free(void *)`.
- Par exemple, pour allouer une zone mémoire destinée à accueillir dix réels double précision :

```
#include <stdlib.h> /* déclaration de malloc */

double *ptr; /* pointeur vers le début de la zone */
ptr = malloc(10 * sizeof(double)); /* initialisation */
if (ptr == NULL) /* plus de mémoire disponible */
    /* traitement de l'erreur */
    .....
free(ptr); /* on n'a plus besoin de la zone mémoire */
.....
```


Allocation dynamique (1/3)

- La primitive système `malloc`, de prototype `void *malloc(size_t)`, permet d'allouer *dynamiquement* une zone mémoire de taille en octets le paramètre de `malloc`. Lorsque cette zone n'est plus utilisée au sein du programme, il faut la désallouer à l'aide de la primitive `free` de prototype `void free(void *)`.
- Par exemple, pour allouer une zone mémoire destinée à accueillir dix réels double précision :

```
#include <stdlib.h> /* déclaration de malloc */

double *ptr; /* pointeur vers le début de la zone */
ptr = malloc(10 * sizeof(double)); /* initialisation */
if (ptr == NULL) /* plus de mémoire disponible */
    /* traitement de l'erreur */
.....
free(ptr); /* on n'a plus besoin de la zone mémoire */
.....
```

Allocation dynamique (2/3)

- Plus généralement pour allouer `n` objets de type `type` :

```
type *obj;  
obj = malloc(n * sizeof(type));  
if (obj == NULL)  
    /* traitement de l'erreur */
```

- Les `n` objets alloués sont contigus en mémoire. Par conséquent, pour parcourir cette zone mémoire :

```
for (type *ptr = obj; ptr - obj < n; ptr++)  
    .....
```

- Un autre exemple, créer et initialiser à 0 une zone de dix entiers :

```
int *i;  
i = malloc(10 * sizeof(int));  
for (int *ptr = i; ptr - i < 10; ptr++) *ptr = 0;
```

Une variante de la boucle précédente :

```
for (int j = 0; j < 10; j++) i[j] = 0;
```

Allocation dynamique (2/3)

- Plus généralement pour allouer `n` objets de type `type` :

```
type *obj;  
obj = malloc(n * sizeof(type));  
if (obj == NULL)  
    /* traitement de l'erreur */
```

- Les `n` objets alloués sont contigus en mémoire. Par conséquent, pour parcourir cette zone mémoire :

```
for (type *ptr = obj; ptr - obj < n; ptr++)  
    .....
```

- Un autre exemple, créer et initialiser à 0 une zone de dix entiers :

```
int *i;  
i = malloc(10 * sizeof(int));  
for (int *ptr = i; ptr - i < 10; ptr++) *ptr = 0;
```

Une variante de la boucle précédente :

```
for (int j = 0; j < 10; j++) i[j] = 0;
```

Allocation dynamique (2/3)

- Plus généralement pour allouer `n` objets de type `type` :

```
type *obj;  
obj = malloc(n * sizeof(type));  
if (obj == NULL)  
    /* traitement de l'erreur */
```

- Les `n` objets alloués sont contigus en mémoire. Par conséquent, pour parcourir cette zone mémoire :

```
for (type *ptr = obj; ptr - obj < n; ptr++)  
    .....
```

- Un autre exemple, créer et initialiser à 0 une zone de dix entiers :

```
int *i;  
i = malloc(10 * sizeof(int));  
for (int *ptr = i; ptr - i < 10; ptr++) *ptr = 0;
```

Une variante de la boucle précédente :

```
for (int j = 0; j < 10; j++) i[j] = 0;
```

Allocation dynamique (2/3)

- Plus généralement pour allouer `n` objets de type `type` :

```
type *obj;  
obj = malloc(n * sizeof(type));  
if (obj == NULL)  
    /* traitement de l'erreur */
```

- Les `n` objets alloués sont contigus en mémoire. Par conséquent, pour parcourir cette zone mémoire :

```
for (type *ptr = obj; ptr - obj < n; ptr++)  
    .....
```

- Un autre exemple, créer et initialiser à 0 une zone de dix entiers :

```
int *i;  
i = malloc(10 * sizeof(int));  
for (int *ptr = i; ptr - i < 10; ptr++) *ptr = 0;
```

Une variante de la boucle précédente :

```
for (int j = 0; j < 10; j++) i[j] = 0;
```

Allocation dynamique (3/3)

- Une autre primitive d'allocation mémoire : `calloc` de prototype `void *calloc(size_t, size_t)`. Elle est identique à la primitive `malloc` à cela près que la zone allouée est **initialisée à 0**. Pour allouer et initialiser une zone mémoire de dix entiers :

```
int *i;  
i = calloc(10, sizeof(int));
```

- Pour retailer une zone mémoire : `realloc` de prototype `void *realloc(void *, size_t)`. Si on veut étendre à 20 entiers la zone mémoire de l'exemple précédent :

```
int *j; /* pour ne pas perdre la valeur de i  
        * en cas d'échec de realloc */  
j = realloc(i, 20);  
if (j == NULL)  
    /* traitement de l'erreur */  
else  
    i = j; /* j peut être différent de i */
```

Allocation dynamique (3/3)

- Une autre primitive d'allocation mémoire : `calloc` de prototype `void *calloc(size_t, size_t)`. Elle est identique à la primitive `malloc` à cela près que la zone allouée est **initialisée à 0**. Pour allouer et initialiser une zone mémoire de dix entiers :

```
int *i;  
i = calloc(10, sizeof(int));
```

- Pour retailer une zone mémoire : `realloc` de prototype `void *realloc(void *, size_t)`. Si on veut étendre à 20 entiers la zone mémoire de l'exemple précédent :

```
int *j; /* pour ne pas perdre la valeur de i  
        * en cas d'échec de realloc */  
j = realloc(i, 20);  
if (j == NULL)  
    /* traitement de l'erreur */  
else  
    i = j; /* j peut être différent de i */
```

Passage de paramètres par **valeur** (1/2)

```
void f(int n)
{
    n = n + n;
}

int main(void)
{
    int i = 2;

    f(i);
    printf("i=%d\n", i);
    return 0;
}
```


Passage de paramètres par **valeur** (1/2)

```
void f(int n)
{
    n = n + n;
}

int main(void)
{
    int i = 2;

    f(i);
    printf("i=%d\n", i);
    return 0;
}
```

Sortie du programme :

```
i = 2
```

Passage de paramètres par **valeur** (1/2)

```
void f(int n)
{
    n = n + n;
}

int main(void)
{
    int i = 2;

    f(i);
    printf("i=%d\n", i);
    return 0;
}
```

La valeur de *i* est inchangée !

Sortie du programme :

i = 2

Passage de paramètres par **valeur** (1/2)

```
void f(int n)
{
    n = n + n;
}
```

Que s'est-il passé lors de l'exécution de **f** ?

```
int main(void)
{
    int i = 2;

    f(i);
    printf("i = %d\n", i);
    return 0;
}
```

Sortie du programme :

```
i = 2
```

Passage de paramètres par **valeur** (1/2)

```
void f(int n)
{
    n = n + n;
}
```

f crée une variable temporaire dans laquelle est copiée la valeur de i.

```
int main(void)
{
    int i = 2;

    f(i);
    printf("i = %d\n", i);
    return 0;
}
```

Sortie du programme :

```
i = 2
```

Passage de paramètres par **valeur** (1/2)

```
void f(int n)
{
    n = n + n;
}
```

f crée une variable temporaire dans laquelle est copiée la valeur de i.

```
int main(void)
{
    int i = 2;

    f(i);
    printf("i=%d\n", i);
    return 0;
}
```

n 2

Sortie du programme :

i = 2

Passage de paramètres par **valeur** (1/2)

```
void f(int n)
{
    n = n + n;
}
```

L'instruction `n = n + n;` est exécutée,
`n` vaut maintenant 4.

`n` 4

```
int main(void)
{
    int i = 2;

    f(i);
    printf("i = %d\n", i);
    return 0;
}
```

Sortie du programme :

```
i = 2
```

Passage de paramètres par **valeur** (1/2)

```
void f(int n)
{
    n = n + n;
}
```

La fonction `f` se termine, la variable temporaire `n` est alors détruite.

```
int main(void)
{
    int i = 2;

    f(i);
    printf("i = %d\n", i);
    return 0;
}
```

Sortie du programme :

```
i = 2
```

Passage de paramètres par **valeur** (1/2)

```
void f(int n)
{
    n = n + n;
}
```

La fonction `f` se termine, la variable temporaire `n` est alors détruite.

```
int main(void)
{
    int i = 2;

    f(i);
    printf("i = %d\n", i);
    return 0;
}
```

Ainsi la valeur de `i` n'a pas été affectée par l'exécution de la fonction `f`.

Sortie du programme :

```
i = 2
```


Passage de paramètres par **valeur** (1/2)

```
void f(int n)
{
    n = n + n;
}
```

La fonction `f` se termine, la variable temporaire `n` est alors détruite.

```
int main(void)
{
    int i = 2;

    f(i);
    printf("i = %d\n", i);
    return 0;
}
```

Ainsi la valeur de `i` n'a pas été affectée par l'exécution de la fonction `f`.

Il aurait fallu passer en paramètre à `f`, non pas la valeur de `i`, mais l'adresse de `i` \implies utilisation des pointeurs.

Sortie du programme :

```
i = 2
```

Passage de paramètres par **valeur** (2/2)

```
void f(int *n)
{
    *n = *n + *n;
}

int main(void)
{
    int i = 2;

    f(&i);
    printf("i = %d\n", i);
    return 0;
}
```

Passage de paramètres par **valeur** (2/2)

```
void f(int *n)
{
    *n = *n + *n;
}

int main(void)
{
    int i = 2;

    f(&i);
    printf("i=%d\n", i);
    return 0;
}
```

Sortie du programme :

```
i = 4
```

Passage de paramètres par **valeur** (2/2)

```
void f(int *n)
{
    *n = *n + *n;
}
```

f crée une variable temporaire dans laquelle est copiée la valeur de &i.

```
int main(void)
{
    int i = 2;

    f(&i);
    printf("i = %d\n", i);
    return 0;
}
```

Sortie du programme :

```
i = 4
```

Passage de paramètres par **valeur** (2/2)

```
void f(int *n)
{
    *n = *n + *n;
}

int main(void)
{
    int i = 2;

    f(&i);
    printf("i = %d\n", i);
    return 0;
}
```

f crée une variable temporaire dans laquelle est copiée la valeur de &i.

n 0x79f2503dc6a28292

Sortie du programme :

i = 4

Passage de paramètres par **valeur** (2/2)

```
void f(int *n)
{
    *n = *n + *n;
}
```

Ainsi le pointeur `n`, de type `int`,
pointe sur `i`,

```
int main(void)
{
    int i = 2;

    f(&i);
    printf("i = %d\n", i);
    return 0;
}
```

0x79f2503dc6a28292

n pointe sur i

2

Sortie du programme :

i = 4

Passage de paramètres par **valeur** (2/2)

```
void f(int *n)
{
    *n = *n + *n;
}

int main(void)
{
    int i = 2;

    f(&i);
    printf("i = %d\n", i);
    return 0;
}
```

Ainsi le pointeur `n`, de type `int`,
pointe sur `i`, avec `*n` valant 2.

0x79f2503dc6a28292

n pointe sur i

2

Sortie du programme :

i = 4

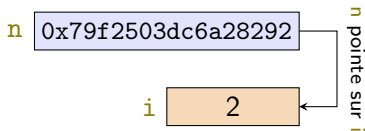
Passage de paramètres par **valeur** (2/2)

```
void f(int *n)
{
    *n = *n + *n;
}
```

L'instruction `*n = *n + *n;` est exécutée,

```
int main(void)
{
    int i = 2;

    f(&i);
    printf("i = %d\n", i);
    return 0;
}
```



Sortie du programme :

```
i = 4
```

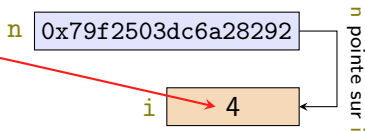

Passage de paramètres par **valeur** (2/2)

```
void f(int *n)
{
    *n = *n + *n;
}
```

L'instruction `*n = *n + *n;` est exécutée, `i` vaut maintenant 4.

```
int main(void)
{
    int i = 2;

    f(&i);
    printf("i = %d\n", i);
    return 0;
}
```



Sortie du programme :

```
i = 4
```

Passage de paramètres par **valeur** (2/2)

```
void f(int *n)
{
    *n = *n + *n;
}
```

La fonction `f` se termine, la variable temporaire `n` est alors détruite.

```
int main(void)
{
    int i = 2;

    f(&i);
    printf("i = %d\n", i);
    return 0;
}
```

Sortie du programme :

```
i = 4
```

Passage de paramètres par **valeur** (2/2)

```
void f(int *n)
{
    *n = *n + *n;
}
```

La fonction `f` se termine, la variable temporaire `n` est alors détruite.

```
int main(void)
{
    int i = 2;

    f(&i);
    printf("i = %d\n", i);
    return 0;
}
```

Cette fois ci, la valeur de `i` a été affectée par l'exécution de la fonction `f`.

Sortie du programme :

```
i = 4
```

Un exemple un peu moins trivial (1/2)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
/* Une fonction qui copie une chaîne de caractères */
char *string_copy(char *in)
{ /* appel : if ((out = string_copy(in)) == NULL)... */
    char *out;

    if (in == NULL)
        return NULL; /* ce n'est pas une erreur */
    out = malloc(strlen(in) + 1);
    if (out == NULL) {
        perror("malloc");
        return NULL; /* erreur sauf qu'on retourne également
                       * le pointeur nul comme ci-dessus */
    }
    for (char *p = in, *q = out; (*q++ = *p++));
    return out;
}
```

Un exemple un peu moins trivial (2/2)

- La copie de la chaîne de caractères est entièrement effectuée lors de l'exécution de l'instruction :

```
for (char *p = in, *q = out; (*q++ = *p++););
```

- La partie déclaration + initialisation : `char *p = in, *q = out;` ne présente aucune difficulté.
- La copie s'effectue lors de l'évaluation de la condition d'arrêt : `(*q++ = *p++);`. Rappelons tout d'abord que :

$$*q++ = *p++; \iff *q = *p; q += 1; p += 1;$$

- Par ailleurs l'expression `(*q++ = *p++);` a pour valeur la valeur affectée, ici la valeur de `*p`.
- On sortira donc de la boucle lorsque `*p` sera nul, c'est à dire lorsque `p` pointera sur le caractère (nul) de fin de chaîne. On aura bien ainsi recopié la chaîne originale.

Un exemple un peu moins trivial (2/2)

- La copie de la chaîne de caractères est entièrement effectuée lors de l'exécution de l'instruction :

```
for (char *p = in, *q = out; (*q++ = *p++););
```

- La partie déclaration + initialisation : `char *p = in, *q = out;` ne présente aucune difficulté.

- La copie s'effectue lors de l'évaluation de la condition d'arrêt : `(*q++ = *p++);`. Rappelons tout d'abord que :

```
*q++ = *p++;  $\iff$  *q = *p; q +=1; p +=1;
```

- Par ailleurs l'expression `(*q++ = *p++);` a pour valeur la valeur affectée, ici la valeur de `*p`.
- On sortira donc de la boucle lorsque `*p` sera nul, c'est à dire lorsque `p` pointera sur le caractère (nul) de fin de chaîne. On aura bien ainsi recopié la chaîne originale.

Un exemple un peu moins trivial (2/2)

- La copie de la chaîne de caractères est entièrement effectuée lors de l'exécution de l'instruction :

```
for (char *p = in, *q = out; (*q++ = *p++););
```

- La partie déclaration + initialisation : `char *p = in, *q = out;` ne présente aucune difficulté.
- La copie s'effectue lors de l'évaluation de la condition d'arrêt : `(*q++ = *p++);`. Rappelons tout d'abord que :

$$*q++ = *p++; \iff *q = *p; q += 1; p += 1;$$

- Par ailleurs l'expression `(*q++ = *p++);` a pour valeur la valeur affectée, ici la valeur de `*p`.
- On sortira donc de la boucle lorsque `*p` sera nul, c'est à dire lorsque `p` pointera sur le caractère (nul) de fin de chaîne. On aura bien ainsi recopié la chaîne originale.

Un exemple un peu moins trivial (2/2)

- La copie de la chaîne de caractères est entièrement effectuée lors de l'exécution de l'instruction :

```
for (char *p = in, *q = out; (*q++ = *p++););
```

- La partie déclaration + initialisation : `char *p = in, *q = out;` ne présente aucune difficulté.

- La copie s'effectue lors de l'évaluation de la condition d'arrêt : `(*q++ = *p++);`. Rappelons tout d'abord que :

```
*q++ = *p++;  $\iff$  *q = *p; q +=1; p +=1;
```

- Par ailleurs l'expression `(*q++ = *p++);` a pour valeur la valeur affectée, ici la valeur de `*p`.
- On sortira donc de la boucle lorsque `*p` sera nul, c'est à dire lorsque `p` pointera sur le caractère (nul) de fin de chaîne. On aura bien ainsi recopié la chaîne originale.

Un exemple un peu moins trivial (2/2)

- La copie de la chaîne de caractères est entièrement effectuée lors de l'exécution de l'instruction :

```
for (char *p = in, *q = out; (*q++ = *p++););
```

- La partie déclaration + initialisation : `char *p = in, *q = out;` ne présente aucune difficulté.

- La copie s'effectue lors de l'évaluation de la condition d'arrêt : `(*q++ = *p++);`. Rappelons tout d'abord que :

```
*q++ = *p++;  $\iff$  *q = *p; q +=1; p +=1;
```

- Par ailleurs l'expression `(*q++ = *p++);` a pour valeur la valeur affectée, ici la valeur de `*p`.
- On sortira donc de la boucle lorsque `*p` sera nul, c'est à dire lorsque `p` pointera sur le caractère (nul) de fin de chaîne. On aura bien ainsi recopié la chaîne originale.

Un exemple un peu moins trivial – Variante (1/2)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int string_copy(char *in, char **out)
{ /* appel : if ((string_copy(in, &out) == -1)... */
    if (in == NULL) {
        *out = NULL;
        return 0; /* ce n'est pas une erreur */
    }
    *out = malloc(strlen(in) + 1);
    if (*out == NULL) {
        perror("malloc");
        return -1; /* ici, il y a eu une erreur */
    }
    for (char *p = in, *q = *out; (*q++ = *p++));
    return 0; /* sortie normale de la fonction */
}
```

Un exemple un peu moins trivial – Variante (2/2)

```
/* compilé avec : gcc -Wall str.c -o str */  
#include <stdio.h>  
  
int main(int argc, char **argv)  
{  
    char *str;  
  
    if (string_copy(*argv, &str) < 0)  
        return 1;  
    printf("%s\n", str);  
    return 0;  
}
```

Un exemple un peu moins trivial – Variante (2/2)

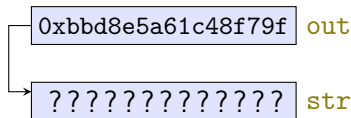
`string_copy` crée une variable temporaire `out`, de type `char **` dans laquelle est copiée la valeur de `&str` (donc `out` pointe sur `str`).

```
/* compilé avec : gcc -Wall str.c -o str */
```

```
#include <stdio.h>
```

```
int main(int argc, char **argv)
{
    char *str;

    if (string_copy(*argv, &str) < 0)
        return 1;
    printf("%s\n", str);
    return 0;
}
```



Un exemple un peu moins trivial – Variante (2/2)

Dans `string_copy`, l'instruction `*out = malloc(strlen(in) + 1);` alloue une zone mémoire (non initialisée) de 4 octets.

```
/* compilé avec : gcc -Wall str.c -o str */
```

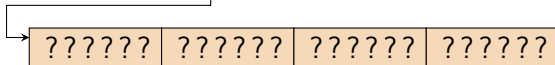
```
#include <stdio.h>
```

```
int main(int argc, char **argv)
{
    char *str;

    if (string_copy(*argv, &str) < 0)
        return 1;
    printf("%s\n", str);
    return 0;
}
```

```
0xbbd8e5a61c48f79f out
```

```
0x3a202f36df539d07  str
```



Un exemple un peu moins trivial – Variante (2/2)

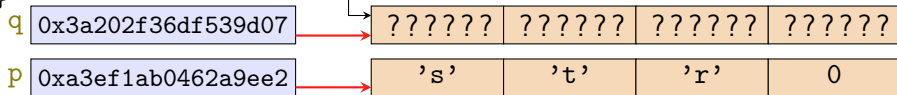
Ensuite, `for (char *p = in, *q = *out; (*q++ = *p++););`
copie la chaîne pointée par `in` dans la zone mémoire allouée.

```
/* compilé avec : gcc -Wall str.c -o str */
```

```
#include <stdio.h>
```

```
int main(int argc, char **argv)
{
    char *str;

    if (string_copy(*argv, &str) < 0)
        return 1;
    printf("%s\n", str);
    return 0;
}
```



Un exemple un peu moins trivial – Variante (2/2)

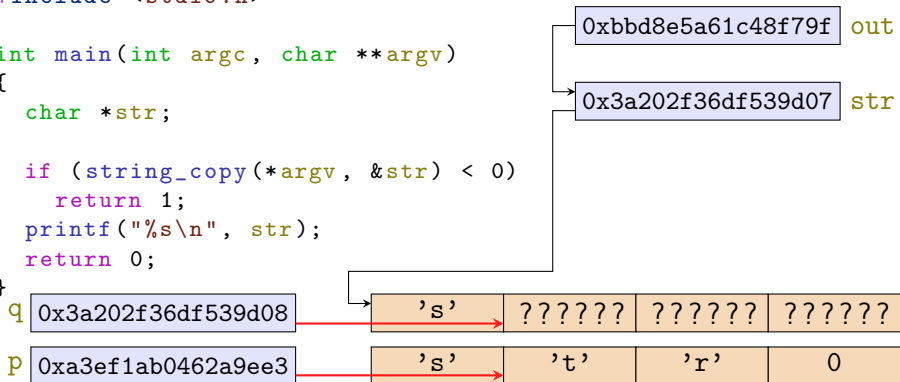
Ensuite, `for (char *p = in, *q = *out; (*q++ = *p++););`
copie la chaîne pointée par `in` dans la zone mémoire allouée.

```
/* compilé avec : gcc -Wall str.c -o str */
```

```
#include <stdio.h>
```

```
int main(int argc, char **argv)
{
    char *str;

    if (string_copy(*argv, &str) < 0)
        return 1;
    printf("%s\n", str);
    return 0;
}
```



Un exemple un peu moins trivial – Variante (2/2)

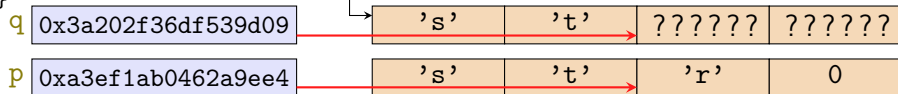
Ensuite, `for (char *p = in, *q = *out; (*q++ = *p++););`
copie la chaîne pointée par `in` dans la zone mémoire allouée.

```
/* compilé avec : gcc -Wall str.c -o str */
```

```
#include <stdio.h>
```

```
int main(int argc, char **argv)
{
    char *str;

    if (string_copy(*argv, &str) < 0)
        return 1;
    printf("%s\n", str);
    return 0;
}
```



Un exemple un peu moins trivial – Variante (2/2)

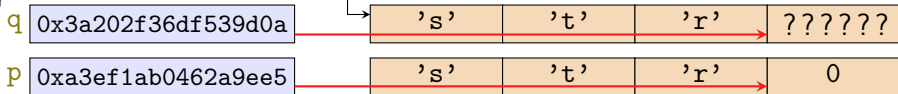
Ensuite, `for (char *p = in, *q = *out; (*q++ = *p++););`
copie la chaîne pointée par `in` dans la zone mémoire allouée.

```
/* compilé avec : gcc -Wall str.c -o str */
```

```
#include <stdio.h>
```

```
int main(int argc, char **argv)
{
    char *str;

    if (string_copy(*argv, &str) < 0)
        return 1;
    printf("%s\n", str);
    return 0;
}
```



Un exemple un peu moins trivial – Variante (2/2)

Ensuite, `for (char *p = in, *q = *out; (*q++ = *p++););`
copie la chaîne pointée par `in` dans la zone mémoire allouée.

```
/* compilé avec : gcc -Wall str.c -o str */
```

```
#include <stdio.h>
```

```
int main(int argc, char **argv)
{
    char *str;

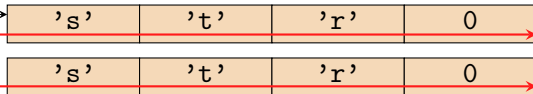
    if (string_copy(*argv, &str) < 0)
        return 1;
    printf("%s\n", str);
    return 0;
}
```

q 0x3a202f36df539d0b

p 0xa3ef1ab0462a9ee6

0xbbd8e5a61c48f79f out

0x3a202f36df539d07 str



Un exemple un peu moins trivial – Variante (2/2)

Au final `str` pointe sur une zone mémoire allouée dynamiquement (ne pas oublier `free(str);`) et contenant une copie de la chaîne pointée par `*argv`.

```
/* compilé avec : gcc -Wall str.c -o str */
```

```
#include <stdio.h>
```

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

```
{
```

```
    char *str;
```

```
    if (string_copy(*argv, &str) < 0)
```

```
        return 1;
```

```
    printf("%s\n", str);
```

```
    return 0;
```

```
}
```

0x3a202f36df539d07 str

's'

't'

'r'

0

Pointeurs constants et pointeurs vers des constantes (1/2)

- Considérons les déclarations suivantes :

```
const char *ptr;
```

Le pointeur `ptr` est un **pointeur vers une constante** de type `char`, ce qui signifie que, via l'opérateur d'indirection, la valeur de l'objet pointé par `ptr` ne peut être modifiée. Autrement dit :

```
*ptr = 'C' /* instruction illégale */
```

- La déclaration qui suit :

```
char * const ptr;
```

indique que le **pointeur `ptr` est constant** :

```
*ptr = 'C' /* instruction valide */  
ptr++;    /* instruction illégale */
```

- On peut combiner les deux déclarations précédentes :

```
const char * const ptr;
```

au quel cas tout est constant : le pointeur et la valeur pointée.

Pointeurs constants et pointeurs vers des constantes (1/2)

- Considérons les déclarations suivantes :

```
const char *ptr;
```

Le pointeur `ptr` est un **pointeur vers une constante** de type `char`, ce qui signifie que, via l'opérateur d'indirection, la valeur de l'objet pointé par `ptr` ne peut être modifiée. Autrement dit :

```
*ptr = 'C' /* instruction illégale */
```

- La déclaration qui suit :

```
char * const ptr;
```

indique que le **pointeur `ptr` est constant** :

```
*ptr = 'C' /* instruction valide */  
ptr++;    /* instruction illégale */
```

- On peut combiner les deux déclarations précédentes :

```
const char * const ptr;
```

au quel cas tout est constant : le pointeur et la valeur pointée.

Pointeurs constants et pointeurs vers des constantes (1/2)

- Considérons les déclarations suivantes :

```
const char *ptr;
```

Le pointeur `ptr` est un **pointeur vers une constante** de type `char`, ce qui signifie que, via l'opérateur d'indirection, la valeur de l'objet pointé par `ptr` ne peut être modifiée. Autrement dit :

```
*ptr = 'C' /* instruction illégale */
```

- La déclaration qui suit :

```
char * const ptr;
```

indique que le **pointeur `ptr` est constant** :

```
*ptr = 'C' /* instruction valide */  
ptr++;    /* instruction illégale */
```

- On peut combiner les deux déclarations précédentes :

```
const char * const ptr;
```

au quel cas tout est constant : le pointeur et la valeur pointée.

Pointeurs constants et pointeurs vers des constantes (2/2)

- L'utilisation des pointeurs vers des constantes n'est pas obligatoire mais recommandé. C'est un outil de **contrôle** supplémentaire pour le programmeur.
- Cet outil est particulièrement utile pour préciser le type de certains paramètres de fonctions lorsque ceux-ci sont des pointeurs.
- Par exemple, considérons la fonction suivante copiant des données de type générique (type `void`) :

```
void *copy_data(const void *data);
```

Le fait de préciser que `data` est un pointeur vers une constante assure que les données **ne seront pas modifiées** lors de la copie.

- Autre exemple, les fonctions d'impression :

```
void print_data(const void *data);
```

Là aussi, les données à imprimer ne doivent pas être modifiées.

Pointeurs constants et pointeurs vers des constantes (2/2)

- L'utilisation des pointeurs vers des constantes n'est pas obligatoire mais recommandé. C'est un outil de **contrôle** supplémentaire pour le programmeur.
- Cet outil est particulièrement utile pour préciser le type de certains paramètres de fonctions lorsque ceux-ci sont des pointeurs.
- Par exemple, considérons la fonction suivante copiant des données de type générique (type `void`) :

```
void *copy_data(const void *data);
```

Le fait de préciser que `data` est un pointeur vers une constante assure que les données **ne seront pas modifiées** lors de la copie.

- Autre exemple, les fonctions d'impression :

```
void print_data(const void *data);
```

Là aussi, les données à imprimer ne doivent pas être modifiées.

Pointeurs constants et pointeurs vers des constantes (2/2)

- L'utilisation des pointeurs vers des constantes n'est pas obligatoire mais recommandé. C'est un outil de **contrôle** supplémentaire pour le programmeur.
- Cet outil est particulièrement utile pour préciser le type de certains paramètres de fonctions lorsque ceux-ci sont des pointeurs.
- Par exemple, considérons la fonction suivante copiant des données de type générique (type `void`) :

```
void *copy_data(const void *data);
```

Le fait de préciser que `data` est un pointeur vers une constante assure que les données **ne seront pas modifiées** lors de la copie.

- Autre exemple, les fonctions d'impression :

```
void print_data(const void *data);
```

Là aussi, les données à imprimer ne doivent pas être modifiées.

Pointeurs constants et pointeurs vers des constantes (2/2)

- L'utilisation des pointeurs vers des constantes n'est pas obligatoire mais recommandé. C'est un outil de **contrôle** supplémentaire pour le programmeur.
- Cet outil est particulièrement utile pour préciser le type de certains paramètres de fonctions lorsque ceux-ci sont des pointeurs.
- Par exemple, considérons la fonction suivante copiant des données de type générique (type `void`) :

```
void *copy_data(const void *data);
```

Le fait de préciser que `data` est un pointeur vers une constante assure que les données **ne seront pas modifiées** lors de la copie.

- Autre exemple, les fonctions d'impression :

```
void print_data(const void *data);
```

Là aussi, les données à imprimer ne doivent pas être modifiées.

Définition

Un tableau est un ensemble d'objets C (à priori quelconques) rangés en mémoire à des adresses contiguës. Il se déclare comme suit :

```
type tab[N];
```

où tab est le nom du tableau et N , un entier positif ou nul, sa dimension, c'est à dire le nombre de ses éléments. En outre, $type \neq void$.

- L'accès aux éléments du tableau se fait à l'aide l'opérateur `[]` :

```
tab[0], ..., tab[N-1]
```

- L'objet tab est un pointeur (constant) vers le premier élément, autrement dit : $tab \iff \&tab[0]$.
- Un petit schéma pour visualiser tout cela (ici N vaut 9) :

Définition

Un tableau est un ensemble d'objets C (à priori quelconques) rangés en mémoire à des adresses contiguës. Il se déclare comme suit :

```
type tab[N];
```

où tab est le nom du tableau et N , un entier positif ou nul, sa dimension, c'est à dire le nombre de ses éléments. En outre, $type \neq void$.

- L'accès aux éléments du tableau se fait à l'aide l'opérateur `[]` :

```
tab[0], ..., tab[N-1]
```

- L'objet tab est un pointeur (constant) vers le premier élément, autrement dit : $tab \iff \&tab[0]$.
- Un petit schéma pour visualiser tout cela (ici N vaut 9) :

Définition

Un tableau est un ensemble d'objets C (à priori quelconques) rangés en mémoire à des adresses contiguës. Il se déclare comme suit :

```
type tab[N];
```

*où **tab** est le nom du tableau et **N**, un entier positif ou nul, sa dimension, c'est à dire le nombre de ses éléments. En outre, **type** \neq **void**.*

- L'accès aux éléments du tableau se fait à l'aide l'opérateur [] :

```
tab[0], ..., tab[N-1]
```

- L'objet **tab** est un pointeur (constant) vers le premier élément, autrement dit : **tab** \iff **&tab[0]**.
- Un petit schéma pour visualiser tout cela (ici **N** vaut 9) :

Définition

Un tableau est un ensemble d'objets C (à priori quelconques) rangés en mémoire à des adresses contiguës. Il se déclare comme suit :

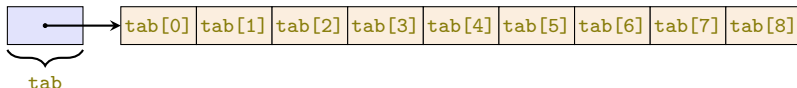
```
type tab[N];
```

où tab est le nom du tableau et N , un entier positif ou nul, sa dimension, c'est à dire le nombre de ses éléments. En outre, $type \neq void$.

- L'accès aux éléments du tableau se fait à l'aide l'opérateur `[]` :

```
tab[0], ..., tab[N-1]
```

- L'objet tab est un pointeur (constant) vers le premier élément, autrement dit : $tab \iff \&tab[0]$.
- Un petit schéma pour visualiser tout cela (ici N vaut 9) :



Parcours d'un tableau

Il y a essentiellement deux manières de parcourir un tableau. Par exemple, pour initialiser un tableau d'entiers à zéro :

- `/* on utilise l'opérateur d'indexation [] */`
`int tab[10];`
`for (int i = 0; i < sizeof(tab)/sizeof(int); i++)`
 `tab[i] = 0;`
- `/* on utilise un pointeur vers le tableau */`
`int tab[10];`
`for (int *i = tab; i - tab < sizeof(tab)/sizeof(int); i++)`
 `*i = 0;`
- On remarquera que l'on a calculé le nombre d'éléments du tableau (`sizeof(tab)/sizeof(int)`) plutôt que de prendre la constante 10. C'est une bonne habitude à prendre, notamment lorsque c'est le compilateur qui calcule ce nombre **automatiquement** lors d'une initialisation (notion que l'on verra plus loin).

Parcours d'un tableau

Il y a essentiellement deux manières de parcourir un tableau. Par exemple, pour initialiser un tableau d'entiers à zéro :

- ```
/* on utilise l'opérateur d'indexation [] */
int tab[10];
for (int i = 0; i < sizeof(tab)/sizeof(int); i++)
 tab[i] = 0;
```
- ```
/* on utilise un pointeur vers le tableau */  
int tab[10];  
for (int *i = tab; i - tab < sizeof(tab)/sizeof(int); i++)  
    *i = 0;
```
- On remarquera que l'on a calculé le nombre d'éléments du tableau (`sizeof(tab)/sizeof(int)`) plutôt que de prendre la constante 10. C'est une bonne habitude à prendre, notamment lorsque c'est le compilateur qui calcule ce nombre **automatiquement** lors d'une initialisation (notion que l'on verra plus loin).

Parcours d'un tableau

Il y a essentiellement deux manières de parcourir un tableau. Par exemple, pour initialiser un tableau d'entiers à zéro :

- `/* on utilise l'opérateur d'indexation [] */`
`int tab[10];`
`for (int i = 0; i < sizeof(tab)/sizeof(int); i++)`
 `tab[i] = 0;`
- `/* on utilise un pointeur vers le tableau */`
`int tab[10];`
`for (int *i = tab; i - tab < sizeof(tab)/sizeof(int); i++)`
 `*i = 0;`
- On remarquera que l'on a calculé le nombre d'éléments du tableau (`sizeof(tab)/sizeof(int)`) plutôt que de prendre la constante 10. C'est une bonne habitude à prendre, notamment lorsque c'est le compilateur qui calcule ce nombre **automatiquement** lors d'une initialisation (notion que l'on verra plus loin).

Parcours d'un tableau

Il y a essentiellement deux manières de parcourir un tableau. Par exemple, pour initialiser un tableau d'entiers à zéro :

- ```
/* on utilise l'opérateur d'indexation [] */
int tab[10];
for (int i = 0; i < sizeof(tab)/sizeof(int); i++)
 tab[i] = 0;
```
- ```
/* on utilise un pointeur vers le tableau */  
int tab[10];  
for (int *i = tab; i - tab < sizeof(tab)/sizeof(int); i++)  
    *i = 0;
```
- On remarquera que l'on a calculé le nombre d'éléments du tableau (`sizeof(tab)/sizeof(int)`) plutôt que de prendre la constante 10. C'est une bonne habitude à prendre, notamment lorsque c'est le compilateur qui calcule ce nombre **automatiquement** lors d'une initialisation (notion que l'on verra plus loin).

Initialisation d'un tableau

- Lors de sa déclaration, un tableau peut être initialisé :

```
type tab[N] = {C0, ..., CN-1};
```

où C_0, \dots, C_{N-1} sont N constantes de type `type`.

- Par exemple :

```
int tab[4] = {1, 8, 7, 6};  
char str[8] = {'C', 'o', 'u', 'r', 's', ' ', 'C', 0};
```

Dans ce cas, la dimension du tableau peut être omise, elle est alors calculée par le compilateur :

```
int tab[] = {1, 8, 7, 6};  
char str[] = {'C', 'o', 'u', 'r', 's', ' ', 'C', 0};
```

- Une initialisation partielle est également possible :

```
int tab[20] = {10, 5, 4};
```

Ici, seuls les premiers éléments seront initialisés. Les autres seront initialisés à 0 si `tab` est une variable globale ou une variable locale de la classe `static`.

Initialisation d'un tableau

- Lors de sa déclaration, un tableau peut être initialisé :

```
type tab[N] = {C0, ..., CN-1};
```

où C_0, \dots, C_{N-1} sont N constantes de type `type`.

- Par exemple :

```
int tab[4] = {1, 8, 7, 6};  
char str[8] = {'C', 'o', 'u', 'r', 's', ' ', 'C', 0};
```

Dans ce cas, la dimension du tableau peut être omise, elle est alors calculée par le compilateur :

```
int tab[] = {1, 8, 7, 6};  
char str[] = {'C', 'o', 'u', 'r', 's', ' ', 'C', 0};
```

- Une initialisation partielle est également possible :

```
int tab[20] = {10, 5, 4};
```

Ici, seuls les premiers éléments seront initialisés. Les autres seront initialisés à 0 si `tab` est une variable globale ou une variable locale de la classe `static`.

Initialisation d'un tableau

- Lors de sa déclaration, un tableau peut être initialisé :

```
type tab[N] = {C0, ..., CN-1};
```

où C_0, \dots, C_{N-1} sont N constantes de type `type`.

- Par exemple :

```
int tab[4] = {1, 8, 7, 6};  
char str[8] = {'C', 'o', 'u', 'r', 's', ' ', 'C', 0};
```

Dans ce cas, la dimension du tableau **peut être omise**, elle est alors calculée par le compilateur :

```
int tab[] = {1, 8, 7, 6};  
char str[] = {'C', 'o', 'u', 'r', 's', ' ', 'C', 0};
```

- Une initialisation partielle est également possible :

```
int tab[20] = {10, 5, 4};
```

Ici, seuls les premiers éléments seront initialisés. Les autres seront initialisés à 0 si `tab` est une variable globale ou une variable locale de la classe `static`.

Initialisation d'un tableau

- Lors de sa déclaration, un tableau peut être initialisé :

```
type tab[N] = {C0, ..., CN-1};
```

où C_0, \dots, C_{N-1} sont N constantes de type `type`.

- Par exemple :

```
int tab[4] = {1, 8, 7, 6};  
char str[8] = {'C', 'o', 'u', 'r', 's', ' ', 'C', 0};
```

Dans ce cas, la dimension du tableau **peut être omise**, elle est alors calculée par le compilateur :

```
int tab[] = {1, 8, 7, 6};  
char str[] = {'C', 'o', 'u', 'r', 's', ' ', 'C', 0};
```

- Une initialisation partielle est également possible :

```
int tab[20] = {10, 5, 4};
```

Ici, seuls les premiers éléments seront initialisés. Les autres seront initialisés à 0 si `tab` est une variable globale ou une variable locale de la classe `static`.

Chaînes de caractères

- Rappelons qu'une constante chaîne de caractères est un tableau de caractères (type `char`) dont le dernier élément est le caractère nul (entier 0) :

`"Cours"` \iff `{'C', 'o', 'u', 'r', 's', 0}`

```
printf("%c%c", *"Cours", "Cours"[2]); /* correct */  
"Cours"[2] = 't'; /* illégal */  
"Cours"++;      /* illégal */
```

- On peut donc initialiser un tableau de caractères ainsi :

```
char tab[] = "Cours";  
          ⇕  
char tab[] = {'C', 'o', 'u', 'r', 's', 0};
```

- Attention (c'est une erreur fréquente) :

```
char tab[] = "Cours";  $\nRightarrow$  char *tab = "Cours";
```

Chaînes de caractères

- Rappelons qu'une constante chaîne de caractères est un tableau de caractères (type `char`) dont le dernier élément est le caractère nul (entier 0) :

`"Cours"` \iff `{'C', 'o', 'u', 'r', 's', 0}`

```
printf("%c%c", *"Cours", "Cours"[2]); /* correct */
"Cours"[2] = 't'; /* illégal */
"Cours"++;      /* illégal */
```

- On peut donc initialiser un tableau de caractères ainsi :

```
char tab[] = "Cours";
          ⇕
char tab[] = {'C', 'o', 'u', 'r', 's', 0};
```

- Attention (c'est une erreur fréquente) :

```
char tab[] = "Cours";  $\nRightarrow$  char *tab = "Cours";
```


Chaînes de caractères

- Rappelons qu'une constante chaîne de caractères est un tableau de caractères (type `char`) dont le dernier élément est le caractère nul (entier 0) :

`"Cours"` \iff `{ 'C', 'o', 'u', 'r', 's', 0 }`

```
printf("%c%c", *"Cours", "Cours"[2]); /* correct */
"Cours"[2] = 't'; /* illégal */
"Cours"++;      /* illégal */
```

- On peut donc initialiser un tableau de caractères ainsi :

```
char tab[] = "Cours";
           ⇕
char tab[] = { 'C', 'o', 'u', 'r', 's', 0 };
```

- Attention (c'est une erreur fréquente) :

```
char tab[] = "Cours";  $\nRightarrow$  char *tab = "Cours";
```

Chaînes de caractères

- Rappelons qu'une constante chaîne de caractères est un tableau de caractères (type `char`) dont le dernier élément est le caractère nul (entier 0) :

`"Cours"` \iff `{ 'C', 'o', 'u', 'r', 's', 0 }`

```
printf("%c%c", *"Cours", "Cours"[2]); /* correct */
"Cours"[2] = 't'; /* illégal */
"Cours"++;      /* illégal */
```

- On peut donc initialiser un tableau de caractères ainsi :

```
char tab[] = "Cours";
           ⇕
char tab[] = { 'C', 'o', 'u', 'r', 's', 0 };
```

- Attention (c'est une erreur fréquente) :

```
char tab[] = "Cours";  $\iff$  char *tab = "Cours";
```

Chaînes de caractères

- Rappelons qu'une constante chaîne de caractères est un tableau de caractères (type `char`) dont le dernier élément est le caractère nul (entier 0) :

`"Cours"` \iff `{'C', 'o', 'u', 'r', 's', 0}`

```
printf("%c%c", *"Cours", "Cours"[2]); /* correct */
"Cours"[2] = 't'; /* illégal */
"Cours"++;      /* illégal */
```

- On peut donc initialiser un tableau de caractères ainsi :

```
char tab[] = "Cours";
           ⇕
char tab[] = {'C', 'o', 'u', 'r', 's', 0};
```

- Attention** (c'est une erreur fréquente) :

```
char tab[] = "Cours";  $\nRightarrow$  char *tab = "Cours";
```

Tableaux multidimensionnels (1/3)

- La déclaration d'un tableau à k dimensions se fait ainsi :

```
type tab[N1]...[Nk];
```

et l'accès à un élément :

```
tab[i1]...[ik] /* 0 ≤ ij < Ni, j = 1, ..., k */
```

- La déclaration et l'initialisation d'un tableau, à 2 dimensions par exemple, se fait comme suit :

```
type tab[M][N] = {{C0,0, ..., C0,N-1}, ..., {CM-1,0, ..., CM-1,N-1}};
```

- On déclare le tableau à 2 dimensions suivant :

```
int tab[2][3]; /* tableau unidimensionnel à 3 éléments */
```

Quelle est le type de `tab[1]` ? Est-ce un tableau unidimensionnel de 2 ou 3 éléments ? Il est malheureusement facile de se tromper, mais il y a un moyen mnémotechnique simple...

Tableaux multidimensionnels (1/3)

- La déclaration d'un tableau à k dimensions se fait ainsi :

```
type tab[N1]...[Nk];
```

et l'accès à un élément :

```
tab[i1]...[ik] /* 0 ≤ ij < Nj, j = 1, ..., k */
```

- La déclaration et l'initialisation d'un tableau, à 2 dimensions par exemple, se fait comme suit :

```
type tab[M][N] = {{C0,0, ..., C0,N-1}, ..., {CM-1,0, ..., CM-1,N-1}};
```

- On déclare le tableau à 2 dimensions suivant :

```
int tab[2][3]; /* tableau unidimensionnel à 3 éléments */
```

Quelle est le type de `tab[1]` ? Est-ce un tableau unidimensionnel de 2 ou 3 éléments ? Il est malheureusement facile de se tromper, mais il y a un moyen mnémotechnique simple...

Tableaux multidimensionnels (1/3)

- La déclaration d'un tableau à k dimensions se fait ainsi :

```
type tab[N1]...[Nk];
```

et l'accès à un élément :

```
tab[i1]...[ik] /* 0 ≤ ij < Nj, j = 1, ..., k */
```

- La déclaration et l'initialisation d'un tableau, à 2 dimensions par exemple, se fait comme suit :

```
type tab[M][N] = {{C0,0, ..., C0,N-1}, ..., {CM-1,0, ..., CM-1,N-1}};
```

- On déclare le tableau à 2 dimensions suivant :

```
int tab[2][3]; /* tableau unidimensionnel à 3 éléments */
```

Quelle est le type de `tab[1]` ? Est-ce un tableau unidimensionnel de 2 ou 3 éléments ? Il est malheureusement facile de se tromper, mais il y a un moyen mnémotechnique simple...

Tableaux multidimensionnels (1/3)

- La déclaration d'un tableau à k dimensions se fait ainsi :

```
type tab[N1]...[Nk];
```

et l'accès à un élément :

```
tab[i1]...[ik] /* 0 ≤ ij < Nj, j = 1, ..., k */
```

- La déclaration et l'initialisation d'un tableau, à 2 dimensions par exemple, se fait comme suit :

```
type tab[M][N] = {{C0,0, ..., C0,N-1}, ..., {CM-1,0, ..., CM-1,N-1}};
```

- On déclare le tableau à 2 dimensions suivant :

```
int          [3]; /* tableau unidimensionnel à 3 éléments */
```

Quelle est le type de `tab[1]` ? Est-ce un tableau unidimensionnel de 2 ou 3 éléments ? Il est malheureusement facile de se tromper, mais il y a un moyen mnémotechnique simple...

Tableaux multidimensionnels (2/3)

- Comme pour les tableaux unidimensionnels, l'objet `tab` est un pointeur (constant) vers `tab[0]` :

```
type tab[N1]...[Nk];
```

```
tab  $\Longleftrightarrow$  &tab[0];
```

Attention : ici `tab` est de type :

```
type (*)[N2]...[Nk];
```

- On peut également appliquer l'opérateur `sizeof` à un tableau multidimensionnel :

```
sizeof(tab)  $\implies$  N1  $\times$  ...  $\times$  Nk  $\times$  sizeof(type)
```

Le nombre d'éléments d'un tableau multidimensionnel sera donc comme précédemment : `sizeof(tab)/sizeof(type)`.

Tableaux multidimensionnels (2/3)

- Comme pour les tableaux unidimensionnels, l'objet `tab` est un pointeur (constant) vers `tab[0]` :

```
type tab[N1]...[Nk];
```

```
tab  $\Longleftrightarrow$  &tab[0];
```

Attention : ici `tab` est de type :

```
type (*)[N2]...[Nk];
```

- On peut également appliquer l'opérateur `sizeof` à un tableau multidimensionnel :

```
sizeof(tab)  $\implies$  N1  $\times$  ...  $\times$  Nk  $\times$  sizeof(type)
```

Le nombre d'éléments d'un tableau multidimensionnel sera donc comme précédemment : `sizeof(tab)/sizeof(type)`.

Tableaux multidimensionnels (3/3)

Parcours d'un tableau multidimensionnel :

- avec l'opérateur d'indexation [] :

```
float tab[M][N];

for (int i = 0; i < M; i++)
    for (int j = 0; j < N; j++)
        tab[i][j] = 1.0;
```

- avec des pointeurs :

```
float tab[M][N]; /* tab est de type : float (*)[N] */

for (float (*i)[N] = tab; i - tab < M; i++)
    for (float *j = *i; j - *i < N; j++)
        *j = 1.0;
```

Tableaux multidimensionnels (3/3)

Parcours d'un tableau multidimensionnel :

- avec l'opérateur d'indexation [] :

```
float tab[M][N];

for (int i = 0; i < M; i++)
    for (int j = 0; j < N; j++)
        tab[i][j] = 1.0;
```

- avec des pointeurs :

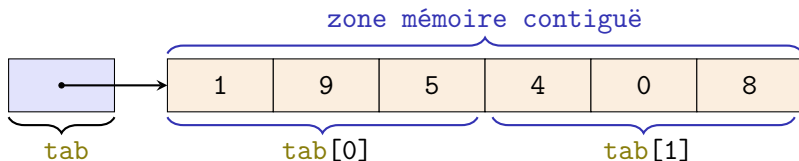
```
float tab[M][N]; /* tab est de type : float (*)[N] */

for (float (*i)[N] = tab; i - tab < M; i++)
    for (float *j = *i; j - *i < N; j++)
        *j = 1.0;
```

Tableaux multidimensionnels : schéma mémoire

```
/* schéma d'un tableau 2×3 d'entiers */
```

```
int tab[2][3] = {{1, 9, 5}, {4, 0, 8}};
```

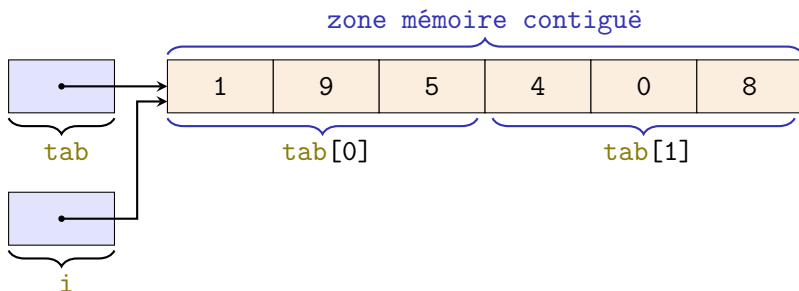


Tableaux multidimensionnels : schéma mémoire

```
/* schéma d'un tableau 2×3 d'entiers */
```

```
int tab[2][3] = {{1, 9, 5}, {4, 0, 8}};
```

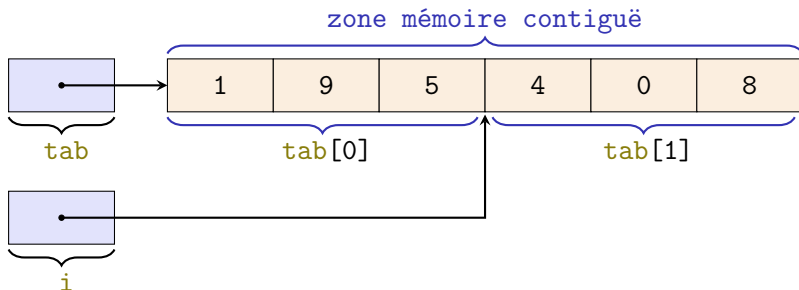
```
int (*i)[3] = tab; /* parenthèses nécessaires ! */
```



Tableaux multidimensionnels : schéma mémoire

```
/* schéma d'un tableau 2×3 d'entiers */
```

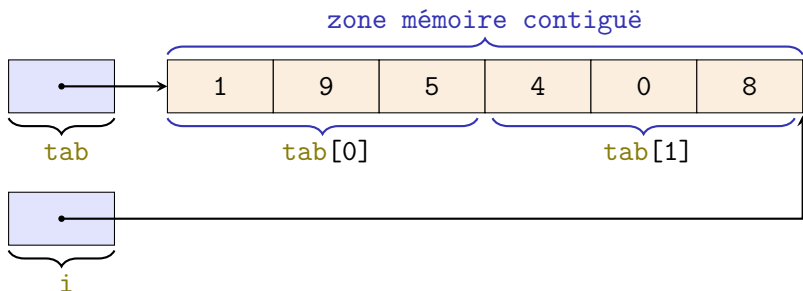
```
int tab[2][3] = {{1, 9, 5}, {4, 0, 8}};  
int (*i)[3] = tab; /* parenthèses nécessaires ! */  
i++;
```



Tableaux multidimensionnels : schéma mémoire

```
/* schéma d'un tableau 2×3 d'entiers */
```

```
int tab[2][3] = {{1, 9, 5}, {4, 0, 8}};  
int (*i)[3] = tab; /* parenthèses nécessaires ! */  
i++;  
i++;
```



Tableaux multidimensionnels & Pointeurs

Création de la matrice nulle $M \times N$ dont les éléments sont des réels double précision :

```
double tab[M][N]; /* statiquement */
```

```
for (int i = 0; i < M; i++)
    for (int j = 0; j < N; j++)
        tab[i][j] = 0.0;
```

```
double **tab;      /* dynamiquement */
```

```
tab = malloc(M * sizeof(double *));
for (int i = 0; i < M; i++)
    tab[i] = calloc(N, sizeof(double));
...
for (int i = 0; i < M; i++)
    free(tab[i]);
free(tab);
```

Tableaux multidimensionnels & Pointeurs

Création de la matrice nulle $M \times N$ dont les éléments sont des réels double précision :

```
double tab[M][N]; /* statiquement */

for (int i = 0; i < M; i++)
    for (int j = 0; j < N; j++)
        tab[i][j] = 0.0;
```

```
double **tab;      /* dynamiquement */

tab = malloc(M * sizeof(double *));
for (int i = 0; i < M; i++)
    tab[i] = calloc(N, sizeof(double));
...
for (int i = 0; i < M; i++)
    free(tab[i]);
free(tab);
```

Une douceur syntaxique

- En C, on a fréquemment à manipuler des **pointeurs vers des structures** et donc, via ces pointeurs, à accéder aux champs de ces structures.

- Par exemple, considérons la structure suivante :

```
struct point {double x; double y};
```

et les déclarations :

```
struct point p = {.x = 0.1, .y = 0.5}, *pp = &p;
```

- Si l'on veut accéder au champ `x` de la structure `p` via le pointeur `pp`, on applique l'opérateur d'indirection `*` puis l'opérateur `.`, ce qui donne : `*pp.x`. Mais l'opérateur `.` étant **prioritaire**, l'expression `*pp.x` est interprétée comme `*(pp.x)`, ce qui n'a pas de sens ici.
- Il aurait fallu écrire : `(*pp).x`, expression maintenant correcte. Ce type d'expression revient si souvent dans les programmes C, qu'une notation spéciale (opérateur `->`) a été introduite :

$$pp \rightarrow x \iff (*pp).x$$

Une douceur syntaxique

- En C, on a fréquemment à manipuler des **pointeurs vers des structures** et donc, via ces pointeurs, à accéder aux champs de ces structures.
- Par exemple, considérons la structure suivante :

```
struct point {double x; double y};
```

et les déclarations :

```
struct point p = {.x = 0.1, .y = 0.5}, *pp = &p;
```

- Si l'on veut accéder au champ `x` de la structure `p` via le pointeur `pp`, on applique l'opérateur d'indirection `*` puis l'opérateur `.`, ce qui donne : `*pp.x`. Mais l'opérateur `.` étant **prioritaire**, l'expression `*pp.x` est interprétée comme `*(pp.x)`, ce qui n'a pas de sens ici.
- Il aurait fallu écrire : `(*pp).x`, expression maintenant correcte. Ce type d'expression revient si souvent dans les programmes C, qu'une notation spéciale (opérateur `->`) a été introduite :

$$pp \rightarrow x \iff (*pp).x$$

Une douceur syntaxique

- En C, on a fréquemment à manipuler des **pointeurs vers des structures** et donc, via ces pointeurs, à accéder aux champs de ces structures.

- Par exemple, considérons la structure suivante :

```
struct point {double x; double y};
```

et les déclarations :

```
struct point p = {.x = 0.1, .y = 0.5}, *pp = &p;
```

- Si l'on veut accéder au champ **x** de la structure **p** via le pointeur **pp**, on applique l'opérateur d'indirection ***** puis l'opérateur **.**, ce qui donne : ***pp.x**. Mais l'opérateur **.** étant **prioritaire**, l'expression ***pp.x** est interprétée comme ***(pp.x)**, ce qui n'a pas de sens ici.
- Il aurait fallu écrire : **(*pp).x**, expression maintenant correcte. Ce type d'expression revient si souvent dans les programmes C, qu'une notation spéciale (opérateur **->**) a été introduite :

$$pp \rightarrow x \iff (*pp).x$$

Une douceur syntaxique

- En C, on a fréquemment à manipuler des **pointeurs vers des structures** et donc, via ces pointeurs, à accéder aux champs de ces structures.

- Par exemple, considérons la structure suivante :

```
struct point {double x; double y};
```

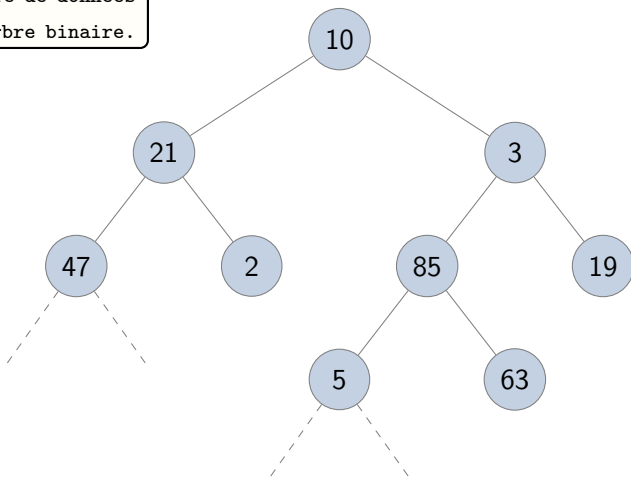
et les déclarations :

```
struct point p = {.x = 0.1, .y = 0.5}, *pp = &p;
```

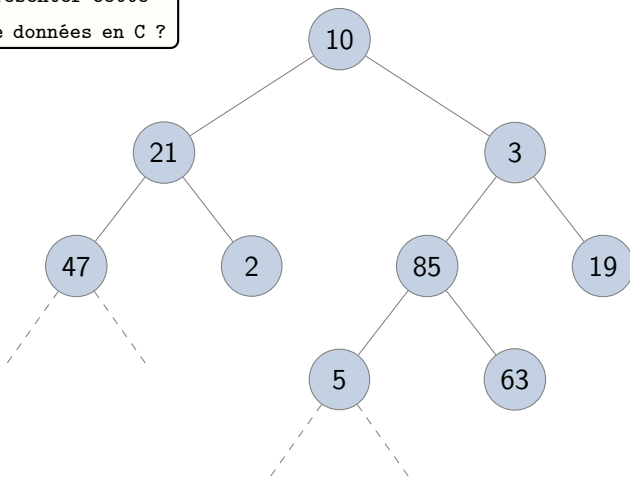
- Si l'on veut accéder au champ **x** de la structure **p** via le pointeur **pp**, on applique l'opérateur d'indirection ***** puis l'opérateur **.**, ce qui donne : ***pp.x**. Mais l'opérateur **.** étant **prioritaire**, l'expression ***pp.x** est interprétée comme ***(pp.x)**, ce qui n'a pas de sens ici.
- Il aurait fallu écrire : **(*pp).x**, expression maintenant correcte. Ce type d'expression revient si souvent dans les programmes C, qu'une notation spéciale (opérateur **->**) a été introduite :

$$pp \rightarrow x \iff (*pp).x$$

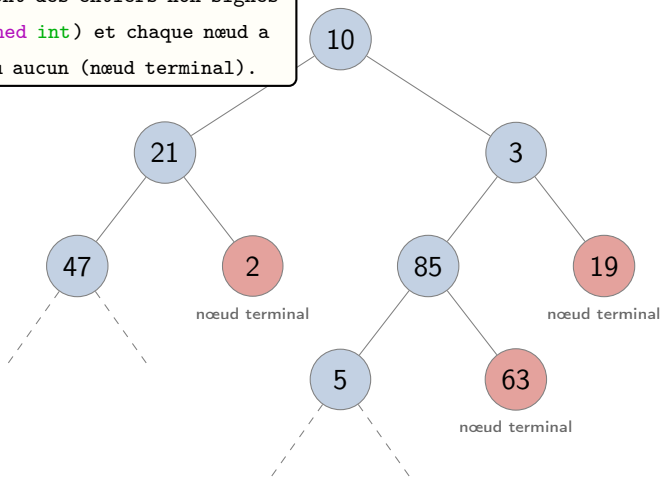
Une structure de données
utile : l'arbre binaire.



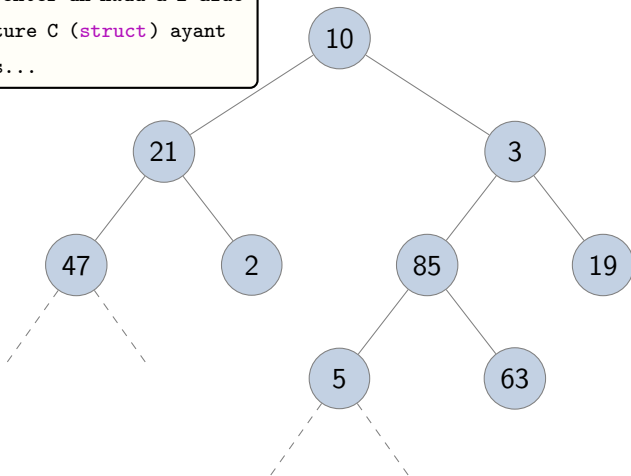
Comment représenter cette
structure de données en C ?



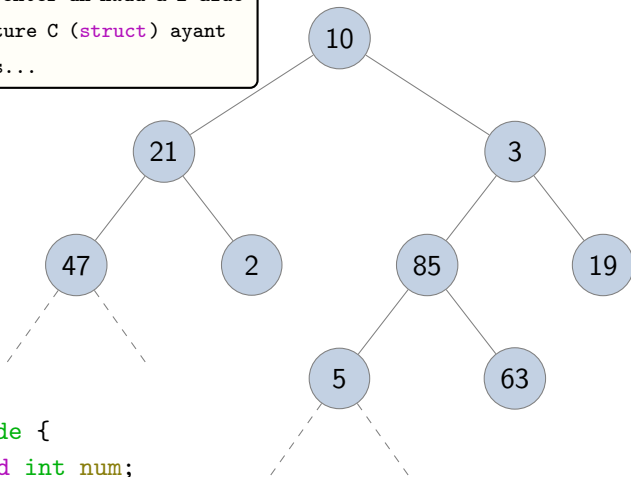
Les nœuds sont des entiers non signés (type `unsigned int`) et chaque nœud a deux fils ou aucun (nœud terminal).



On va représenter un nœud à l'aide d'une structure C (**struct**) ayant trois champs...

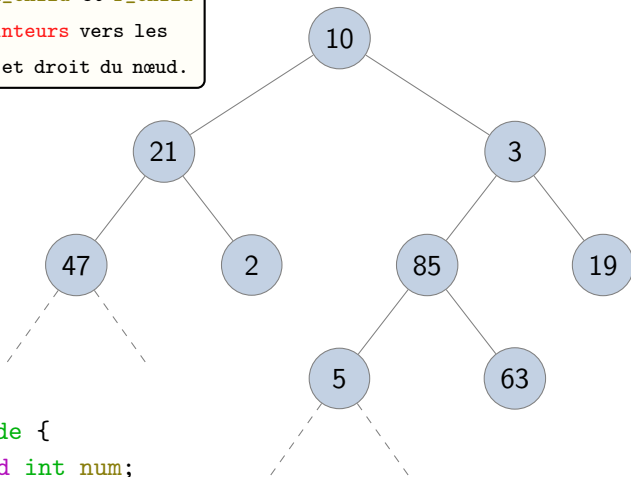


On va représenter un nœud à l'aide d'une structure C (**struct**) ayant trois champs...



```
struct node {  
    unsigned int num;  
    struct node *l_child; /* NULL si nœud terminal */  
    struct node *r_child; /* NULL si nœud terminal */  
};
```

Les champs `l_child` et `r_child` sont des **pointeurs** vers les fils gauche et droit du nœud.



```
struct node {  
    unsigned int num;  
    struct node *l_child; /* NULL si nœud terminal */  
    struct node *r_child; /* NULL si nœud terminal */  
};
```

Un point important

- La définition du type de données `struct node` est bien une définition **pseudo-réursive**.

```
struct node {  
    unsigned int num;  
    struct node *l_child;  
    struct node *r_child; };
```

En effet les champs sont des **pointeurs** sur la structure que l'on est en train de définir et non pas des structures `node`.

- En revanche, la définition suivante est **incorrecte** :

```
struct node {  
    unsigned int num;  
    struct node l_child;  
    struct node r_child; };
```

Si on compile avec GNU CC, on a l'erreur suivante :

```
$ gcc -Wall struct.c -o struct  
error: field 'l_child' has incomplete type
```

Un point important

- La définition du type de données `struct node` est bien une définition **pseudo-réursive**.

```
struct node {  
    unsigned int num;  
    struct node *l_child;  
    struct node *r_child; };
```

En effet les champs sont des **pointeurs** sur la structure que l'on est en train de définir et non pas des structures `node`.

- En revanche, la définition suivante est **incorrecte** :

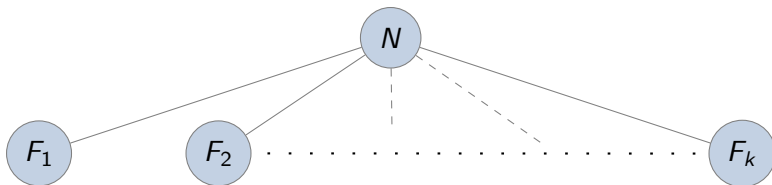
```
struct node {  
    unsigned int num;  
    struct node l_child;  
    struct node r_child; };
```

Si on compile avec GNU CC, on a l'erreur suivante :

```
$ gcc -Wall struct.c -o struct  
error: field 'l_child' has incomplete type
```

Un autre exemple : arbres quelconques

- Nous allons maintenant considérer des arbres quelconques, à savoir des arbres dont les nœuds peuvent avoir un **nombre quelconque de fils**.



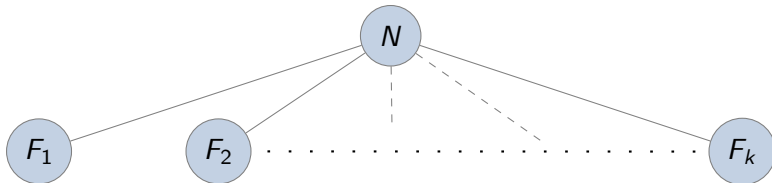
- Cela suggère la définition qui suit :

```
typedef struct __node { /* nécessairement non anonyme */
    unsigned int    num;      /* numéro du nœud */
    struct __node **childs;   /* pointeurs vers les fils */
    size_t          nchilds; /* nombre de fils */
} tree_t;
```

- A titre d'illustration, nous allons coder la fonction `add_leaf_child` qui insère un nouveau nœud terminal dans un arbre.

Un autre exemple : arbres quelconques

- Nous allons maintenant considérer des arbres quelconques, à savoir des arbres dont les nœuds peuvent avoir un **nombre quelconque de fils**.



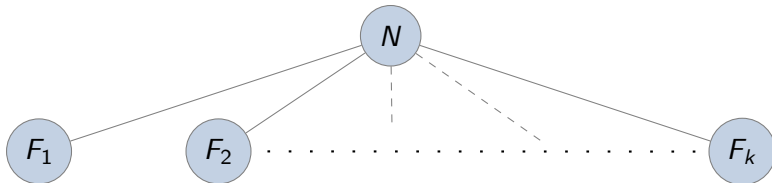
- Cela suggère la définition qui suit :

```
typedef struct __node { /* nécessairement non anonyme */
    unsigned int    num;      /* numéro du nœud */
    struct __node **childs;   /* pointeurs vers les fils */
    size_t          nchilds; /* nombre de fils */
} tree_t;
```

- A titre d'illustration, nous allons coder la fonction `add_leaf_child` qui insère un nouveau nœud terminal dans un arbre.

Un autre exemple : arbres quelconques

- Nous allons maintenant considérer des arbres quelconques, à savoir des arbres dont les nœuds peuvent avoir un **nombre quelconque de fils**.



- Cela suggère la définition qui suit :

```
typedef struct __node { /* nécessairement non anonyme */
    unsigned int    num;      /* numéro du nœud */
    struct __node **childs;   /* pointeurs vers les fils */
    size_t          nchilds; /* nombre de fils */
} tree_t;
```

- A titre d'illustration, nous allons coder la fonction `add_leaf_child` qui insère un nouveau nœud terminal dans un arbre.


```
int add_leaf_child(tree_t *t, unsigned int num) {
```

```
int add_leaf_child(tree_t *t, unsigned int num) {  
    tree_t **ptr, *new;  
    /* création du nouveau nœud terminal */  
    new = malloc(sizeof(tree_t));  
    if (!new) return -1;  
    new->num = num;  
    new->childs = NULL;  
    new->nchilds = 0;
```

```
int add_leaf_child(tree_t *t, unsigned int num) {
    tree_t **ptr, *new;
    /* création du nouveau nœud terminal */
    new = malloc(sizeof(tree_t));
    if (!new) return -1;
    new->num = num;
    new->childs = NULL;
    new->nchilds = 0;
    /* insertion du nouveau nœud terminal */
    ptr = realloc(t->childs,
                  (t->nchilds + 1) * sizeof(tree_t *));
    if (!ptr) { /* t->childs est inchangé dans ce cas */
        free(new);
        return -1;
    }
    t->childs = ptr;
    t->nchilds++;
    for (ptr = t->childs; ptr - t->childs < t->nchilds - 1;
         ptr++); /* on se positionne sur le dernier fils */
    *ptr = new; /* *ptr pointe sur le dernier et nouveau fils */
    return 0;
}
```

Valeurs de vérité

- Il n'y a pas en C, contrairement à d'autres langages, de **type booléen**.
- Toute expression en C ayant une valeur, une expression est considérée comme **vraie** si sa valeur est $\neq 0$, **fausse** sinon :

```
int i, j;  
.....  
if ((i = j)) {  
    ..... /* code exécuté si j  $\neq$  0 */  
}
```

Ici la valeur de l'affectation ($i = j$) est la valeur affectée, à savoir la valeur de la variable j .

- Si l'on omet les doubles parenthèses dans le test `if ((i = j))`, le compilateur `gcc` avec l'option `-Wall` lève l'avertissement :

```
warning: suggest parentheses around assignment used as  
truth value [-Wparentheses]
```

Valeurs de vérité

- Il n'y a pas en C, contrairement à d'autres langages, de **type booléen**.
- Toute expression en C ayant une valeur, une expression est considérée comme **vraie** si sa valeur est $\neq 0$, **fausse** sinon :

```
int i, j;  
.....  
if ((i = j)) {  
    ..... /* code exécuté si j  $\neq$  0 */  
}
```

Ici la valeur de l'affectation ($i = j$) est la valeur affectée, à savoir la valeur de la variable j .

- Si l'on omet les doubles parenthèses dans le test `if ((i = j))`, le compilateur `gcc` avec l'option `-Wall` lève l'avertissement :

```
warning: suggest parentheses around assignment used as  
truth value [-Wparentheses]
```

Valeurs de vérité

- Il n'y a pas en C, contrairement à d'autres langages, de **type booléen**.
- Toute expression en C ayant une valeur, une expression est considérée comme **vraie** si sa valeur est $\neq 0$, **fausse** sinon :

```
int i, j;  
.....  
if ((i = j)) {  
    ..... /* code exécuté si j  $\neq$  0 */  
}
```

Ici la valeur de l'affectation ($i = j$) est la valeur affectée, à savoir la valeur de la variable j .

- Si l'on omet les doubles parenthèses dans le test `if ((i = j))`, le compilateur `gcc` avec l'option `-Wall` lève l'avertissement :

```
warning: suggest parentheses around assignment used as  
truth value [-Wparentheses]
```

Non commutativité des opérateurs logiques

- En logique mathématique, $A \wedge B \iff B \wedge A$. Ceci est **faux** en C :
 $A \ \&\& \ B \not\iff B \ \&\& \ A$.
- En effet, l'expression $A \ \&\& \ B$ est évaluée de la gauche vers la droite. Donc A est d'abord évaluée. Si la valeur de A est 0 (faux), la conjonction $A \ \&\& \ B$ est donc fausse et l'évaluation s'arrête. Par conséquent B **n'est pas évaluée**. Dans le cas contraire, B est évaluée.
- En particulier, si B produit un effet de bord, on voit bien qu'en général : $A \ \&\& \ B \not\iff B \ \&\& \ A$. Par exemple :

```
int i = 2, j = 2;
```

```
(i == j && (i = 1)) /* vrai */
```

alors que :

```
((i = 1) && i == j) /* faux */
```

Non commutativité des opérateurs logiques

- En logique mathématique, $A \wedge B \iff B \wedge A$. Ceci est **faux** en C :
 $A \ \&\& \ B \not\iff B \ \&\& \ A$.
- En effet, l'expression $A \ \&\& \ B$ est évaluée de la gauche vers la droite. Donc A est d'abord évaluée. Si la valeur de A est 0 (faux), la conjonction $A \ \&\& \ B$ est donc fausse et l'évaluation s'arrête. Par conséquent B **n'est pas évaluée**. Dans le cas contraire, B est évaluée.
- En particulier, si B produit un effet de bord, on voit bien qu'en général : $A \ \&\& \ B \not\iff B \ \&\& \ A$. Par exemple :

```
int i = 2, j = 2;
```

```
(i == j && (i = 1)) /* vrai */
```

alors que :

```
((i = 1) && i == j) /* faux */
```


Non commutativité des opérateurs logiques

- En logique mathématique, $A \wedge B \iff B \wedge A$. Ceci est **faux** en C :
 $A \ \&\& \ B \not\iff B \ \&\& \ A$.
- En effet, l'expression $A \ \&\& \ B$ est évaluée de la gauche vers la droite. Donc A est d'abord évaluée. Si la valeur de A est 0 (faux), la conjonction $A \ \&\& \ B$ est donc fausse et l'évaluation s'arrête. Par conséquent B **n'est pas évaluée**. Dans le cas contraire, B est évaluée.
- En particulier, si B produit un effet de bord, on voit bien qu'en général : $A \ \&\& \ B \not\iff B \ \&\& \ A$. Par exemple :

```
int i = 2, j = 2;
```

```
(i == j && (i = 1)) /* vrai */
```

alors que :

```
((i = 1) && i == j) /* faux */
```

Caractéristiques & Déclaration

- Une variable de la classe `static` est une variable **permanente**.
- Une telle variable occupe en mémoire une **zone fixe** qui reste la même durant toute l'exécution du programme.
- Cette zone est allouée au moment de la **compilation** et est **initialisée** à 0 par défaut.
- On distingue deux types de variables de la classe `static` : les variables **globales** (variables déclarées en dehors de toute fonction) et les variables **locales** (variables déclarées au sein d'une fonction ou d'un bloc).
- Pour déclarer une variable globale :

```
static int i; /* initialisée à 0 par défaut */
```

et une variable locale (à une fonction) :

```
int f(void) {  
    static int i = 1;  
    .....  
}
```

Caractéristiques & Déclaration

- Une variable de la classe `static` est une variable **permanente**.
- Une telle variable occupe en mémoire une **zone fixe** qui reste la même durant toute l'exécution du programme.
- Cette zone est allouée au moment de la **compilation** et est **initialisée** à 0 par défaut.
- On distingue deux types de variables de la classe `static` : les variables **globales** (variables déclarées en dehors de toute fonction) et les variables **locales** (variables déclarées au sein d'une fonction ou d'un bloc).
- Pour déclarer une variable globale :

```
static int i; /* initialisée à 0 par défaut */
```

et une variable locale (à une fonction) :

```
int f(void) {  
    static int i = 1;  
    .....  
}
```

Caractéristiques & Déclaration

- Une variable de la classe `static` est une variable **permanente**.
- Une telle variable occupe en mémoire une **zone fixe** qui reste la même durant toute l'exécution du programme.
- Cette zone est allouée au moment de la **compilation** et est **initialisée à 0** par défaut.
- On distingue deux types de variables de la classe `static` : les variables **globales** (variables déclarées en dehors de toute fonction) et les variables **locales** (variables déclarées au sein d'une fonction ou d'un bloc).
- Pour déclarer une variable globale :

```
static int i; /* initialisée à 0 par défaut */
```

et une variable locale (à une fonction) :

```
int f(void) {  
    static int i = 1;  
    .....  
}
```

Caractéristiques & Déclaration

- Une variable de la classe `static` est une variable **permanente**.
- Une telle variable occupe en mémoire une **zone fixe** qui reste la même durant toute l'exécution du programme.
- Cette zone est allouée au moment de la **compilation** et est **initialisée** à 0 par défaut.
- On distingue deux types de variables de la classe `static` : les variables **globales** (variables déclarées en dehors de toute fonction) et les variables **locales** (variables déclarées au sein d'une fonction ou d'un bloc).
- Pour déclarer une variable globale :

```
static int i; /* initialisée à 0 par défaut */
```

et une variable locale (à une fonction) :

```
int f(void) {  
    static int i = 1;  
    .....  
}
```

Caractéristiques & Déclaration

- Une variable de la classe `static` est une variable **permanente**.
- Une telle variable occupe en mémoire une **zone fixe** qui reste la même durant toute l'exécution du programme.
- Cette zone est allouée au moment de la **compilation** et est **initialisée** à 0 par défaut.
- On distingue deux types de variables de la classe `static` : les variables **globales** (variables déclarées en dehors de toute fonction) et les variables **locales** (variables déclarées au sein d'une fonction ou d'un bloc).
- Pour déclarer une variable globale :

```
static int i; /* initialisée à 0 par défaut */
```

et une variable locale (à une fonction) :

```
int f(void) {  
    static int i = 1;  
    .....  
}
```

Caractéristiques & Déclaration

- Une variable de la classe `static` est une variable **permanente**.
- Une telle variable occupe en mémoire une **zone fixe** qui reste la même durant toute l'exécution du programme.
- Cette zone est allouée au moment de la **compilation** et est **initialisée** à 0 par défaut.
- On distingue deux types de variables de la classe `static` : les variables **globales** (variables déclarées en dehors de toute fonction) et les variables **locales** (variables déclarées au sein d'une fonction ou d'un bloc).
- Pour déclarer une variable globale :

```
static int i; /* initialisée à 0 par défaut */
```

et une variable locale (à une fonction) :

```
int f(void) {  
    static int i = 1;  
    .....  
}
```

Variables globales

- De manière analogue aux fonctions, une variable globale de la classe `static` n'est connue que dans le module où elle a été déclarée.
- Un exemple pratique, une fonction d'impression de messages d'erreur :

```
static char *err_msg[] = {
#define ENOERR 0
    "no_error",
#define ENOMEM 1
    "out_of_memory",
    .....
#define EMAX 64
    "unexpected_error"
}; /* variable locale à l'implémentation */

void print_error(int ecode, FILE *os) {
    if (ecode < ENOERR || ecode >= EMAX)
        fprintf(os, "%s", err_msg[EMAX]);
    else
        fprintf(os, "%s", err_msg[ecode]);
} /* fonction (publique) de la bibliothèque */
```


Variables globales

- De manière analogue aux fonctions, une variable globale de la classe `static` n'est connue que dans le module où elle a été déclarée.
- Un exemple pratique, une fonction d'impression de messages d'erreur :

```
static char *err_msg[] = {
#define ENOERR 0
    "no_error",
#define ENOMEM 1
    "out_of_memory",
    .....
#define EMAX 64
    "unexpected_error"
}; /* variable locale à l'implémentation */

void print_error(int ecode, FILE *os) {
    if (ecode < ENOERR || ecode >= EMAX)
        fprintf(os, "%s", err_msg[EMAX]);
    else
        fprintf(os, "%s", err_msg[ecode]);
} /* fonction (publique) de la bibliothèque */
```

Variables locales

- On a vu que qu'une variable de la classe `static` était allouée et **initialisée** au moment de la compilation.
- Ainsi, dans le cas d'une variable locale à une fonction, si cette variable est modifiée par la fonction, la modification sera **persistante**.
- Un exemple pour bien comprendre cette « persistance » :

```
void f(void) {  
    static int flag = 1;  
  
    if (flag) {  
        .....  
        flag = 0;  
    }  
    .....  
}
```

Cela peut être utile si, par exemple, la fonction `f` doit initialiser une bibliothèque (donc une seule fois).

Variables locales

- On a vu que qu'une variable de la classe `static` était allouée et **initialisée** au moment de la compilation.
- Ainsi, dans le cas d'une variable locale à une fonction, si cette variable est modifiée par la fonction, la modification sera **persistante**.
- Un exemple pour bien comprendre cette « persistance » :

```
void f(void) {  
    static int flag = 1;  
  
    if (flag) {  
        .....  
        flag = 0;  
    }  
    .....  
}
```

Cela peut être utile si, par exemple, la fonction `f` doit initialiser une bibliothèque (donc une seule fois).

Variables locales

- On a vu que qu'une variable de la classe `static` était allouée et **initialisée** au moment de la compilation.
- Ainsi, dans le cas d'une variable locale à une fonction, si cette variable est modifiée par la fonction, la modification sera **persistante**.
- Un exemple pour bien comprendre cette « persistance » :

```
void f(void) {  
    static int flag = 1;  
  
    if (flag) {  
        .....  
        flag = 0;  
    }  
    .....  
}
```

Cela peut être utile si, par exemple, la fonction `f` doit initialiser une bibliothèque (donc une seule fois).

Variables locales

- On a vu que qu'une variable de la classe `static` était allouée et **initialisée** au moment de la compilation.
- Ainsi, dans le cas d'une variable locale à une fonction, si cette variable est modifiée par la fonction, la modification sera **persistante**.
- Un exemple pour bien comprendre cette « persistance » :

```
void f(void) {  
    static int flag = 1;  
  
    if (flag) {  
        .....  
        flag = 0;  
    }  
    .....  
}
```

Ce bloc de code ne sera exécuté qu'au **premier appel** de la fonction `f`.

Cela peut être utile si, par exemple, la fonction `f` doit initialiser une bibliothèque (donc une seule fois).

Variables locales

- On a vu que qu'une variable de la classe `static` était allouée et **initialisée** au moment de la compilation.
- Ainsi, dans le cas d'une variable locale à une fonction, si cette variable est modifiée par la fonction, la modification sera **persistante**.
- Un exemple pour bien comprendre cette « persistance » :

```
void f(void) {  
    static int flag = 1;  
  
    if (flag) {  
        lib_init(); /* initialisation */  
        flag = 0;  
    }  
    .....  
}
```

Cela peut être utile si, par exemple, la fonction `f` doit initialiser une bibliothèque (donc une seule fois).

Motivation et type d'un pointeur de fonction

- Il n'y a pas de **type fonction** en C contrairement à d'autres langages.
- Or on a parfois besoin de passer en **paramètres à une fonction** une ou plusieurs fonctions. Nous aurons l'occasion de le voir lorsque nous aborderons la bibliothèque de gestion de listes.
- Un pointeur vers une fonction de prototype :

```
type fonction(type_1, ..., type_N);
```

a pour type :

```
type (*)(type_1, ..., type_N);
```

- Par exemple, pour déclarer un pointeur vers la fonction de prototype `void *copy_data(const void *)` :

```
void *(*ptr)(const void *);
```

Les parenthèses autour de `*ptr` sont indispensables. Sinon la déclaration devient (`ptr` n'est alors plus un pointeur de fonction) :

```
void **ptr(const void *);
```

Motivation et type d'un pointeur de fonction

- Il n'y a pas de **type fonction** en C contrairement à d'autres langages.
- Or on a parfois besoin de passer en **paramètres à une fonction** une ou plusieurs fonctions. Nous aurons l'occasion de le voir lorsque nous aborderons la bibliothèque de gestion de listes.

- Un pointeur vers une fonction de prototype :

```
type fonction(type_1, ..., type_N);
```

a pour type :

```
type (*)(type_1, ..., type_N);
```

- Par exemple, pour déclarer un pointeur vers la fonction de prototype

```
void *copy_data(const void *); :
```

```
void *(*ptr)(const void *);
```

Les parenthèses autour de *ptr sont indispensables. Sinon la déclaration devient (ptr n'est alors plus un pointeur de fonction) :

```
void **ptr(const void *);
```


Motivation et type d'un pointeur de fonction

- Il n'y a pas de **type fonction** en C contrairement à d'autres langages.
- Or on a parfois besoin de passer en **paramètres à une fonction** une ou plusieurs fonctions. Nous aurons l'occasion de le voir lorsque nous aborderons la bibliothèque de gestion de listes.
- Un pointeur vers une fonction de prototype :

```
type fonction(type_1, ..., type_N);
```

a pour type :

```
type (*) (type_1, ..., type_N);
```

- Par exemple, pour déclarer un pointeur vers la fonction de prototype
`void *copy_data(const void *) ;`

```
void *(*ptr)(const void *) ;
```

Les parenthèses autour de `*ptr` sont indispensables. Sinon la déclaration devient (`ptr` n'est alors plus un pointeur de fonction) :

```
void **ptr(const void *) ;
```

Motivation et type d'un pointeur de fonction

- Il n'y a pas de **type fonction** en C contrairement à d'autres langages.
- Or on a parfois besoin de passer en **paramètres à une fonction** une ou plusieurs fonctions. Nous aurons l'occasion de le voir lorsque nous aborderons la bibliothèque de gestion de listes.
- Un pointeur vers une fonction de prototype :

```
type fonction(type_1, ..., type_N);
```

a pour type :

```
type (*) (type_1, ..., type_N);
```

- Par exemple, pour déclarer un pointeur vers la fonction de prototype `void *copy_data(const void *)` :

```
void *(*ptr)(const void *);
```

Les parenthèses autour de `*ptr` sont indispensables. Sinon la déclaration devient (`ptr` n'est alors plus un pointeur de fonction) :

```
void **ptr(const void *);
```

Quelques remarques (1/2)

- Si `fct` est une fonction, l'objet `fct` est en fait un pointeur (constant) vers le premier octet du code de la fonction.
- Plus précisément, si `fct` a, par exemple, pour prototype :

```
int fct(int *);
```

la déclaration/initialisation suivante :

```
int (*ptr)(int *) = fct;
```

a pour effet de copier l'adresse de ce premier octet dans le pointeur de fonction `ptr`.

- A priori, l'expression `sizeof(fct)` n'a guère de sens et par voie de conséquence l'instruction `ptr++`; non plus. Cependant, si on compile avec GNU CC, on aura ni erreur ni avertissement sauf avec l'option `-Wpointer-arith` (non comprise dans `-Wall`). Cela étant,

```
int i = 2; ptr++; ptr(&i); /* on a avancé d'un octet */
```

déclenchera inmanquablement une erreur fatale (*segmentation fault*).

Quelques remarques (1/2)

- Si `fct` est une fonction, l'objet `fct` est en fait un pointeur (constant) vers le premier octet du code de la fonction.
- Plus précisément, si `fct` a, par exemple, pour prototype :

```
int fct(int *);
```

la déclaration/initialisation suivante :

```
int (*ptr)(int *) = fct;
```

a pour effet de copier l'adresse de ce premier octet dans le pointeur de fonction `ptr`.

- A priori, l'expression `sizeof(fct)` n'a guère de sens et par voie de conséquence l'instruction `ptr++`; non plus. Cependant, si on compile avec GNU CC, on aura ni erreur ni avertissement sauf avec l'option `-Wpointer-arith` (non comprise dans `-Wall`). Cela étant,

```
int i = 2; ptr++; ptr(&i); /* on a avancé d'un octet */
```

déclenchera inmanquablement une erreur fatale (*segmentation fault*).

Quelques remarques (1/2)

- Si `fct` est une fonction, l'objet `fct` est en fait un pointeur (constant) vers le premier octet du code de la fonction.
- Plus précisément, si `fct` a, par exemple, pour prototype :

```
int fct(int *);
```

la déclaration/initialisation suivante :

```
int (*ptr)(int *) = fct;
```

a pour effet de copier l'adresse de ce premier octet dans le pointeur de fonction `ptr`.

- A priori, l'expression `sizeof(fct)` n'a guère de sens et par voie de conséquence l'instruction `ptr++`; non plus. Cependant, si on compile avec GNU CC, on aura ni erreur ni avertissement sauf avec l'option `-Wpointer-arith` (non comprise dans `-Wall`). Cela étant,

```
int i = 2; ptr++; ptr(&i); /* on a avancé d'un octet */
```

déclenchera inmanquablement une erreur fatale (*segmentation fault*).

Quelques remarques (2/2)

- Au niveau syntaxique, les pointeurs de fonctions rendent souvent le code **peu lisible**.
- Si vous n'en êtes pas convaincu, quel est le prototype d'une fonction `f` prenant en argument un pointeur vers une fonction `g` de prototype `int g(void)` et retournant un pointeur de même type ?
- La réponse : `int (*f(int (*)(void)))(void);` ☹️☹️☹️
- Une parade efficace : utiliser `typedef`. Dans notre cas :

```
typedef int (*g_t)(void);  
  
g_t f(g_t);
```

Cette écriture est incontestablement plus lisible. D'une manière générale, et donc pas seulement avec les pointeurs de fonctions, il est recommandé d'introduire des **nouveaux types** avec `typedef`. Cela rend le code plus clair et plus **aisément modifiable**.

Quelques remarques (2/2)

- Au niveau syntaxique, les pointeurs de fonctions rendent souvent le code **peu lisible**.
- Si vous n'en êtes pas convaincu, quel est le prototype d'une fonction `f` prenant en argument un pointeur vers une fonction `g` de prototype `int g(void)` et retournant un pointeur de même type ?
- La réponse : `int (*f(int (*)(void)))(void);` ☹️☹️☹️
- Une parade efficace : utiliser `typedef`. Dans notre cas :

```
typedef int (*g_t)(void);

g_t f(g_t);
```

Cette écriture est incontestablement plus lisible. D'une manière générale, et donc pas seulement avec les pointeurs de fonctions, il est recommandé d'introduire des **nouveaux types** avec `typedef`. Cela rend le code plus clair et plus **aisément modifiable**.

Quelques remarques (2/2)

- Au niveau syntaxique, les pointeurs de fonctions rendent souvent le code **peu lisible**.
- Si vous n'en êtes pas convaincu, quel est le prototype d'une fonction `f` prenant en argument un pointeur vers une fonction `g` de prototype `int g(void)` et retournant un pointeur de même type ?
- La réponse : `int (*f(int (*)(void)))(void);` ☹☹☹
- Une parade efficace : utiliser `typedef`. Dans notre cas :

```
typedef int (*g_t)(void);  
  
g_t f(g_t);
```

Cette écriture est incontestablement plus lisible. D'une manière générale, et donc pas seulement avec les pointeurs de fonctions, il est recommandé d'introduire des **nouveaux types** avec `typedef`. Cela rend le code plus clair et plus **aisément modifiable**.

Quelques remarques (2/2)

- Au niveau syntaxique, les pointeurs de fonctions rendent souvent le code **peu lisible**.
- Si vous n'en êtes pas convaincu, quel est le prototype d'une fonction `f` prenant en argument un pointeur vers une fonction `g` de prototype `int g(void)` et retournant un pointeur de même type ?
- La réponse : `int (*f(int (*)(void)))(void);` ☹☹☹
- Une parade efficace : utiliser `typedef`. Dans notre cas :

```
typedef int (*g_t)(void);
```

```
g_t f(g_t);
```

Cette écriture est incontestablement plus lisible. D'une manière générale, et donc pas seulement avec les pointeurs de fonctions, il est recommandé d'introduire des **nouveaux types** avec `typedef`. Cela rend le code plus clair et plus **aisément modifiable**.

Une ébauche de bibliothèque

- À titre d'illustration d'utilisation des pointeurs de fonction, nous allons commencer l'écriture d'une bibliothèque de **fonctions de hachage cryptographique**.
- Nous allons nous appuyer sur le logiciel *OpenSSL* et plus précisément sur la bibliothèque `libcrypto` qui implémente les **algorithmes cryptographiques** (*OpenSSL* est essentiellement constitué de deux bibliothèques : `libcrypto` et `libssl` qui implémente le protocole *TLS*).
- On va bien entendu commencer par écrire l'interface `hash.h` et en premier lieu définir la structure de données `hash_t` décrivant une empreinte.
- On aura souci que n'apparaisse pas dans l'interface le fait que l'on utilise la bibliothèque `libcrypto`, car le choix de cette bibliothèque ne concerne que l'implémentation.

Une ébauche de bibliothèque

- À titre d'illustration d'utilisation des pointeurs de fonction, nous allons commencer l'écriture d'une bibliothèque de **fonctions de hachage cryptographique**.
- Nous allons nous appuyer sur le logiciel *OpenSSL* et plus précisément sur la bibliothèque `libcrypto` qui implémente les **algorithmes cryptographiques** (*OpenSSL* est essentiellement constitué de deux bibliothèques : `libcrypto` et `libssl` qui implémente le protocole *TLS*).
- On va bien entendu commencer par écrire l'interface `hash.h` et en premier lieu définir la structure de données `hash_t` décrivant une empreinte.
- On aura souci que n'apparaisse pas dans l'interface le fait que l'on utilise la bibliothèque `libcrypto`, car le choix de cette bibliothèque ne concerne que l'implémentation.

Une ébauche de bibliothèque

- À titre d'illustration d'utilisation des pointeurs de fonction, nous allons commencer l'écriture d'une bibliothèque de **fonctions de hachage cryptographique**.
- Nous allons nous appuyer sur le logiciel *OpenSSL* et plus précisément sur la bibliothèque `libcrypto` qui implémente les **algorithmes cryptographiques** (*OpenSSL* est essentiellement constitué de deux bibliothèques : `libcrypto` et `libssl` qui implémente le protocole *TLS*).
- On va bien entendu commencer par écrire l'interface `hash.h` et en premier lieu définir la structure de données `hash_t` décrivant une **empreinte**.
- On aura souci que n'apparaisse pas dans l'interface le fait que l'on utilise la bibliothèque `libcrypto`, car le choix de cette bibliothèque ne concerne que l'implémentation.

Une ébauche de bibliothèque

- À titre d'illustration d'utilisation des pointeurs de fonction, nous allons commencer l'écriture d'une bibliothèque de **fonctions de hachage cryptographique**.
- Nous allons nous appuyer sur le logiciel *OpenSSL* et plus précisément sur la bibliothèque `libcrypto` qui implémente les **algorithmes cryptographiques** (*OpenSSL* est essentiellement constitué de deux bibliothèques : `libcrypto` et `libssl` qui implémente le protocole *TLS*).
- On va bien entendu commencer par écrire l'interface `hash.h` et en premier lieu définir la structure de données `hash_t` décrivant une **empreinte**.
- On aura souci que n'apparaisse pas dans l'interface le fait que l'on utilise la bibliothèque `libcrypto`, car le choix de cette bibliothèque **ne concerne que l'implémentation**.

L'interface : la structure de données `hash_t`

fichier en-tête : `hash.h`

```
#ifndef HASH_H  
#define HASH_H
```

```
#endif /* HASH_H */
```

L'interface : la structure de données `hash_t`

fichier en-tête : `hash.h`

```
#ifndef HASH_H
#define HASH_H

#include <sys/types.h> /* pour le type 'size_t' */
#include <inttypes.h>  /* pour le type 'uint8_t' */

#endif /* HASH_H */
```

L'interface : la structure de données `hash_t`

fichier en-tête : `hash.h`

```
#ifndef HASH_H
#define HASH_H

#include <sys/types.h>
#include <inttypes.h>

typedef struct {

} hash_t;

#endif /* HASH_H */
```

L'interface : la structure de données `hash_t`

fichier en-tête : `hash.h`

```
#ifndef HASH_H
#define HASH_H

#include <sys/types.h>
#include <inttypes.h>

typedef struct {

    uint8_t *md;        /* empreinte */
    size_t   md_len;    /* longueur de l'empreinte */
} hash_t;

#endif /* HASH_H */
```

L'interface : la structure de données `hash_t`

fichier en-tête : `hash.h`

```
#ifndef HASH_H
#define HASH_H

#include <sys/types.h>
#include <inttypes.h>

typedef struct {
    void      *id;          /* données internes (opaques) */
    uint8_t   *md;          /* empreinte */
    size_t     md_len;      /* longueur de l'empreinte */
} hash_t;

#endif /* HASH_H */
```

Le champ `id` est un pointeur vers des données opaques qui seront, dans **cette implémentation**, utilisées par la bibliothèque `libcrypto`. Le choix du type `void *` permet de masquer les détails de l'implémentation.

L'interface : les fonctions de hachage

Notre bibliothèque va offrir les fonctions de hachage les plus courantes :

fichier en-tête : **hash.h**

```
typedef enum {
```

```
} hash_md_t;
```

L'interface : les fonctions de hachage

Notre bibliothèque va offrir les fonctions de hachage les plus courantes :

fichier en-tête : hash.h

```
typedef enum {  
    HASH_MD5 ,                /* empreinte MD5 */  
    HASH_SHA1 ,               /* empreinte SHA1 */  
  
} hash_md_t;  
  
#define HASH_SHA HASH_SHA1    /* alias pour SHA1 */
```

L'interface : les fonctions de hachage

Notre bibliothèque va offrir les fonctions de hachage les plus courantes :

fichier en-tête : hash.h

```
typedef enum {  
    HASH_MD5 ,                /* empreinte MD5 */  
    HASH_SHA1 ,              /* empreinte SHA1 */  
#ifdef HAVE_HASH_SHA256  
    HASH_SHA224 ,            /* empreinte SHA224 */  
    HASH_SHA256 ,            /* empreinte SHA256 */  
#endif /* HAVE_HASH_SHA256 */  
  
} hash_md_t;  
  
#define HASH_SHA HASH_SHA1    /* alias pour SHA1 */
```

L'interface : les fonctions de hachage

Notre bibliothèque va offrir les fonctions de hachage les plus courantes :

fichier en-tête : hash.h

```
typedef enum {  
    HASH_MD5,                /* empreinte MD5 */  
    HASH_SHA1,               /* empreinte SHA1 */  
#ifdef HAVE_HASH_SHA256  
    HASH_SHA224,             /* empreinte SHA224 */  
    HASH_SHA256,             /* empreinte SHA256 */  
#endif /* HAVE_HASH_SHA256 */  
#ifdef HAVE_HASH_SHA512  
    HASH_SHA384,             /* empreinte SHA384 */  
    HASH_SHA512,             /* empreinte SHA512 */  
#endif /* HAVE_HASH_SHA512 */  
} hash_md_t;  
  
#define HASH_SHA HASH_SHA1  /* alias pour SHA1 */
```

L'interface : les fonctions de la bibliothèque

fichier en-tête : hash.h

```
int  hash_init(hash_t *, hash_md_t);  
void hash_free(hash_t *);
```

L'interface : les fonctions de la bibliothèque

fichier en-tête : `hash.h`

```
int  hash_init(hash_t *, hash_md_t);  
void hash_free(hash_t *);
```

-
- `hash_init(h, md)` initialise l'empreinte pointée par `h` avec le type de fonction de hachage `md`.

L'interface : les fonctions de la bibliothèque

fichier en-tête : `hash.h`

```
int  hash_init(hash_t *, hash_md_t);  
void hash_free(hash_t *);
```

-
- `hash_init(h, md)` initialise l'empreinte pointée par `h` avec le type de fonction de hachage `md`.
 - `hash_free(h)` libère les ressources allouées par l'empreinte pointée par `h`.

L'interface : les fonctions de la bibliothèque

fichier en-tête : `hash.h`

```
int  hash_init(hash_t *, hash_md_t);
void hash_free(hash_t *);

int  hash_update(hash_t *, const uint8_t *, size_t);
int  hash_digest(hash_t *);
int  hash_digestfile(hash_t *, const char *);
int  hash_reset(hash_t *);
```

L'interface : les fonctions de la bibliothèque

fichier en-tête : `hash.h`

```
int  hash_init(hash_t *, hash_md_t);
void hash_free(hash_t *);

int  hash_update(hash_t *, const uint8_t *, size_t);
int  hash_digest(hash_t *);
int  hash_digestfile(hash_t *, const char *);
int  hash_reset(hash_t *);
```

-
- `hash_update(h, data, dlen)` ajoute les données pointées par `data`, de longueur `dlen` dans les données à hacher. La fonction `hash_update` peut être appelée plusieurs fois.

L'interface : les fonctions de la bibliothèque

fichier en-tête : `hash.h`

```
int  hash_init(hash_t *, hash_md_t);
void hash_free(hash_t *);

int  hash_update(hash_t *, const uint8_t *, size_t);
int  hash_digest(hash_t *);
int  hash_digestfile(hash_t *, const char *);
int  hash_reset(hash_t *);
```

-
- `hash_update(h, data, dlen)` ajoute les données pointées par `data`, de longueur `dlen` dans les données à hacher. La fonction `hash_update` peut être appelée plusieurs fois.
 - `hash_digest(h)` calcule l'empreinte après un ou plusieurs appels à `hash_update`.

L'interface : les fonctions de la bibliothèque

fichier en-tête : `hash.h`

```
int  hash_init(hash_t *, hash_md_t);
void hash_free(hash_t *);

int  hash_update(hash_t *, const uint8_t *, size_t);
int  hash_digest(hash_t *);
int  hash_digestfile(hash_t *, const char *);
int  hash_reset(hash_t *);
```

-
- `hash_update(h, data, dlen)` ajoute les données pointées par `data`, de longueur `dlen` dans les données à hacher. La fonction `hash_update` peut être appelée plusieurs fois.
 - `hash_digest(h)` calcule l'empreinte après un ou plusieurs appels à `hash_update`.
 - `hash_digestfile(h, fname)` calcule l'empreinte du fichier de nom `fname`.

L'interface : les fonctions de la bibliothèque

fichier en-tête : `hash.h`

```
int  hash_init(hash_t *, hash_md_t);
void hash_free(hash_t *);

int  hash_update(hash_t *, const uint8_t *, size_t);
int  hash_digest(hash_t *);
int  hash_digestfile(hash_t *, const char *);
int  hash_reset(hash_t *);
```

-
- `hash_update(h, data, dlen)` ajoute les données pointées par `data`, de longueur `dlen` dans les données à hacher. La fonction `hash_update` peut être appelée plusieurs fois.
 - `hash_digest(h)` calcule l'empreinte après un ou plusieurs appels à `hash_update`.
 - `hash_digestfile(h, fname)` calcule l'empreinte du fichier de nom `fname`.
 - `hash_reset(h)` réinitialise l'empreinte pointée par `h` (annule tous les précédents appels à la fonction `hash_update`).

L'interface : les fonctions de la bibliothèque

fichier en-tête : hash.h

```
#include <stdio.h>
```

```
int  hash_init(hash_t *, hash_md_t);
```

```
void hash_free(hash_t *);
```

```
int  hash_update(hash_t *, const uint8_t *, size_t);
```

```
int  hash_digest(hash_t *);
```

```
int  hash_digestfile(hash_t *, const char *);
```

```
int  hash_reset(hash_t *);
```

```
void hash_print(const hash_t *, FILE *);
```

L'interface : les fonctions de la bibliothèque

fichier en-tête : `hash.h`

```
#include <stdio.h>
```

```
int  hash_init(hash_t *, hash_md_t);
```

```
void hash_free(hash_t *);
```

```
int  hash_update(hash_t *, const uint8_t *, size_t);
```

```
int  hash_digest(hash_t *);
```

```
int  hash_digestfile(hash_t *, const char *);
```

```
int  hash_reset(hash_t *);
```

```
void hash_print(const hash_t *, FILE *);
```

- `hash_print(h, os)` imprime l'empreinte (champ `h->md`) comme chaîne de caractères en hexadécimal sur le flux pointé par `os`.

L'implémentation : le tableau de pointeurs `hash_md_list`

```
#include <openssl/evp.h>
#include "hash.h"

/* Tableau de correspondance entre le type hash_md_t
 * et la fonction de hachage cryptographique fournie
 * par la bibliothèque OpenSSL
 */

static const EVP_MD *(*hash_md_list[])(void) = {
    EVP_md5,      /* fonction de hachage MD5 (OpenSSL) */
    EVP_sha1,     /* hash_md_list[HASH_SHA1] */
#ifdef HAVE_HASH_SHA256
    EVP_sha224,
    EVP_sha256,
#endif /* HAVE_HASH_SHA256 */
#ifdef HAVE_HASH_SHA512
    EVP_sha384,
    EVP_sha512,
#endif /* HAVE_HASH_SHA512 */
    NULL
};
```

L'implémentation : hash_init()

```
int hash_init(hash_t *h, hash_md_t md) {
    h->id = (void *) EVP_MD_CTX_create();
    if (!h->id) {
        error_ssl_set(__func__, "EVP_MD_CTX_create");
        return -1;
    }
}
```

L'implémentation : hash_init()

```
int hash_init(hash_t *h, hash_md_t md) {
    h->id = (void *) EVP_MD_CTX_create();
    if (!h->id) {
        error_ssl_set(__func__, "EVP_MD_CTX_create");
        return -1;
    }
    if (!EVP_DigestInit_ex((EVP_MD_CTX *) h->id,
        hash_md_list[md](), NULL)) {
        error_ssl_set(__func__, "EVP_DigestInit_ex");
        EVP_MD_CTX_destroy((EVP_MD_CTX *) h->id);
        return -1;
    }
};
```

L'implémentation : hash_init()

```
int hash_init(hash_t *h, hash_md_t md) {
    h->id = (void *) EVP_MD_CTX_create();
    if (!h->id) {
        error_ssl_set(__func__, "EVP_MD_CTX_create");
        return -1;
    }
    if (!EVP_DigestInit_ex((EVP_MD_CTX *) h->id,
        hash_md_list[md](), NULL)) {
        error_ssl_set(__func__, "EVP_DigestInit_ex");
        EVP_MD_CTX_destroy((EVP_MD_CTX *) h->id);
        return -1;
    };
    h->md_len = EVP_MD_CTX_size((EVP_MD_CTX *) h->id);
    h->md = (uint8_t *) malloc(h->md_len);
    if (!h->md) {
        error_sys_set(__func__, "malloc");
        EVP_MD_CTX_destroy((EVP_MD_CTX *) h->id);
        return -1;
    }
    return 0;
}
```

L'implémentation : hash_free() et hash_update()

```
void hash_free(hash_t *h)
{
    EVP_MD_CTX_destroy((EVP_MD_CTX *) h->id);
    (void) memset((void *) h->md, 0, h->md_len);
    free((void *) h->md);
}
```

L'implémentation : hash_free() et hash_update()

```
void hash_free(hash_t *h)
{
    EVP_MD_CTX_destroy((EVP_MD_CTX *) h->id);
    (void) memset((void *) h->md, 0, h->md_len);
    free((void *) h->md);
}
```

```
int hash_update(hash_t *h, const uint8_t *d, size_t dl)
{
    if (!EVP_DigestUpdate((EVP_MD_CTX *) h->id,
        (const void *) d, dl)) {
        error_ssl_set(__func__, "EVP_DigestUpdate");
        return -1;
    }
    return 0;
}
```

L'implémentation : `hash_digest()` et `hash_reset()`

```
int hash_digest(hash_t *h)
{
    if (!EVP_DigestFinal_ex((EVP_MD_CTX *) h->id, h->md,
        (unsigned int *) &h->md_len)) {
        error_ssl_set(__func__, "EVP_DigestFinal_ex");
        return -1;
    }
    return 0;
}
```

L'implémentation : `hash_digest()` et `hash_reset()`

```
int hash_digest(hash_t *h)
{
    if (!EVP_DigestFinal_ex((EVP_MD_CTX *) h->id, h->md,
        (unsigned int *) &h->md_len)) {
        error_ssl_set(__func__, "EVP_DigestFinal_ex");
        return -1;
    }
    return 0;
}
```

```
int hash_reset(hash_t *h)
{
    if (!EVP_DigestInit_ex(
        (EVP_MD_CTX *) h->id, EVP_MD_CTX_md((EVP_MD_CTX *) h->id),
        NULL)) {
        error_ssl_set(__func__, "EVP_DigestInit_ex");
        return -1;
    }
    return 0;
}
```


L'implémentation : hash_print()

```
void hash_print(const hash_t *h, FILE *os)
{
    int s = EVP_MD_CTX_size((EVP_MD_CTX *) h->id);
    uint8_t *p;

    for (p = h->md; p - h->md < s; p++)
        (void) fprintf(os, "%02hhx", *p);
}
```

L'implémentation : hash_print()

```
void hash_print(const hash_t *h, FILE *os)
{
    int s = EVP_MD_CTX_size((EVP_MD_CTX *) h->id);
    uint8_t *p;

    for (p = h->md; p - h->md < s; p++)
        (void) fprintf(os, "%02hhx", *p);
}
```

L'implémentation de la dernière fonction de la bibliothèque, la fonction `hash_digestfile`, est laissée en exercice 😊

Une (modeste) bibliothèque de gestion de listes

- On veut créer une nouvelle structure de données, la **liste** (chaînée), analogue au tableau, sauf que les éléments d'une liste seront **alloués dynamiquement** et donc pas forcément contigus en mémoire.
- Les données des éléments d'une liste seront toutes du **même type**, ce dernier étant **quelconque**. Ce seront donc des listes d'entiers, de flottants, voire de listes d'entiers.
- Le type des données d'un élément étant à priori quelconque, nous utiliserons le type `void` pour le représenter.
- Outre un pointeur vers les données, chaque élément contiendra un pointeur vers l'élément suivant ou le pointeur `NULL` si c'est le dernier.
- Une liste sera représentée comme un **pointeur** vers le premier élément de la liste, qui pointe sur le suivant, etc..., d'où la terminologie liste (simplement) **chaînée**.
- La liste **vide** sera, par convention, représentée par le pointeur `NULL`.

Une (modeste) bibliothèque de gestion de listes

- On veut créer une nouvelle structure de données, la **liste** (chaînée), analogue au tableau, sauf que les éléments d'une liste seront **alloués dynamiquement** et donc pas forcément contigus en mémoire.
- Les données des éléments d'une liste seront toutes du **même type**, ce dernier étant **quelconque**. Ce seront donc des listes d'entiers, de flottants, voire de listes d'entiers.
- Le type des données d'un élément étant à priori quelconque, nous utiliserons le type `void` pour le représenter.
- Outre un pointeur vers les données, chaque élément contiendra un pointeur vers l'élément suivant ou le pointeur `NULL` si c'est le dernier.
- Une liste sera représentée comme un **pointeur** vers le premier élément de la liste, qui pointe sur le suivant, etc..., d'où la terminologie liste (simplement) **chaînée**.
- La liste **vide** sera, par convention, représentée par le pointeur `NULL`.

Une (modeste) bibliothèque de gestion de listes

- On veut créer une nouvelle structure de données, la **liste** (chaînée), analogue au tableau, sauf que les éléments d'une liste seront **alloués dynamiquement** et donc pas forcément contigus en mémoire.
- Les données des éléments d'une liste seront toutes du **même type**, ce dernier étant **quelconque**. Ce seront donc des listes d'entiers, de flottants, voire de listes d'entiers.
- Le type des données d'un élément étant à priori quelconque, nous utiliserons le type **void** pour le représenter.
- Outre un pointeur vers les données, chaque élément contiendra un pointeur vers l'élément suivant ou le pointeur **NULL** si c'est le dernier.
- Une liste sera représentée comme un **pointeur** vers le premier élément de la liste, qui pointe sur le suivant, etc..., d'où la terminologie liste (simplement) **chaînée**.
- La liste **vide** sera, par convention, représentée par le pointeur **NULL**.

Une (modeste) bibliothèque de gestion de listes

- On veut créer une nouvelle structure de données, la **liste** (chaînée), analogue au tableau, sauf que les éléments d'une liste seront **alloués dynamiquement** et donc pas forcément contigus en mémoire.
- Les données des éléments d'une liste seront toutes du **même type**, ce dernier étant **quelconque**. Ce seront donc des listes d'entiers, de flottants, voire de listes d'entiers.
- Le type des données d'un élément étant à priori quelconque, nous utiliserons le type **void** pour le représenter.
- Outre un pointeur vers les données, chaque élément contiendra un pointeur vers l'élément suivant ou le pointeur **NULL** si c'est le dernier.
- Une liste sera représentée comme un **pointeur** vers le premier élément de la liste, qui pointe sur le suivant, etc..., d'où la terminologie liste (simplement) **chaînée**.
- La liste **vide** sera, par convention, représentée par le pointeur **NULL**.

Une (modeste) bibliothèque de gestion de listes

- On veut créer une nouvelle structure de données, la **liste** (chaînée), analogue au tableau, sauf que les éléments d'une liste seront **alloués dynamiquement** et donc pas forcément contigus en mémoire.
- Les données des éléments d'une liste seront toutes du **même type**, ce dernier étant **quelconque**. Ce seront donc des listes d'entiers, de flottants, voire de listes d'entiers.
- Le type des données d'un élément étant à priori quelconque, nous utiliserons le type **void** pour le représenter.
- Outre un pointeur vers les données, chaque élément contiendra un pointeur vers l'élément suivant ou le pointeur **NULL** si c'est le dernier.
- Une liste sera représentée comme un **pointeur** vers le premier élément de la liste, qui pointe sur le suivant, etc..., d'où la terminologie liste (simplement) **chaînée**.
- La liste **vide** sera, par convention, représentée par le pointeur **NULL**.

Une (modeste) bibliothèque de gestion de listes

- On veut créer une nouvelle structure de données, la **liste** (chaînée), analogue au tableau, sauf que les éléments d'une liste seront **alloués dynamiquement** et donc pas forcément contigus en mémoire.
- Les données des éléments d'une liste seront toutes du **même type**, ce dernier étant **quelconque**. Ce seront donc des listes d'entiers, de flottants, voire de listes d'entiers.
- Le type des données d'un élément étant à priori quelconque, nous utiliserons le type **void** pour le représenter.
- Outre un pointeur vers les données, chaque élément contiendra un pointeur vers l'élément suivant ou le pointeur **NULL** si c'est le dernier.
- Une liste sera représentée comme un **pointeur** vers le premier élément de la liste, qui pointe sur le suivant, etc..., d'où la terminologie liste (simplement) **chaînée**.
- La liste **vide** sera, par convention, représentée par le pointeur **NULL**.

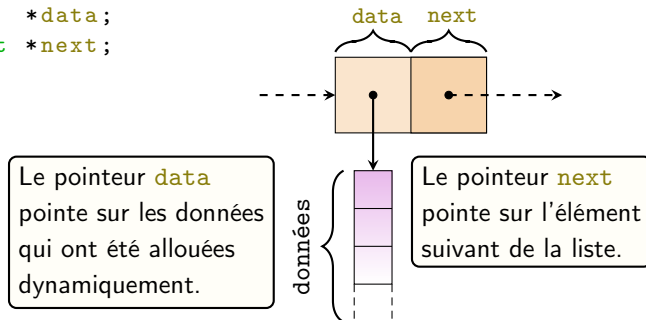
Élément d'une liste : implémentation

```
/* définition d'un élément (structure pseudo-réursive) */  
  
typedef struct __elt {  
    void      *data;  
    struct __elt *next;  
} elt_t;
```

Élément d'une liste : implémentation

```
/* définition d'un élément (structure pseudo-réursive) */
```

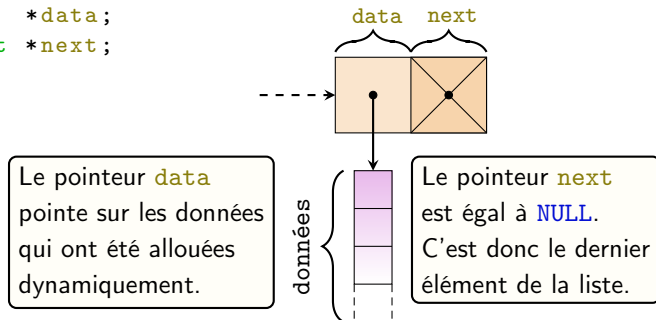
```
typedef struct __elt {  
    void      *data;  
    struct __elt *next;  
} elt_t;
```



Élément d'une liste : implémentation

```
/* définition d'un élément (structure pseudo-réursive) */
```

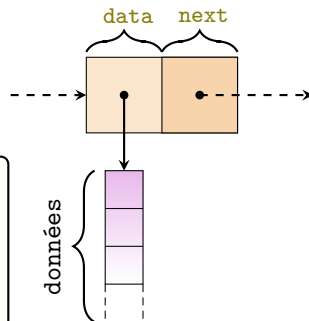
```
typedef struct __elt {  
    void      *data;  
    struct __elt *next;  
} elt_t;
```



Élément d'une liste : implémentation

```
/* définition d'un élément (structure pseudo-réursive) */
```

```
typedef struct __elt {  
    void      *data;  
    struct __elt *next;  
} elt_t;
```



Un élément d'une liste est donc une structure composée de deux pointeurs, l'un pointant sur les données, l'autre pointant sur l'élément suivant ou étant, si c'est le dernier, le pointeur **NULL**.

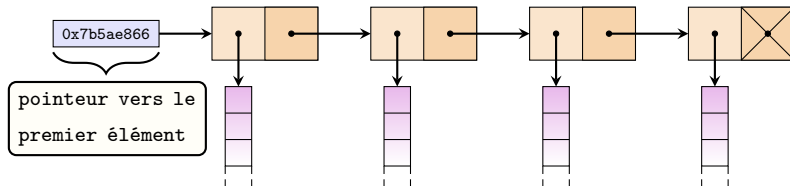
Élément d'une liste : implémentation

```
/* définition d'une liste (simplement chaînée) */  
  
typedef elt_t *list_t;
```

Élément d'une liste : implémentation

```
/* définition d'une liste (simplement chaînée) */
```

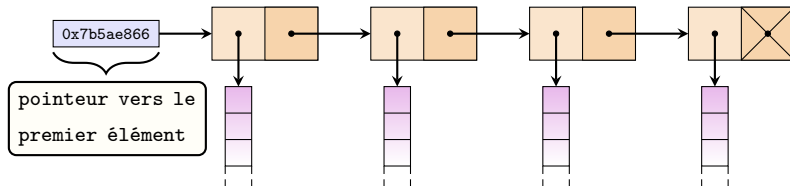
```
typedef elt_t *list_t;
```



Élément d'une liste : implémentation

```
/* définition d'une liste (simplement chaînée) */
```

```
typedef elt_t *list_t;
```



liste vide

NULL

Longueur d'une liste : la fonction `list_length`

```
#ifndef LIST_H
#define LIST_H

typedef struct __elt {
    void          *data;
    struct __elt *next;
} elt_t;
typedef elt_t *list_t;

size_t list_length(const list_t);

#endif /* LIST_H */
```


Longueur d'une liste : la fonction `list_length`

```
#ifndef LIST_H
#define LIST_H

typedef struct __elt {
    void *data;
    struct __elt *next;
} elt_t;
typedef elt_t *list_t;
size_t list_length(const list_t);

#endif /* LIST_H */
```

prototype de `list_length`

Longueur d'une liste : la fonction `list_length`

```
#ifndef LIST_H
#define LIST_H

typedef struct __elt {
    void      *data;
    struct __elt *next;
} elt_t;
typedef elt_t *list_t;
size_t list_length(const list_t);

#endif /* LIST_H */
```

prototype de `list_length`

```
size_t list_length(const list_t l)
{
    size_t len = 0;

    for (const elt_t *e = l; e; e = e->next) len++;
    return len;
}
```

Longueur d'une liste : la fonction `list_length`

```
#ifndef LIST_H
#define LIST_H

typedef struct __elt {
    void      *data;
    struct __elt *next;
} elt_t;
typedef elt_t *list_t;
size_t list_length(const list_t);

#endif /* LIST_H */
```

prototype de `list_length`

```
size_t list_length(const list_t l)
{
    size_t len = 0;
    for (const elt_t *e = l; e; e = e->next) len++;
    return len;
}
```

parcours de la liste

Ajout d'un élément : la fonction `list_insert` (version 1)

Nous allons maintenant construire la fonction `list_insert` permettant d'insérer un nouvel élément *en tête* d'une liste :

```
#ifndef LIST_H
#define LIST_H
```

```
typedef struct __elt {
    void      *data;
    struct __elt *next;
} elt_t;
typedef elt_t *list_t;
```

```
list_t list_insert(list_t, const void *);
```

```
#endif /* LIST_H */
```

pointeur vers les données
de l'élément à insérer

la liste à traiter

valeur de retour : un pointeur
vers le nouvel élément inséré

Une première proposition de **prototype** pour `list_insert`.

Ajout d'un élément : la fonction `list_insert` (version 2)

Nous allons maintenant construire la fonction `list_insert` permettant d'insérer un nouvel élément *en tête* d'une liste :

```
#ifndef LIST_H
#define LIST_H
```

```
typedef struct __elt {
    void      *data;
    struct __elt *next;
} elt_t;
typedef elt_t *list_t;
```

```
int list_insert(list_t *, const void *);
```

```
#endif /* LIST_H */
```

pointeur vers les données
de l'élément à insérer

pointeur vers la
liste à traiter

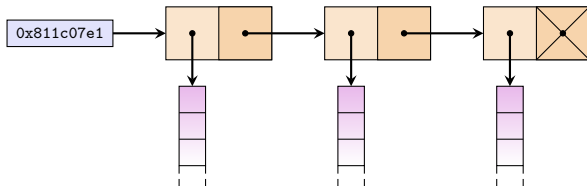
valeur de retour :
un code d'erreur

Une autre proposition de **prototype** sensiblement différente...

Le code de la fonction `list_insert` (version 1)

```
list_t list_insert(list_t l, const void *d)  
{
```

```
}
```

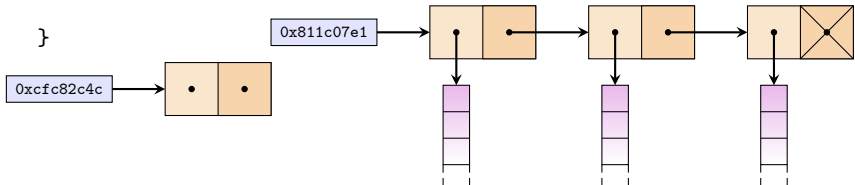


Le code de la fonction `list_insert` (version 1)

```
list_t list_insert(list_t l, const void *d)
{
    elt_t *e;

    e = malloc(sizeof(*e));
    if (e == NULL)
        return NULL;

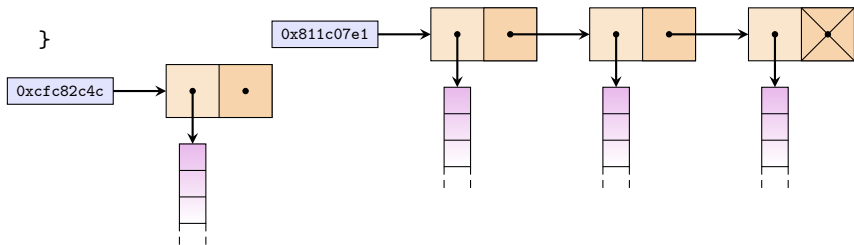
    }
```



Le code de la fonction `list_insert` (version 1)

```
list_t list_insert(list_t l, const void *d)
{
    elt_t *e;

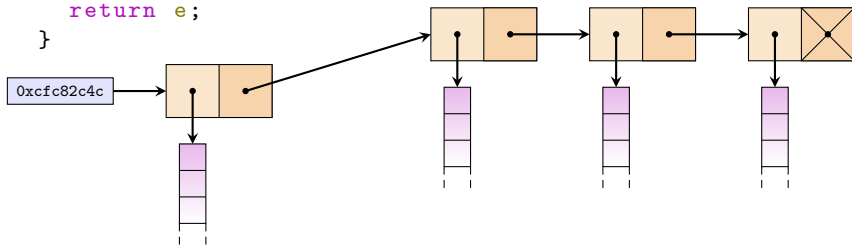
    e = malloc(sizeof(*e));
    if (e == NULL)
        return NULL;
    e->data = (void *) d; /* données non copiées ! */
}
```



Le code de la fonction `list_insert` (version 1)

```
list_t list_insert(list_t l, const void *d)
{
    elt_t *e;

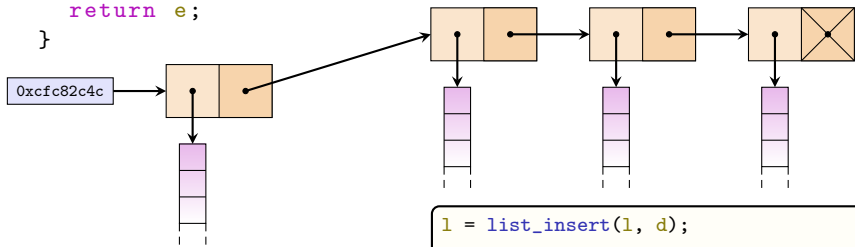
    e = malloc(sizeof(*e));
    if (e == NULL)
        return NULL;
    e->data = (void *) d; /* données non copiées ! */
    e->next = l;
    return e;
}
```



Le code de la fonction `list_insert` (version 1)

```
list_t list_insert(list_t l, const void *d)
{
    elt_t *e;

    e = malloc(sizeof(*e));
    if (e == NULL)
        return NULL;
    e->data = (void *) d; /* données non copiées ! */
    e->next = l;
    return e;
}
```



```
l = list_insert(l, d);
if (l == NULL) /* traitement erreur */
```

Le code de la fonction `list_insert` (version 2)

```
int list_insert(list_t *l, const void *d)
{
    elt_t *e;

    e = malloc(sizeof(*e));
    if (e == NULL)
        return -1;
    e->data = (void *) d; /* données non copiées ! */
    e->next = *l;
    *l = e;
    return 0;
}
```

Le code de la fonction `list_insert` (version 2)

```
int list_insert(list_t *l, const void *d)
{
    elt_t *e;

    e = malloc(sizeof(*e));
    if (e == NULL)
        return -1;
    e->data = (void *) d; /* données non copiées ! */
    e->next = *l;
    *l = e;
    return 0;
}
```

```
int i = 2;
list_t l = NULL;

if (list_insert(&l, &i) == -1)
    /* traitement de l'erreur */
```

list_insert : le problème des données

- On a vu que les données (2^{ème} argument de `list_insert`) ne sont pas copiées avant d'être transmises au nouvel élément inséré, **seul le pointeur** vers ces données est transmis.
- Si, quelque part ailleurs dans le programme, ces données sont libérées (avec `free` par exemple), la liste, dans laquelle aura été inséré cet élément, deviendra **incohérente**.
- Il faut donc au préalable copier les données. Le problème est que ces données sont de type générique (`void`) et on ne sait pas à priori comment les copier.
- Une solution est d'ajouter à la fonction `list_insert` un paramètre, celui-ci étant un pointeur vers la fonction réalisant la copie des données. Il aura donc pour type : `void (*)(const void *)` et la fonction `list_insert` pour prototype :

```
list_t list_insert(list_t, const void *, void (*)(const void *));
```

list_insert : le problème des données

- On a vu que les données (2^{ème} argument de `list_insert`) ne sont pas copiées avant d'être transmises au nouvel élément inséré, **seul le pointeur** vers ces données est transmis.
- Si, quelque part ailleurs dans le programme, ces **données sont libérées** (avec `free` par exemple), la liste, dans laquelle aura été inséré cet élément, deviendra **incohérente**.
- Il faut donc au préalable copier les données. Le problème est que ces données sont de type générique (`void`) et on ne sait pas à priori comment les copier.
- Une solution est d'ajouter à la fonction `list_insert` un paramètre, celui-ci étant un pointeur vers la fonction réalisant la copie des données. Il aura donc pour type : `void (*)(const void *)` et la fonction `list_insert` pour prototype :

```
list_t list_insert(list_t, const void *, void (*)(const void *));
```

list_insert : le problème des données

- On a vu que les données (2^{ème} argument de `list_insert`) ne sont pas copiées avant d'être transmises au nouvel élément inséré, **seul le pointeur** vers ces données est transmis.
- Si, quelque part ailleurs dans le programme, ces **données sont libérées** (avec `free` par exemple), la liste, dans laquelle aura été inséré cet élément, deviendra **incohérente**.
- Il faut donc au préalable copier les données. Le problème est que ces données sont de type générique (`void`) et on ne sait pas à priori comment les copier.
- Une solution est d'ajouter à la fonction `list_insert` un paramètre, celui-ci étant un pointeur vers la fonction réalisant la copie des données. Il aura donc pour type : `void (*)(const void *)` et la fonction `list_insert` pour prototype :

```
list_t list_insert(list_t, const void *, void (*)(const void *));
```

list_insert : le problème des données

- On a vu que les données (2^{ème} argument de `list_insert`) ne sont pas copiées avant d'être transmises au nouvel élément inséré, **seul le pointeur** vers ces données est transmis.
- Si, quelque part ailleurs dans le programme, ces **données sont libérées** (avec `free` par exemple), la liste, dans laquelle aura été inséré cet élément, deviendra **incohérente**.
- Il faut donc au préalable copier les données. Le problème est que ces données sont de type générique (`void`) et on ne sait pas à priori comment les copier.
- Une solution est d'ajouter à la fonction `list_insert` un paramètre, celui-ci étant un pointeur vers la fonction réalisant la copie des données. Il aura donc pour type : `void (*)(const void *)` et la fonction `list_insert` pour prototype :

```
list_t list_insert(list_t, const void *, void (*)(const void *));
```


list_insert : le code modifié

```
#ifndef LIST_H
#define LIST_H
typedef void *(*copy_t)(const void *); /* plus lisible */
list_t list_insert(list_t, const void *, copy_t);
#endif /* LIST_H */
```

list_insert : le code modifié

```
#ifndef LIST_H
#define LIST_H
typedef void *(*copy_t)(const void *); /* plus lisible */
list_t list_insert(list_t, const void *, copy_t);
#endif /* LIST_H */
```

```
list_t list_insert(list_t l, const void *d, copy_t c) {
    elt_t *e;

    e = malloc(sizeof(*e));
    if (e == NULL) return NULL;
    e->data = c(d); /* les données sont copiées */
    if (e->data == NULL) { /* la copie peut échouer */
        free(e);
        return NULL;
    }
    e->next = l;
    return e;
}
```

Impression d'une liste : la fonction `list_print`

```
#ifndef LIST_H
#define LIST_H

typedef void (*print_t)(const void *, FILE *);
void list_print(const list_t, FILE *, print_t);

#endif /* LIST_H */
```

Impression d'une liste : la fonction list_print

```
#ifndef LIST_H
#define LIST_H

typedef void (*print_t)(const void *, FILE *);
void list_print(const list_t, FILE *, print_t);

#endif /* LIST_H */

void list_print(const list_t l, FILE *os, print_t p)
{
    fprintf(os, "[" );
    for (const elt_t *e = l; e; e = e->next) {
        p(e->data, os);
        if (e->next) /* e n'est pas le dernier élément */
            fprintf(os, "□,");
    }
    fprintf(os, "]\n");
}
```

Désallocation d'une liste : la fonction `list_free`

```
#ifndef LIST_H
#define LIST_H

typedef void (*free_t)(void *);
void list_free(list_t, free_t);

#endif /* LIST_H */
```

Désallocation d'une liste : la fonction `list_free`

```
#ifndef LIST_H
#define LIST_H

typedef void (*free_t)(void *);
void list_free(list_t, free_t);

#endif /* LIST_H */

void list_free(list_t l, free_t f)
{
    for (elt_t *e = l; e;) {
        elt_t *tmp = e->next;

        f(e->data);
        free(e);
        e = tmp;
    }
}
```

Désallocation d'une liste : la fonction `list_free`

```
#ifndef LIST_H
#define LIST_H

typedef void (*free_t)(void *);
void list_free(list_t, free_t);

#endif /* LIST_H */

void list_free(list_t l, free_t f)
{
    for (elt_t *e = l; e;) {
        elt_t *tmp = e->next;
        f(e->data);
        free(e);
        e = tmp; /* élément suivant */
    }
}
```

Il est nécessaire de sauvegarder `e->next` car après avoir libéré `e`, on peut plus accéder à cette valeur et donc à l'élément suivant de la liste.

Spécialisation des listes génériques

- Nous allons maintenant **spécialiser les listes génériques** aux listes d'entiers, c'est à dire aux listes dont les données sont des objets de type **int**.
- Cette spécialisation va consister pour l'essentiel à coder les **fonctions auxiliaires** de copie, de désallocation et d'impression des données.
- Ces fonctions auxiliaires n'apparaîtront pas dans l'interface et seront donc déclarées comme **static** dans le module implémentant les fonctions de la bibliothèque :

```
- static void *copy_int(const void *);  
- static void free_int(void *); (On verra, que dans  
l'implémentation proposée, cette fonction sera tout simplement la  
primitive free de la bibliothèque standard)  
- static void print_int(const void *, FILE *);
```

- Cette approche permet de réutiliser la quasi-totalité du code développé pour les listes génériques et est facilement transposable à d'autres types de données (**float**, **double**, etc.).

Spécialisation des listes génériques

- Nous allons maintenant **spécialiser les listes génériques** aux listes d'entiers, c'est à dire aux listes dont les données sont des objets de type `int`.
- Cette spécialisation va consister pour l'essentiel à coder les **fonctions auxiliaires** de copie, de désallocation et d'impression des données.
- Ces fonctions auxiliaires n'apparaîtront pas dans l'interface et seront donc déclarées comme `static` dans le module implémentant les fonctions de la bibliothèque :

```
- static void *copy_int(const void *);  
- static void free_int(void *); (On verra, que dans  
l'implémentation proposée, cette fonction sera tout simplement la  
primitive free de la bibliothèque standard)  
- static void print_int(const void *, FILE *);
```

- Cette approche permet de réutiliser la quasi-totalité du code développé pour les listes génériques et est facilement transposable à d'autres types de données (`float`, `double`, etc.).

Spécialisation des listes génériques

- Nous allons maintenant **spécialiser les listes génériques** aux listes d'entiers, c'est à dire aux listes dont les données sont des objets de type `int`.
- Cette spécialisation va consister pour l'essentiel à coder les **fonctions auxiliaires** de copie, de désallocation et d'impression des données.
- Ces fonctions auxiliaires n'apparaîtront pas dans l'interface et seront donc déclarées comme **static** dans le module implémentant les fonctions de la bibliothèque :

```
- static void *copy_int(const void *);  
- static void free_int(void *); (On verra, que dans  
  l'implémentation proposée, cette fonction sera tout simplement la  
  primitive free de la bibliothèque standard)  
- static void print_int(const void *, FILE *);
```

- Cette approche permet de réutiliser la quasi-totalité du code développé pour les listes génériques et est facilement transposable à d'autres types de données (`float`, `double`, etc.).

Spécialisation des listes génériques

- Nous allons maintenant **spécialiser les listes génériques** aux listes d'entiers, c'est à dire aux listes dont les données sont des objets de type `int`.
- Cette spécialisation va consister pour l'essentiel à coder les **fonctions auxiliaires** de copie, de désallocation et d'impression des données.
- Ces fonctions auxiliaires n'apparaîtront pas dans l'interface et seront donc déclarées comme **static** dans le module implémentant les fonctions de la bibliothèque :

- `static void *copy_int(const void *)`;
- `static void free_int(void *)`; *(On verra, que dans l'implémentation proposée, cette fonction sera tout simplement la primitive `free` de la bibliothèque standard)*
- `static void print_int(const void *, FILE *)`;

- Cette approche permet de réutiliser la quasi-totalité du code développé pour les listes génériques et est facilement transposable à d'autres types de données (`float`, `double`, etc.).

Spécialisation des listes génériques

- Nous allons maintenant **spécialiser les listes génériques** aux listes d'entiers, c'est à dire aux listes dont les données sont des objets de type `int`.
- Cette spécialisation va consister pour l'essentiel à coder les **fonctions auxiliaires** de copie, de désallocation et d'impression des données.
- Ces fonctions auxiliaires n'apparaîtront pas dans l'interface et seront donc déclarées comme **static** dans le module implémentant les fonctions de la bibliothèque :
 - `static void *copy_int(const void *)`;
 - `static void free_int(void *)`; (*On verra, que dans l'implémentation proposée, cette fonction sera tout simplement la primitive `free` de la bibliothèque standard*)
 - `static void print_int(const void *, FILE *)`;
- Cette approche permet de réutiliser la quasi-totalité du code développé pour les listes génériques et est facilement transposable à d'autres types de données (`float`, `double`, etc.).

Spécialisation des listes génériques

- Nous allons maintenant **spécialiser les listes génériques** aux listes d'entiers, c'est à dire aux listes dont les données sont des objets de type `int`.
- Cette spécialisation va consister pour l'essentiel à coder les **fonctions auxiliaires** de copie, de désallocation et d'impression des données.
- Ces fonctions auxiliaires n'apparaîtront pas dans l'interface et seront donc déclarées comme `static` dans le module implémentant les fonctions de la bibliothèque :
 - `static void *copy_int(const void *)`;
 - `static void free_int(void *)`; (*On verra, que dans l'implémentation proposée, cette fonction sera tout simplement la primitive `free` de la bibliothèque standard*)
 - `static void print_int(const void *, FILE *)`;
- Cette approche permet de réutiliser la quasi-totalité du code développé pour les listes génériques et est facilement transposable à d'autres types de données (`float`, `double`, etc.).

Spécialisation des listes génériques

- Nous allons maintenant **spécialiser les listes génériques** aux listes d'entiers, c'est à dire aux listes dont les données sont des objets de type `int`.
- Cette spécialisation va consister pour l'essentiel à coder les **fonctions auxiliaires** de copie, de désallocation et d'impression des données.
- Ces fonctions auxiliaires n'apparaîtront pas dans l'interface et seront donc déclarées comme `static` dans le module implémentant les fonctions de la bibliothèque :
 - `static void *copy_int(const void *)`;
 - `static void free_int(void *)`; (*On verra, que dans l'implémentation proposée, cette fonction sera tout simplement la primitive `free` de la bibliothèque standard*)
 - `static void print_int(const void *, FILE *)`;
- Cette approche permet de réutiliser la quasi-totalité du code développé pour les listes génériques et est facilement transposable à d'autres types de données (`float`, `double`, etc.).

L'interface pour les listes d'entiers

```
#ifndef LISTINT_H
#define LISTINT_H

#include <list.h>

typedef list_t listint_t;

inline size_t listint_length(const listint_t l)
{
    return list_length((const list_t) l);
}
listint_t listint_insert(listint_t, int);
void      listint_print(const listint_t, FILE *);
void      listint_free(listint_t);

#endif /* LISTINT_H */
```

Impression d'une liste d'entiers : `listint_print`

On a besoin de définir une fonction auxiliaire d'impression des données, la fonction `print_int` (dans la classe `static` car c'est une fonction locale) :

```
static void print_int(const void *d, FILE *os)
{
    fprintf(os, "%d", *(int *) d);
}
```


Impression d'une liste d'entiers : `listint_print`

On a besoin de définir une fonction auxiliaire d'impression des données, la fonction `print_int` (dans la classe `static` car c'est une fonction locale) :

```
static void print_int(const void *d, FILE *os)
{
    fprintf(os, "%d", *(int *) d);
}
```

Conversion nécessaire avant d'appliquer l'opérateur d'indirection * car `d` est de type `void *`.

Impression d'une liste d'entiers : `listint_print`

On a besoin de définir une fonction auxiliaire d'impression des données, la fonction `print_int` (dans la classe `static` car c'est une fonction locale) :

```
static void print_int(const void *d, FILE *os)
{
    fprintf(os, "%d", *(int *) d);
}
```

Conversion nécessaire avant d'appliquer l'opérateur d'indirection * car `d` est de type `void *`.

```
void listint_print(const listint_t l, FILE *os)
{
    list_print((const list_t) l, os, print_int);
}
```

Ajout d'un entier : la fonction `listint_insert`

```
static void *copy_int(const void *n)
{
    void *ret;

    ret = malloc(sizeof(int));
    if (!ret)
        return NULL;
    *(int *) ret = *(int *) n;
    return ret;
}

listint_t listint_insert(listint_t l, int n)
{
    return list_insert((list_t) l, &n, copy_int);
}
```

Ajout d'un entier : la fonction `listint_insert`

```
static void *copy_int(const void *n)
{
    void *ret;

    ret = malloc(sizeof(int));
    if (!ret)
        return NULL;
    *(int *) ret = *(int *) n;
    return ret;
}

listint_t listint_insert(listint_t l, int n)
{
    return list_insert((list_t) l, &n, copy_int);
}
```

Pointeur (constant) vers la variable temporaire `n`

Désallocation d'une liste d'entiers : `listint_free`

On n'a pas besoin ici de définir une fonction auxiliaire de désallocation mémoire. En effet, on a utilisé pour allouer les données la primitive système `malloc`. Il suffit donc de passer en argument à `listint_free`, la primitive système `free`. Ce qui donne le code suivant :

```
void listint_free(listint_t l)
{
    list_free((list_t) l, free);
}
```

Une optimisation des listes génériques (1/2)

- Le type d'une liste générique, à savoir un pointeur vers le premier élément, est certes simple mais pas forcément optimal.
- Par exemple, pour obtenir la longueur d'une liste, il faut parcourir celle-ci, ce qui peut prendre du temps au delà d'une certaine taille.
- On pourrait donc envisager d'ajouter un champ longueur dans le type `list_t`. Ainsi le type `list_t` deviendrait une structure composée de deux champs : un *pointeur vers le premier élément* et le nouveau *champ longueur*.
- Pour optimiser des fonctions telles que `list_append`, il conviendrait d'également ajouter le champ *pointeur vers le dernier élément*.
- Au final, cela donnerait :

```
typedef struct { /* structure anonyme */
    elt_t *first; /* pointeur vers le premier élément */
    elt_t *last;  /* pointeur vers le dernier élément */
    size_t len;   /* longueur de la liste */
} list_t;
```

Une optimisation des listes génériques (1/2)

- Le type d'une liste générique, à savoir un pointeur vers le premier élément, est certes simple mais pas forcément optimal.
- Par exemple, pour obtenir la longueur d'une liste, il faut parcourir celle-ci, ce qui peut prendre du temps au delà d'une certaine taille.
- On pourrait donc envisager d'ajouter un champ longueur dans le type `list_t`. Ainsi le type `list_t` deviendrait une structure composée de deux champs : un *pointeur vers le premier élément* et le nouveau *champ longueur*.
- Pour optimiser des fonctions telles que `list_append`, il conviendrait d'également ajouter le champ *pointeur vers le dernier élément*.
- Au final, cela donnerait :

```
typedef struct { /* structure anonyme */
    elt_t *first; /* pointeur vers le premier élément */
    elt_t *last;  /* pointeur vers le dernier élément */
    size_t len;   /* longueur de la liste */
} list_t;
```

Une optimisation des listes génériques (1/2)

- Le type d'une liste générique, à savoir un pointeur vers le premier élément, est certes simple mais pas forcément optimal.
- Par exemple, pour obtenir la longueur d'une liste, il faut parcourir celle-ci, ce qui peut prendre du temps au delà d'une certaine taille.
- On pourrait donc envisager d'ajouter un champ longueur dans le type `list_t`. Ainsi le type `list_t` deviendrait une structure composée de deux champs : un *pointeur vers le premier élément* et le nouveau *champ longueur*.
- Pour optimiser des fonctions telles que `list_append`, il conviendrait d'également ajouter le champ *pointeur vers le dernier élément*.
- Au final, cela donnerait :

```
typedef struct { /* structure anonyme */
    elt_t *first; /* pointeur vers le premier élément */
    elt_t *last;  /* pointeur vers le dernier élément */
    size_t len;   /* longueur de la liste */
} list_t;
```


Une optimisation des listes génériques (1/2)

- Le type d'une liste générique, à savoir un pointeur vers le premier élément, est certes simple mais pas forcément optimal.
- Par exemple, pour obtenir la longueur d'une liste, il faut parcourir celle-ci, ce qui peut prendre du temps au delà d'une certaine taille.
- On pourrait donc envisager d'ajouter un champ longueur dans le type `list_t`. Ainsi le type `list_t` deviendrait une structure composée de deux champs : un *pointeur vers le premier élément* et le nouveau *champ longueur*.
- Pour optimiser des fonctions telles que `list_append`, il conviendrait d'également ajouter le champ *pointeur vers le dernier élément*.
- Au final, cela donnerait :

```
typedef struct { /* structure anonyme */
    elt_t *first; /* pointeur vers le premier élément */
    elt_t *last;  /* pointeur vers le dernier élément */
    size_t len;   /* longueur de la liste */
} list_t;
```

Une optimisation des listes génériques (1/2)

- Le type d'une liste générique, à savoir un pointeur vers le premier élément, est certes simple mais pas forcément optimal.
- Par exemple, pour obtenir la longueur d'une liste, il faut parcourir celle-ci, ce qui peut prendre du temps au delà d'une certaine taille.
- On pourrait donc envisager d'ajouter un champ longueur dans le type `list_t`. Ainsi le type `list_t` deviendrait une structure composée de deux champs : un *pointeur vers le premier élément* et le nouveau *champ longueur*.
- Pour optimiser des fonctions telles que `list_append`, il conviendrait d'également ajouter le champ *pointeur vers le dernier élément*.
- Au final, cela donnerait :

```
typedef struct { /* structure anonyme */
    elt_t *first; /* pointeur vers le premier élément */
    elt_t *last;  /* pointeur vers le dernier élément */
    size_t len;   /* longueur de la liste */
} list_t;
```

Une optimisation des listes génériques (2/2)

- Le problème avec cette première approche, est que `list_t` n'est plus un pointeur.
- En soi, ce n'est pas un problème, sauf qu'il serait plus agréable que `list_t` soit « naturellement » un pointeur.
- Il y a un moyen simple de réaliser cela : définir `list_t` non pas comme une structure décrite précédemment mais comme un tableau d'une (donc un **tableau de longueur 1**) telle structure. Du coup `list_t` redevient un pointeur (constant) vers le premier (et unique) élément.
- En résumé, on a :

```
typedef struct {  
    elt_t *first;  
    elt_t *last;  
    size_t len;  
} list_t[1]; /* tableau de une structure */
```

Une optimisation des listes génériques (2/2)

- Le problème avec cette première approche, est que `list_t` n'est plus un pointeur.
- En soi, ce n'est pas un problème, sauf qu'il serait plus agréable que `list_t` soit « naturellement » un pointeur.
- Il y a un moyen simple de réaliser cela : définir `list_t` non pas comme une structure décrite précédemment mais comme un tableau d'une (donc un **tableau de longueur 1**) telle structure. Du coup `list_t` redevient un pointeur (constant) vers le premier (et unique) élément.
- En résumé, on a :

```
typedef struct {  
    elt_t *first;  
    elt_t *last;  
    size_t len;  
} list_t[1]; /* tableau de une structure */
```

Une optimisation des listes génériques (2/2)

- Le problème avec cette première approche, est que `list_t` n'est plus un pointeur.
- En soi, ce n'est pas un problème, sauf qu'il serait plus agréable que `list_t` soit « naturellement » un pointeur.
- Il y a un moyen simple de réaliser cela : définir `list_t` non pas comme une structure décrite précédemment mais comme un tableau d'**une** (donc un **tableau de longueur 1**) telle structure. Du coup `list_t` redevient un pointeur (constant) vers le premier (et unique) élément.
- En résumé, on a :

```
typedef struct {  
    elt_t *first;  
    elt_t *last;  
    size_t len;  
} list_t[1]; /* tableau de une structure */
```

Une optimisation des listes génériques (2/2)

- Le problème avec cette première approche, est que `list_t` n'est plus un pointeur.
- En soi, ce n'est pas un problème, sauf qu'il serait plus agréable que `list_t` soit « naturellement » un pointeur.
- Il y a un moyen simple de réaliser cela : définir `list_t` non pas comme une structure décrite précédemment mais comme un tableau d'**une** (donc un **tableau de longueur 1**) telle structure. Du coup `list_t` redevient un pointeur (constant) vers le premier (et unique) élément.
- En résumé, on a :

```
typedef struct {  
    elt_t *first;  
    elt_t *last;  
    size_t len;  
} list_t[1]; /* tableau de une structure */
```

Une optimisation des listes génériques (2/2)

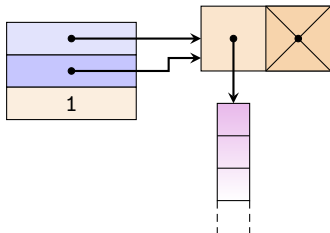
- Le problème avec cette première approche, est que `list_t` n'est plus un pointeur.
- En soi, ce n'est pas un problème, sauf qu'il serait plus agréable que `list_t` soit « naturellement » un pointeur.
- Il y a un moyen simple de réaliser cela : définir `list_t` non pas comme une structure décrite précédemment mais comme un tableau d'**une** (donc un **tableau de longueur 1**) telle structure. Du coup `list_t` redevient un pointeur (constant) vers le premier (et unique) élément.
- En résumé, on a :

```
typedef struct {  
    elt_t *first;  
    elt_t *last;  
    size_t len;  
} list_t[1]; /* tableau de une structure */
```

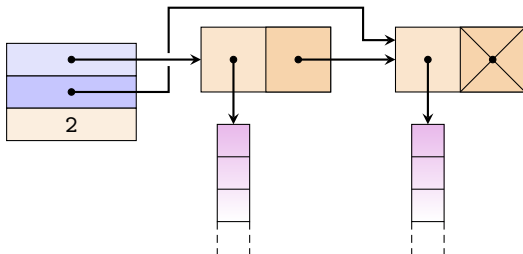
```
list_t l = LIST_EMPTY;
```

NULL
NULL
0

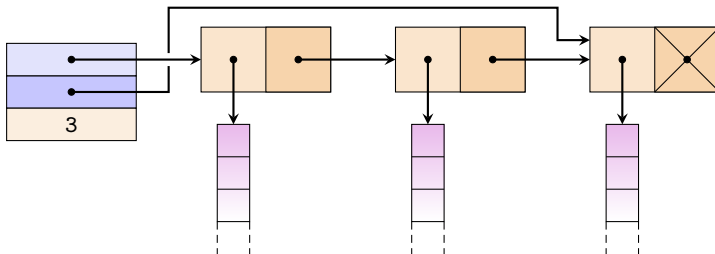

```
list_t l = LIST_EMPTY;  
list_insert(l, d1, c);
```



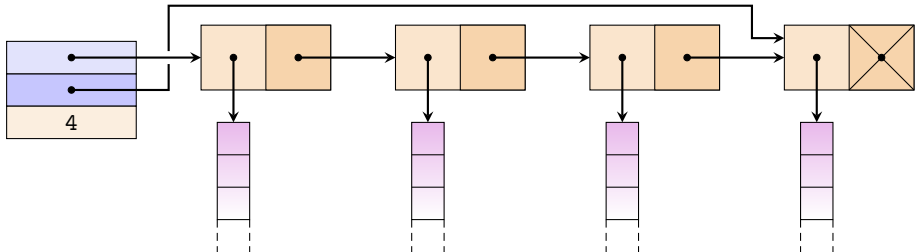
```
list_t l = LIST_EMPTY;  
  
list_insert(l, d1, c);  
list_insert(l, d2, c);
```



```
list_t l = LIST_EMPTY;  
  
list_insert(l, d1, c);  
list_insert(l, d2, c);  
list_insert(l, d3, c);
```



```
list_t l = LIST_EMPTY;  
  
list_insert(l, d1, c);  
list_insert(l, d2, c);  
list_insert(l, d3, c);  
list_insert(l, d4, c);
```



Interface : définitions des types

```
#ifndef LIST_H
#define LIST_H

typedef struct __elt { /* inchangé */
    void      *data;
    struct __elt *next;
} elt_t;

typedef struct { /* structure anonyme */
    elt_t *first; /* pointeur vers le premier élément */
    elt_t *last;  /* pointeur vers le dernier élément */
    size_t len;   /* longueur de la liste */
} list_t[1];     /* tableau de une structure */

#define LIST_EMPTY {{NULL, NULL, 0}} /* liste vide */

#endif /* LIST_H */
```

Interface : prototypes des fonctions

```
#ifndef LIST_H
#define LIST_H

typedef void *(*copy_t)(const void *);
typedef void (*free_t)(void *);
typedef void (*print_t)(const void *, FILE *);

inline size_t list_length(const list_t l)
{
    return l->len;
}

int list_insert(list_t, const void *, copy_t);
void list_free(list_t, free_t);
void list_print(const list_t, FILE *, print_t);

#endif /* LIST_H */
```

Le code de list_insert

```
int list_insert(list_t l, const void *d, copy_t c) {
    elt_t *e;

    e = malloc(sizeof(*e));
    if (e == NULL)
        return -1;
    e->data = c(d);
    if (e->data == NULL) {
        free(e);
        return -1;
    }
    e->next = l->first;
    l->first = e; /* e est le nouveau premier élément */
    if (e->next == NULL) /* la liste était vide */
        l->last = e; /* e est aussi le dernier élément */
    l->len++; /* la longueur de la liste est incrémentée */
    return 0;
}
```

De nouvelles fonctions (1/2)

À titre d'exercice, on peut chercher à enrichir cette bibliothèque en introduisant de nouvelles fonctions (dans ce qui suit, `list_t` est du type `elt_t *`) :

- `list_t list_append(list_t, const void *, copy_t)`; une fonction qui ajoute un élément à la fin de la liste.
- `list_t list_copy(const list_t, copy_t)`; une fonction réalisant la copie d'une liste.
- `int list_search(const list_t, const void *, comp_t)`; une fonction qui dit si des données sont présentes ou non dans une liste. Plus précisément, `list_search(l, d, c)` retourne 1 si la donnée `d` est présente dans la liste `l` et 0 sinon. L'argument `c` est un pointeur de type `int (*)(const void *, const void *)`, pointant vers une fonction telle que : `c(d1, d2)` vaut 0 si `d1 == d2`, est `< 0` si `d1 < d2` et est `> 0` si `d1 > d2`.

De nouvelles fonctions (1/2)

À titre d'exercice, on peut chercher à enrichir cette bibliothèque en introduisant de nouvelles fonctions (dans ce qui suit, `list_t` est du type `elt_t *`) :

- `list_t list_append(list_t, const void *, copy_t)`; une fonction qui ajoute un élément à la fin de la liste.
- `list_t list_copy(const list_t, copy_t)`; une fonction réalisant la copie d'une liste.
- `int list_search(const list_t, const void *, comp_t)`; une fonction qui dit si des données sont présentes ou non dans une liste. Plus précisément, `list_search(l, d, c)` retourne 1 si la donnée `d` est présente dans la liste `l` et 0 sinon. L'argument `c` est un pointeur de type `int (*)(const void *, const void *)`, pointant vers une fonction telle que : `c(d1, d2)` vaut 0 si `d1 == d2`, est `< 0` si `d1 < d2` et est `> 0` si `d1 > d2`.

De nouvelles fonctions (1/2)

À titre d'exercice, on peut chercher à enrichir cette bibliothèque en introduisant de nouvelles fonctions (dans ce qui suit, `list_t` est du type `elt_t *`) :

- `list_t list_append(list_t, const void *, copy_t)`; une fonction qui ajoute un élément à la fin de la liste.
- `list_t list_copy(const list_t, copy_t)`; une fonction réalisant la copie d'une liste.
- `int list_search(const list_t, const void *, comp_t)`; une fonction qui dit si des données sont présentes ou non dans une liste. Plus précisément, `list_search(l, d, c)` retourne 1 si la donnée `d` est présente dans la liste `l` et 0 sinon. L'argument `c` est un pointeur de type `int (*)(const void *, const void *)`, pointant vers une fonction telle que : `c(d1, d2)` vaut 0 si `d1 == d2`, est `< 0` si `d1 < d2` et est `> 0` si `d1 > d2`.

De nouvelles fonctions (1/2)

À titre d'exercice, on peut chercher à enrichir cette bibliothèque en introduisant de nouvelles fonctions (dans ce qui suit, `list_t` est du type `elt_t *`) :

- `list_t list_append(list_t, const void *, copy_t)`; une fonction qui ajoute un élément à la fin de la liste.
- `list_t list_copy(const list_t, copy_t)`; une fonction réalisant la copie d'une liste.
- `int list_search(const list_t, const void *, comp_t)`; une fonction qui dit si des données sont présentes ou non dans une liste. Plus précisément, `list_search(l, d, c)` retourne 1 si la donnée `d` est présente dans la liste `l` et 0 sinon. L'argument `c` est un pointeur de type `int (*)(const void *, const void *)`, pointant vers une fonction telle que : `c(d1, d2)` vaut 0 si `d1 == d2`, est `< 0` si `d1 < d2` et est `> 0` si `d1 > d2`.

De nouvelles fonctions (2/2)

- `list_t list_delete(list_t, int, free_t)`; une fonction qui supprime l'élément d'indice `n` (2^{ème} argument).
- `list_t list_concat(const list_t, const list_t)`; une fonction qui réalise la concaténation de deux listes.
- une extension de la fonction `list_insert` permettant d'insérer un élément n'importe où dans la liste. Elle aura pour prototype `list_t list_insert(list_t, int, const void *, copy_t)`; le nouvel élément étant inséré juste avant l'élément d'indice donné en deuxième argument.
- `list_t list_sort(list_t, comp_t)`; une fonction effectuant le tri d'une liste. Le 2^{ème} argument est du même type que le dernier argument de la fonction `list_search` vue précédemment.

De nouvelles fonctions (2/2)

- `list_t list_delete(list_t, int, free_t)`; une fonction qui supprime l'élément d'indice `n` (2^{ème} argument).
- `list_t list_concat(const list_t, const list_t)`; une fonction qui réalise la concaténation de deux listes.
- une extension de la fonction `list_insert` permettant d'insérer un élément n'importe où dans la liste. Elle aura pour prototype `list_t list_insert(list_t, int, const void *, copy_t)`; le nouvel élément étant inséré juste avant l'élément d'indice donné en deuxième argument.
- `list_t list_sort(list_t, comp_t)`; une fonction effectuant le tri d'une liste. Le 2^{ème} argument est du même type que le dernier argument de la fonction `list_search` vue précédemment.

De nouvelles fonctions (2/2)

- `list_t list_delete(list_t, int, free_t)`; une fonction qui supprime l'élément d'indice `n` (2^{ème} argument).
- `list_t list_concat(const list_t, const list_t)`; une fonction qui réalise la concaténation de deux listes.
- une extension de la fonction `list_insert` permettant d'insérer un élément n'importe où dans la liste. Elle aura pour prototype `list_t list_insert(list_t, int, const void *, copy_t)`; le nouvel élément étant inséré juste avant l'élément d'indice donné en deuxième argument.
- `list_t list_sort(list_t, comp_t)`; une fonction effectuant le tri d'une liste. Le 2^{ème} argument est du même type que le dernier argument de la fonction `list_search` vue précédemment.

De nouvelles fonctions (2/2)

- `list_t list_delete(list_t, int, free_t)`; une fonction qui supprime l'élément d'indice `n` (2^{ème} argument).
- `list_t list_concat(const list_t, const list_t)`; une fonction qui réalise la concaténation de deux listes.
- une extension de la fonction `list_insert` permettant d'insérer un élément n'importe où dans la liste. Elle aura pour prototype `list_t list_insert(list_t, int, const void *, copy_t)`; le nouvel élément étant inséré juste avant l'élément d'indice donné en deuxième argument.
- `list_t list_sort(list_t, comp_t)`; une fonction effectuant le tri d'une liste. Le 2^{ème} argument est du même type que le dernier argument de la fonction `list_search` vue précédemment.

La fonction `list_sort`

- De toutes les nouvelles fonctions, `list_sort` est la moins facile à implémenter et c'est pourquoi nous allons proposer une implémentation.
- Celle-ci va s'appuyer une primitive de la bibliothèque standard, la primitive `qsort` (*quick sort*). Cette primitive effectue le tri d'un tableau d'objets C, à priori quelconques.
- Un tableau étant une zone mémoire contiguë, la primitive `qsort` ne va pas être directement applicable aux listes. Il va donc falloir effectuer un travail préparatoire.
- Le prototype de primitive `qsort` est :

```
qsort(void *base, size_t nmemb, size_t size, comp_t compar);
```


où `base`, `nmemb`, `size` et `compar` sont respectivement un pointeur vers le premier élément du tableau, le nombre d'éléments du tableau, la taille d'un élément et la fonction de comparaison comme précédemment (`list_search`).

La fonction `list_sort`

- De toutes les nouvelles fonctions, `list_sort` est la moins facile à implémenter et c'est pourquoi nous allons proposer une implémentation.
- Celle-ci va s'appuyer une primitive de la bibliothèque standard, la primitive `qsort` (*quick sort*). Cette primitive effectue **le tri d'un tableau** d'objets C, à priori quelconques.
- Un tableau étant une zone mémoire contiguë, la primitive `qsort` ne va pas être directement applicable aux listes. Il va donc falloir effectuer un travail préparatoire.
- Le prototype de primitive `qsort` est :

```
qsort(void *base, size_t nmemb, size_t size, comp_t compar);
```


où `base`, `nmemb`, `size` et `compar` sont respectivement un pointeur vers le premier élément du tableau, le nombre d'éléments du tableau, la taille d'un élément et la fonction de comparaison comme précédemment (`list_search`).

La fonction `list_sort`

- De toutes les nouvelles fonctions, `list_sort` est la moins facile à implémenter et c'est pourquoi nous allons proposer une implémentation.
- Celle-ci va s'appuyer une primitive de la bibliothèque standard, la primitive `qsort` (*quick sort*). Cette primitive effectue **le tri d'un tableau** d'objets C, à priori quelconques.
- Un tableau étant une zone mémoire contiguë, la primitive `qsort` ne va pas être directement applicable aux listes. Il va donc falloir effectuer un travail préparatoire.

- Le prototype de primitive `qsort` est :

```
qsort(void *base, size_t nmemb, size_t size, comp_t compar);
```

où `base`, `nmemb`, `size` et `compar` sont respectivement un pointeur vers le premier élément du tableau, le nombre d'éléments du tableau, la taille d'un élément et la fonction de comparaison comme précédemment (`list_search`).

La fonction `list_sort`

- De toutes les nouvelles fonctions, `list_sort` est la moins facile à implémenter et c'est pourquoi nous allons proposer une implémentation.
- Celle-ci va s'appuyer une primitive de la bibliothèque standard, la primitive `qsort` (*quick sort*). Cette primitive effectue **le tri d'un tableau** d'objets C, à priori quelconques.
- Un tableau étant une zone mémoire contiguë, la primitive `qsort` ne va pas être directement applicable aux listes. Il va donc falloir effectuer un travail préparatoire.
- Le prototype de primitive `qsort` est :

```
qsort(void *base, size_t nmemb, size_t size, comp_t compar);
```

où `base`, `nmemb`, `size` et `compar` sont respectivement un pointeur vers le premier élément du tableau, le nombre d'éléments du tableau, la taille d'un élément et la fonction de comparaison comme précédemment (`list_search`).

qsort : tri d'un tableau d'entiers

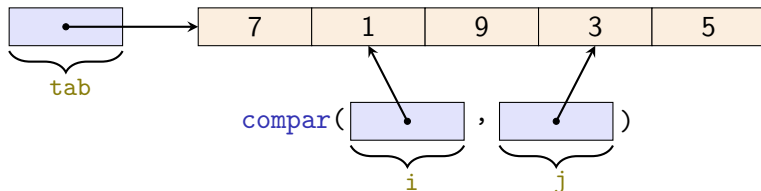
```
#include <stdio.h>
#include <stdlib.h>

static int
compar(const void *i, const void *j) {
    return *(int *) i - *(int *) j;
}

int main(void) {
    int tab[] = {7, 1, 3, 9, 5};
    size_t nmemb = sizeof(tab) / sizeof(int);

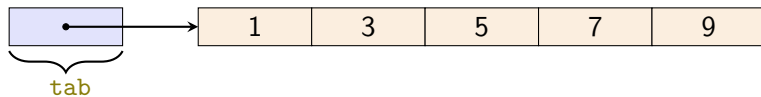
    qsort(tab, nmemb, sizeof(int), compar);
    for (int *ptr = tab; ptr - tab < nmemb; ptr++)
        printf("%d\n", *ptr);
    return 0;
}
```

```
int tab[] = {7, 1, 3, 9, 5};  
size_t nmemb = sizeof(tab) / sizeof(int);
```



Les arguments `i` et `j` sont des **pointeurs** (`const void *`) vers des éléments du tableau et non pas directement des éléments du tableau.

```
int tab[] = {7, 1, 3, 9, 5};  
size_t nmemb = sizeof(tab) / sizeof(int);  
qsort(tab, nmemb, sizeof(int), compar);
```



Le tableau est trié sur place

Le code de list_sort

```
#ifndef LIST_H
#define LIST_H

typedef int (*comp_t)(const void *, const void *);

list_t list_sort(list_t, comp_t);

#endif /* LIST_H */
```

Le code de list_sort

```
#ifndef LIST_H
#define LIST_H

typedef int (*comp_t)(const void *, const void *);

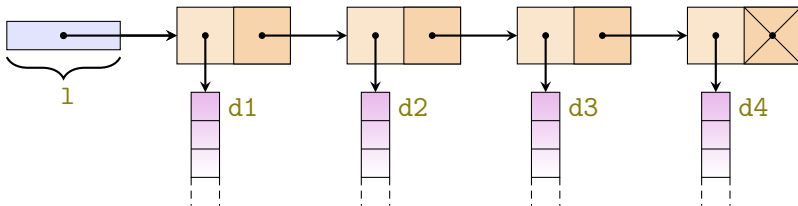
list_t list_sort(list_t, comp_t);

#endif /* LIST_H */

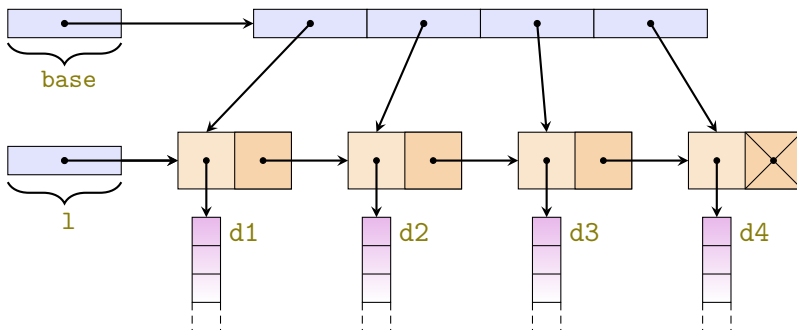
static comp_t comp_data; /* pas "thread safe" */

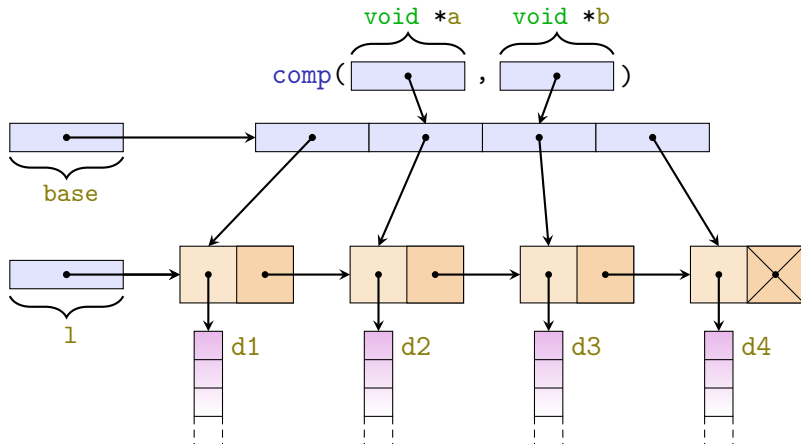
static int comp(const void *d1, const void *d2)
{
    return comp_data(
        (*(elt_t **) d1)->data, (*(elt_t **) d2)->data);
}
```

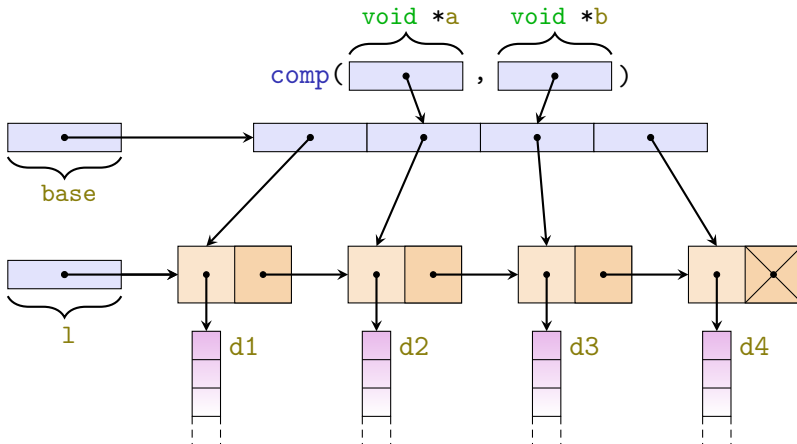

La liste à trier...



```
base = malloc(size * sizeof(*base));  
for (elt_t **pb = base, *pl = l; pl; pb++, pl = pl->next)  
    *pb = pl;
```

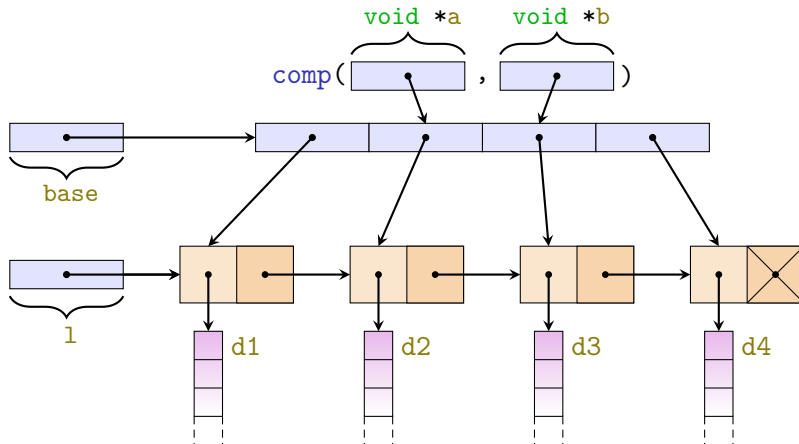




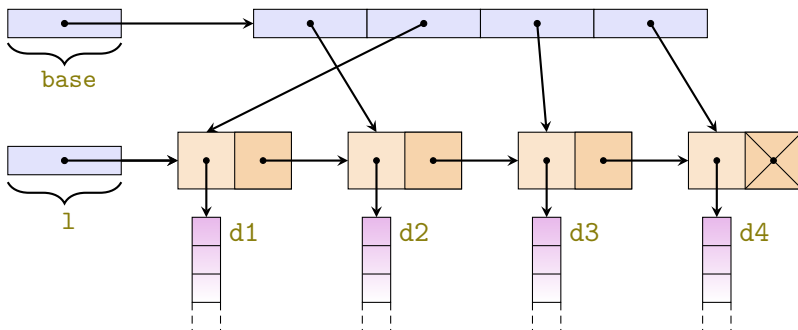
$$*(\text{elt_t } **)a \iff \text{base}[1] \iff l \rightarrow \text{next}$$


```
(* (elt_t **) a) -> data  $\iff$  d2  $\iff$  l->next->data
```

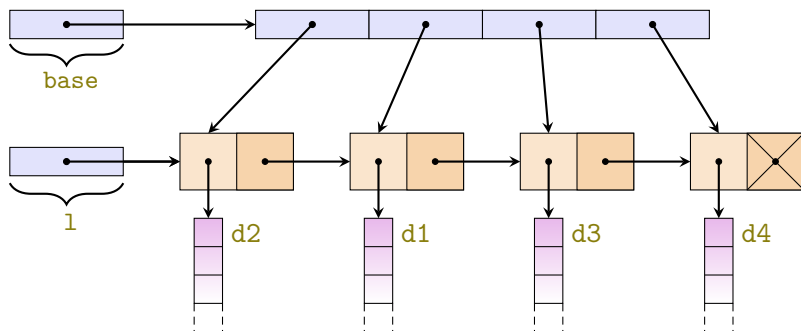
```
*(elt_t **) a  $\iff$  base[1]  $\iff$  l->next
```



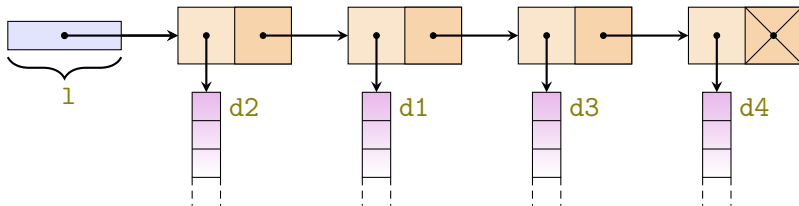
```
qsort(base, size, sizeof(*base), comp);
```



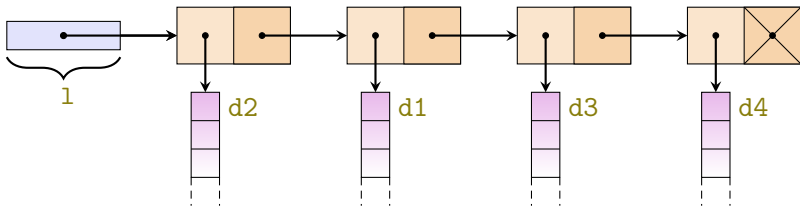
```
for (elt_t **pb = base; pb - base < size; pb++)  
    (*pb)->next = pb - base < size - 1 ? pb[1] : NULL;
```



```
free(base);
```



La liste triée !



Le code de list_sort

```
list_t list_sort(list_t l, comp_t c) {
    elt_t **base, *ret;
    size_t size = list_length(l);

    if (size < 2) /* liste déjà triée ! */
        return l;
    base = malloc(size * sizeof(*base));
    if (base == NULL)
        return l; /* on laisse la liste inchangée */
    for (elt_t **pb = base, *pl = l; pl; pb++, pl = pl->next)
        *pb = pl;
    comp_data = c;
    qsort(base, size, sizeof(*base), comp);
    for (elt_t **pb = base; pb - base < size; pb++)
        (*pb)->next = pb - base < size - 1 ? pb[1] : NULL;
    ret = *base;
    free(base);
    return ret;
}
```

Le code de list_sort – variante avec qsort_r

```
#ifndef LIST_H
#define LIST_H
#define _GNU_SOURCE /* pour qsort_r() */

typedef int (*comp_t)(const void *, const void *);

list_t list_sort(list_t l, comp_t);

#endif /* LIST_H */
```

Le code de list_sort – variante avec qsort_r

```
#ifndef LIST_H
#define LIST_H
#define _GNU_SOURCE /* pour qsort_r() */

typedef int (*comp_t)(const void *, const void *);

list_t list_sort(list_t l, comp_t);

#endif /* LIST_H */

static int comp(const void *d1, const void *d2, void *c)
{
    comp_t cmp = c;

    return cmp(
        (*(elt_t **) d1)->data, (*(elt_t **) d2)->data);
}
```

Le code de list_sort – variante avec qsort_r

```
list_t list_sort(list_t l, comp_t c) {
    elt_t **base, *ret;
    size_t size = list_length(l);

    if (size < 2) /* liste déjà triée ! */
        return l;
    base = malloc(size * sizeof(*base));
    if (base == NULL)
        return l; /* on laisse la liste inchangée */
    for (elt_t **pb = base, *pl = l; pl; pb++, pl = pl->next)
        *pb = pl;
    qsort_r(base, size, sizeof(*base), comp, c);
    for (elt_t **pb = base; pb - base < size; pb++)
        (*pb)->next = pb - base < size - 1 ? pb[1] : NULL;
    ret = *base;
    free(base);
    return ret;
}
```

Le tri d'une liste d'entiers : listint_sort

```
static int
comp_int(const void *d1, const void *d2)
{
    return *(int *) d1 - *(int *) d2;
}
```

Le tri d'une liste d'entiers : listint_sort

```
static int
comp_int(const void *d1, const void *d2)
{
    return *(int *) d1 - *(int *) d2;
}

listint_t
listint_sort(listint_t l)
{
    return (listint_t) list_sort((list_t) l, comp_int);
}
```

Une autre approche pour trier une liste

- Comment souvent en programmation, il n'y a pas une unique manière de résoudre un problème donné. On va proposer maintenant une approche quelque peu différente qui, si elle n'est pas significativement plus simple, ne nécessite pas d'avoir recours à `qsort_r` pour être « *thread safe* ».
- Dans le tableau `base`, au lieu de copier les **pointeurs vers les éléments** de la liste, on va maintenant copier les **pointeurs vers les données** de chaque élément.
- Dans ce cas, `base` aura pour type `void **` et non pas `elt_t **` comme précédemment.
- La fonction de comparaison `comp` (dernier argument de la primitive `qsort`) prenant en argument **deux pointeurs vers les éléments** du tableau `base`, il faudra modifier le code de la fonction `comp_int`.

Une autre approche pour trier une liste

- Comment souvent en programmation, il n'y a pas une unique manière de résoudre un problème donné. On va proposer maintenant une approche quelque peu différente qui, si elle n'est pas significativement plus simple, ne nécessite pas d'avoir recours à `qsort_r` pour être « *thread safe* ».
- Dans le tableau `base`, au lieu de copier les **pointeurs vers les éléments** de la liste, on va maintenant copier les **pointeurs vers les données** de chaque élément.
- Dans ce cas, `base` aura pour type `void **` et non pas `elt_t **` comme précédemment.
- La fonction de comparaison `comp` (dernier argument de la primitive `qsort`) prenant en argument **deux pointeurs vers les éléments** du tableau `base`, il faudra modifier le code de la fonction `comp_int`.

Une autre approche pour trier une liste

- Comment souvent en programmation, il n'y a pas une unique manière de résoudre un problème donné. On va proposer maintenant une approche quelque peu différente qui, si elle n'est pas significativement plus simple, ne nécessite pas d'avoir recours à `qsort_r` pour être « *thread safe* ».
- Dans le tableau `base`, au lieu de copier les **pointeurs vers les éléments** de la liste, on va maintenant copier les **pointeurs vers les données** de chaque élément.
- Dans ce cas, `base` aura pour type `void **` et non pas `elt_t **` comme précédemment.
- La fonction de comparaison `comp` (dernier argument de la primitive `qsort`) prenant en argument **deux pointeurs vers les éléments** du tableau `base`, il faudra modifier le code de la fonction `comp_int`.

Une autre approche pour trier une liste

- Comment souvent en programmation, il n'y a pas une unique manière de résoudre un problème donné. On va proposer maintenant une approche quelque peu différente qui, si elle n'est pas significativement plus simple, ne nécessite pas d'avoir recours à `qsort_r` pour être « *thread safe* ».
- Dans le tableau `base`, au lieu de copier les **pointeurs vers les éléments** de la liste, on va maintenant copier les **pointeurs vers les données** de chaque élément.
- Dans ce cas, `base` aura pour type `void **` et non pas `elt_t **` comme précédemment.
- La fonction de comparaison `comp` (dernier argument de la primitive `qsort`) prenant en argument **deux pointeurs vers les éléments** du tableau `base`, il faudra modifier le code de la fonction `comp_int`.

Le code modifié de list_sort

```
void list_sort(list_t l, comp_t c) {
    void **base;
    size_t size = list_length(l);

    if (size < 2)
        return;
    base = malloc(size * sizeof(*base));
    if (base == NULL)
        return;
    list_t pl = l;
    for (void **pb = base; pl; pb++, pl = pl->next)
        *pb = pl->data;
    qsort(base, size, sizeof(*base), c);
    pl = l;
    for (void **pb = base; pl; pb++, pl = pl->next)
        pl->data = *pb;
    free(base);
}
```

Le code modifié de : listint_sort

```
static int
comp_int(const void *d1, const void *d2)
{
    return **(int **) d1 - **(int **) d2;
}
```

Le code modifié de : listint_sort

```
static int
comp_int(const void *d1, const void *d2)
{
    return **((int **) d1) - **((int **) d2);
}

void
listint_sort(listint_t l)
{
    list_sort((list_t) l, comp_int);
}
```

Présentation

- **make** est un outil permettant de construire automatiquement des programmes exécutables et des bibliothèques à partir de codes source en analysant des fichiers généralement appelés **makefile** ou **Makefile** qui spécifient comment dériver le programme cible.
- C'est un logiciel assez ancien puisqu'apparu en 1976. Il existe trois implémentations principales : *BSD make*, *GNU make* et *Microsoft nmake*.
- La syntaxe des fichiers **makefile** ou **Makefile** diffère notablement d'une implémentation à l'autre. On ne s'intéressera ici qu'à l'implémentation *GNU make*.
- La commande **make** est invoquée ainsi :

```
$ make [cible1 [cible2]...]
```

Exécutée sans arguments, **make** construit la première cible apparaissant dans le fichier **makefile**, cible traditionnellement nommée **all**.

Présentation

- **make** est un outil permettant de construire automatiquement des programmes exécutables et des bibliothèques à partir de codes source en analysant des fichiers généralement appelés **makefile** ou **Makefile** qui spécifient comment dériver le programme cible.
- C'est un logiciel assez ancien puisqu'apparu en 1976. Il existe trois implémentations principales : *BSD make*, *GNU make* et *Microsoft nmake*.
- La syntaxe des fichiers **makefile** ou **Makefile** diffère notablement d'une implémentation à l'autre. On ne s'intéressera ici qu'à l'implémentation *GNU make*.
- La commande **make** est invoquée ainsi :

```
$ make [cible1 [cible2]...]
```

Exécutée sans arguments, **make** construit la première cible apparaissant dans le fichier **makefile**, cible traditionnellement nommée **all**.

Présentation

- **make** est un outil permettant de construire automatiquement des programmes exécutables et des bibliothèques à partir de codes source en analysant des fichiers généralement appelés **makefile** ou **Makefile** qui spécifient comment dériver le programme cible.
- C'est un logiciel assez ancien puisqu'apparu en 1976. Il existe trois implémentations principales : *BSD make*, *GNU make* et *Microsoft nmake*.
- La syntaxe des fichiers **makefile** ou **Makefile** diffère notablement d'une implémentation à l'autre. On ne s'intéressera ici qu'à l'implémentation *GNU make*.
- La commande **make** est invoquée ainsi :

```
$ make [cible1 [cible2]...]
```

Exécutée sans arguments, **make** construit la première cible apparaissant dans le fichier **makefile**, cible traditionnellement nommée **all**.

Présentation

- **make** est un outil permettant de construire automatiquement des programmes exécutables et des bibliothèques à partir de codes source en analysant des fichiers généralement appelés **makefile** ou **Makefile** qui spécifient comment dériver le programme cible.
- C'est un logiciel assez ancien puisqu'apparu en 1976. Il existe trois implémentations principales : *BSD make*, *GNU make* et *Microsoft nmake*.
- La syntaxe des fichiers **makefile** ou **Makefile** diffère notablement d'une implémentation à l'autre. On ne s'intéressera ici qu'à l'implémentation *GNU make*.
- La commande **make** est invoquée ainsi :

```
$ make [cible1 [cible2]...]
```

Exécutée sans arguments, **make** construit la première cible apparaissant dans le fichier **makefile**, cible traditionnellement nommée **all**.

Un peu de syntaxe

- Un fichier **makefile** est constitué d'une liste de *cibles* ayant le format suivant :

```
cible: prérequis_1 prérequis_2 ... prérequis_N
    tab→ command_1
    tab→ command_2
    ...
    tab→ command_N
```

Attention : tabulations \neq espaces !




- La première commande peut apparaître sur la même ligne que les prérequis, après ceux-ci et séparée par un point-virgule :

```
cible: prérequis_1 ... prérequis_N; command_1
```

- make** décide si une cible doit être régénérée en **comparant les temps de modification** des fichiers. Par exemple, si **prérequis_2** est plus récent que **cible**, alors **cible** sera reconstruite.

Un peu de syntaxe

- Un fichier **makefile** est constitué d'une liste de *cibles* ayant le format suivant :

```
cible: prérequis_1 prérequis_2 ... prérequis_N
     command_1
     command_2
    ...
     command_N
```

Attention : tabulations \neq espaces !

- La première commande peut apparaître sur la même ligne que les prérequis, après ceux-ci et séparée par un point-virgule :

```
cible: prérequis_1 ... prérequis_N; command_1
```

- make** décide si une cible doit être régénérée en **comparant les temps de modification** des fichiers. Par exemple, si **prérequis_2** est plus récent que **cible**, alors **cible** sera reconstruite.

Un peu de syntaxe

- Un fichier **makefile** est constitué d'une liste de *cibles* ayant le format suivant :

```
cible: prérequis_1 prérequis_2 ... prérequis_N
    tab→ command_1
    tab→ command_2
    ...
    tab→ command_N
```

Attention : tabulations \neq espaces !

- La première commande peut apparaître sur la même ligne que les prérequis, après ceux-ci et séparée par un point-virgule :

```
cible: prérequis_1 ... prérequis_N; command_1
```

- make** décide si une cible doit être régénérée en **comparant les temps de modification** des fichiers. Par exemple, si **prérequis_2** est plus récent que **cible**, alors **cible** sera reconstruite.

Un premier exemple

Supposons que l'on ait un logiciel nommé `prog` (exécutable) et composé de deux fichiers source `src.c`, `prog.c` et d'un fichier d'en-tête `src.h`.

Un fichier `makefile` pour gérer ce logiciel pourrait être :

```
prog: src.o prog.o # 'prog' est la cible par défaut
    gcc src.o prog.o -o prog

prog.o: src.h prog.c
    gcc -Wall -Wpointer-arith -c prog.c


src.o: src.h src.c
    gcc -Wall -Wpointer-arith -c src.c
```


Ceci n'est qu'une **ébauche** de fichier `makefile`. En effet, d'une part rien n'est paramétré (absence de *variables*) et d'autre part ce fichier `makefile` deviendra difficile à gérer si le logiciel devient conséquent en taille (dizaines, voire centaines de fichiers source/en-tête).


Un premier exemple

Supposons que l'on ait un logiciel nommé `prog` (exécutable) et composé de deux fichiers source `src.c`, `prog.c` et d'un fichier d'en-tête `src.h`.

Un fichier **makefile** pour gérer ce logiciel pourrait être :

```
prog: src.o prog.o # 'prog' est la cible par défaut
     gcc src.o prog.o -o prog

prog.o: src.h prog.c
     gcc -Wall -Wpointer-arith -c prog.c


src.o: src.h src.c
     gcc -Wall -Wpointer-arith -c src.c
```


Ceci n'est qu'une *ébauche* de fichier **makefile**. En effet, d'une part rien n'est paramétré (absence de *variables*) et d'autre part ce fichier **makefile** deviendra difficile à gérer si le logiciel devient conséquent en taille (dizaines, voire centaines de fichiers source/en-tête).


Un premier exemple

Supposons que l'on ait un logiciel nommé `prog` (exécutable) et composé de deux fichiers source `src.c`, `prog.c` et d'un fichier d'en-tête `src.h`.

Un fichier **makefile** pour gérer ce logiciel pourrait être :

```
prog: src.o prog.o # 'prog' est la cible par défaut
     gcc src.o prog.o -o prog

prog.o: src.h prog.c
     gcc -Wall -Wpointer-arith -c prog.c

src.o: src.h src.c
     gcc -Wall -Wpointer-arith -c src.c
```

Ceci n'est qu'une **ébauche** de fichier **makefile**. En effet, d'une part rien n'est paramétré (absence de *variables*) et d'autre part ce fichier **makefile** deviendra difficile à gérer si le logiciel devient conséquent en taille (dizaines, voire centaines de fichiers source/en-tête).

Compléments sur la syntaxe

- Le caractère de continuation de ligne ‘\’ permet de couper les lignes trop longues :

```
cible: prérequis_1 prérequis_2 prérequis_3 \  
  tab prérequis_4 prérequis_5 # ‘tab’ non interprété ici  
  tab command argument_1 argument_2 argument_3 \  
  tab tab argument_4
```

- Une cible peut être répartie en plusieurs entrées de même nom :

```
cible: prérequis_1 prérequis_2 ... prérequis_N  
  
cible:  
  tab commande_1  
  ...  
  tab commande_N
```

Compléments sur la syntaxe

- Le caractère de continuation de ligne ‘\’ permet de couper les lignes trop longues :

```
cible: prérequis_1 prérequis_2 prérequis_3 \  
  tab→ prérequis_4 prérequis_5 # ‘tab’ non interprété ici  
  tab→ command argument_1 argument_2 argument_3 \  
  tab→ tab→ argument_4
```

- Une cible peut être répartie en plusieurs entrées de même nom :

```
cible: prérequis_1 prérequis_2 ... prérequis_N  
  
cible:  
  tab→ commande_1  
  ...  
  tab→ commande_N
```

Préfixes des commandes de cibles

- Toute commande de cible peut être préfixée par l'un des caractères suivants : `-`, `+` ou `@`.
- `-cmd` : les éventuelles erreurs générées lors de l'exécution de `cmd` sont ignorées et `make` continuera normalement son exécution.
- `+cmd` : la commande `cmd` sera exécutée même si `make` a été lancée avec l'option `-n` (auquel cas toutes les commandes sont imprimées mais ne sont pas exécutées).
- `@cmd` : la commande `cmd` sera exécutée mais ne sera pas imprimée (elle est imprimée par défaut). Par exemple, si l'on veut imprimer un message avec la commande `echo` :

```
hello:
    @echo "hello_world!"
```

- On peut également combiner les préfixes :

```
hello:
    @+echo "hello_world!"
```

Préfixes des commandes de cibles

- Toute commande de cible peut être préfixée par l'un des caractères suivants : `-`, `+` ou `@`.
- `-cmd` : les éventuelles **erreurs** générées lors de l'exécution de `cmd` sont **ignorées** et `make` continuera normalement son exécution.
- `+cmd` : la commande `cmd` sera exécutée même si `make` a été lancée avec l'option `-n` (auquel cas toutes les commandes sont imprimées mais ne sont pas exécutées).
- `@cmd` : la commande `cmd` sera exécutée mais ne sera pas imprimée (elle est imprimée par défaut). Par exemple, si l'on veut imprimer un message avec la commande `echo` :

```
hello:
    @echo "hello_world!"
```

- On peut également combiner les préfixes :

```
hello:
    @+echo "hello_world!"
```

Préfixes des commandes de cibles

- Toute commande de cible peut être préfixée par l'un des caractères suivants : `-`, `+` ou `@`.
- `-cmd` : les éventuelles **erreurs** générées lors de l'exécution de `cmd` sont **ignorées** et `make` continuera normalement son exécution.
- `+cmd` : la commande `cmd` sera exécutée même si `make` a été lancée avec l'option `-n` (auquel cas toutes les commandes sont imprimées mais ne sont pas exécutées).
- `@cmd` : la commande `cmd` sera exécutée mais ne sera pas imprimée (elle est imprimée par défaut). Par exemple, si l'on veut imprimer un message avec la commande `echo` :

```
hello:
    @echo "hello_world!"
```

- On peut également combiner les préfixes :

```
hello:
    @+echo "hello_world!"
```

Préfixes des commandes de cibles

- Toute commande de cible peut être préfixée par l'un des caractères suivants : `-`, `+` ou `@`.
- `-cmd` : les éventuelles **erreurs** générées lors de l'exécution de `cmd` sont **ignorées** et `make` continuera normalement son exécution.
- `+cmd` : la commande `cmd` sera exécutée même si `make` a été lancée avec l'option `-n` (auquel cas toutes les commandes sont imprimées mais ne sont pas exécutées).
- `@cmd` : la commande `cmd` sera exécutée mais ne sera pas imprimée (elle est imprimée par défaut). Par exemple, si l'on veut imprimer un message avec la commande `echo` :

```
hello:
    @echo "hello_world!"
```

- On peut également combiner les préfixes :

```
hello:
    @+echo "hello_world!"
```

Préfixes des commandes de cibles

- Toute commande de cible peut être préfixée par l'un des caractères suivants : `-`, `+` ou `@`.
- `-cmd` : les éventuelles **erreurs** générées lors de l'exécution de `cmd` sont **ignorées** et `make` continuera normalement son exécution.
- `+cmd` : la commande `cmd` sera exécutée même si `make` a été lancée avec l'option `-n` (auquel cas toutes les commandes sont imprimées mais ne sont pas exécutées).
- `@cmd` : la commande `cmd` sera exécutée mais ne sera pas imprimée (elle est imprimée par défaut). Par exemple, si l'on veut imprimer un message avec la commande `echo` :

```
hello:
    @echo "hello_world!"
```

- On peut également combiner les préfixes :

```
hello:
    @+echo "hello_world!"
```

Les variables

(1/7)

- Elles se définissent comme suit :

```
VARIABLE = expression
```

et se référencent ainsi : `$(VARIABLE)`

- Par exemple :

```
PKG_NAME = test
PKG_VERSION = 0.1
DIST_NAME = $(PKG_NAME)-$(PKG_VERS) # test-0.1
```

ou si l'on développe en C :

```
CC = gcc
CPPFLAGS = -I .
CFLAGS = -Wall -Wpointer-arith # -O3
```

- On peut aussi définir une variable en évaluant une commande *shell* :

```
SRCS = `ls *.c` # backquote
```


Les variables

(1/7)

- Elles se définissent comme suit :

```
VARIABLE = expression
```

et se référencent ainsi : `$(VARIABLE)`

- Par exemple :

```
PKG_NAME = test
PKG_VERSION = 0.1
DIST_NAME = $(PKG_NAME)-$(PKG_VERSION) # test-0.1
```

ou si l'on développe en C :

```
CC = gcc
CPPFLAGS = -I .
CFLAGS = -Wall -Wpointer-arith # -O3
```

- On peut aussi définir une variable en évaluant une commande *shell* :

```
SRCS = `ls *.c` # backquote
```

Les variables

(1/7)

- Elles se définissent comme suit :

```
VARIABLE = expression
```

et se référencent ainsi : `$(VARIABLE)`

- Par exemple :

```
PKG_NAME = test
PKG_VERSION = 0.1
DIST_NAME = $(PKG_NAME)-$(PKG_VERSION) # test-0.1
```

ou si l'on développe en C :

```
CC = gcc
CPPFLAGS = -I .
CFLAGS = -Wall -Wpointer-arith # -O3
```

- On peut aussi définir une variable en évaluant une commande *shell* :

```
SRCS = `ls *.c` # backquote
```


Les variables – le premier exemple revisité


(2/7)


```
# définition des variables
PROG = prog # nom du logiciel

CC = gcc
CPPFLAGS = -I .
CFLAGS = -Wall -Wpointer-arith # -O3

OBJS = src.o $(PROG).o

# les cibles réécrites avec les variables
$(PROG): $(OBJS)
     $(CC) $(OBJS) -o $(PROG)

prog.o: src.h $(PROG).c
     $(CC) $(CPPFLAGS) $(CFLAGS) -c $(PROG).c

src.o: src.h src.c
     $(CC) $(CPPFLAGS) $(CFLAGS) -c src.c
```


Les variables – le premier exemple revisité


(2/7)


```
# définition des variables
PROG = prog # nom du logiciel

CC = gcc
CPPFLAGS = -I .
CFLAGS = -Wall -Wpointer-arith # -O3

OBJS = src.o $(PROG).o

# les cibles réécrites avec les variables
$(PROG): $(OBJS)
     $(CC) $(OBJS) -o $(PROG)

prog.o: src.h $(PROG).c
     $(CC) $(CPPFLAGS) $(CFLAGS) -c $(PROG).c

src.o: src.h src.c
     $(CC) $(CPPFLAGS) $(CFLAGS) -c src.c
```

Les variables

(3/7)

- Les variables définies précédemment sont des variables qui seront évaluées de manière **paresseuse** , c'est à dire qu'elles ne seront évaluées que lorsque nécessaire. Ainsi dans la définition suivante,

```
FILES = $(SRCS) $(HDRS)
```

il n'est pas nécessaire que les variables **SRCS** et **HDRS** aient été définies préalablement.

- Il existe un autre type de variables, les variables à **évaluation immédiate**. Elles se définissent ainsi :

```
VARIABLE := expression # ':= ' au lieu de '='
```

Par exemple, si l'on pose :

```
VAR_1 := $(VAR_0)
```

et que la variable **VAR_0** n'a pas été définie auparavant, alors la variable **VAR_1** est la chaîne vide.

Les variables

(3/7)

- Les variables définies précédemment sont des variables qui seront évaluées de manière **paresseuse** , c'est à dire qu'elles ne seront évaluées que lorsque nécessaire. Ainsi dans la définition suivante,

```
FILES = $(SRCS) $(HDRS)
```

il n'est pas nécessaire que les variables **SRCS** et **HDRS** aient été définies préalablement.

- Il existe un autre type de variables, les variables à **évaluation immédiate**. Elles se définissent ainsi :

```
VARIABLE := expression # ':= ' au lieu de '='
```

Par exemple, si l'on pose :

```
VAR_1 := $(VAR_0)
```

et que la variable **VAR_0** n'a pas été définie auparavant, alors la variable **VAR_1** est la chaîne vide.

Les variables

(4/7)

- Les variables **conditionnelles** :

```
VARIABLE ?= expression
```

La variable **VARIABLE** ne sera effectivement définie que si elle ne l'a pas déjà été. Utile pour la gestion des *variables d'environnement* (automatiquement chargées par **make**), comme la variable **HOME**.

- L'opérateur d'affectation **'+='** :

```
VARIABLE += expression
```

Par exemple :

```
CFLAGS = -Wall -Wpointer-arith  
CFLAGS += -O3 # -Wall -Wpointer-arith -O3
```

Évaluation : le membre droit est évalué immédiatement si le membre gauche est à évaluation immédiate.

Les variables

(4/7)

- Les variables **conditionnelles** :

```
VARIABLE ?= expression
```

La variable **VARIABLE** ne sera effectivement définie que si elle ne l'a pas déjà été. Utile pour la gestion des *variables d'environnement* (automatiquement chargées par **make**), comme la variable **HOME**.

- L'opérateur d'affectation **'+='** :

```
VARIABLE += expression
```

Par exemple :

```
CFLAGS = -Wall -Wpointer-arith  
CFLAGS += -O3 # -Wall -Wpointer-arith -O3
```

Évaluation : le membre droit est évalué immédiatement si le membre gauche est à évaluation immédiate.

Les variables

(5/7)

- La commande `make` permet de définir des variables à la volée :

```
$ make VAR_1=value_1 ... VAR_1=value_N
```

- Les valeurs des variables ainsi passées en option écrasent les valeurs des variables de même nom définies dans le fichier `Makefile` :

```
# fichier Makefile
DEBUG_LEVEL = 1 # niveau de debug par défaut
```

Pour changer le niveau de debug par défaut :

```
$ make DEBUG_LEVEL=2
```

- Attention : le caractère '\$' est **toujours** interprété, même au sein des chaînes de caractères :

```
test: ; @echo "cd_$HOME" # retourne "cd OME"
```

Pour échapper le caractère '\$', il faut le doubler :

```
test: ; @echo "cd_$$HOME" # retourne "cd /home/ylg"
```

Les variables

(5/7)

- La commande `make` permet de définir des variables à la volée :

```
$ make VAR_1=value_1 ... VAR_1=value_N
```

- Les valeurs des variables ainsi passées en option écrasent les valeurs des variables de même nom définies dans le fichier **Makefile** :

```
# fichier Makefile
DEBUG_LEVEL = 1 # niveau de debug par défaut
```

Pour changer le niveau de debug par défaut :

```
$ make DEBUG_LEVEL=2
```

- Attention : le caractère '\$' est toujours interprété, même au sein des chaînes de caractères :

```
test: ; @echo "cd_$HOME" # retourne "cd OME"
```

Pour échapper le caractère '\$', il faut le doubler :

```
test: ; @echo "cd_$$HOME" # retourne "cd /home/ylg"
```

Les variables

(5/7)

- La commande `make` permet de définir des variables à la volée :

```
$ make VAR_1=value_1 ... VAR_1=value_N
```

- Les valeurs des variables ainsi passées en option écrasent les valeurs des variables de même nom définies dans le fichier **Makefile** :

```
# fichier Makefile
DEBUG_LEVEL = 1 # niveau de debug par défaut
```

Pour changer le niveau de debug par défaut :

```
$ make DEBUG_LEVEL=2
```

- Attention** : le caractère '\$' est **toujours** interprété, même au sein des chaînes de caractères :

```
test: ; @echo "cd_$HOME" # retourne "cd OME"
```

Pour échapper le caractère '\$', il faut le doubler :

```
test: ; @echo "cd_$$HOME" # retourne "cd /home/ylg"
```

Les variables

(6/7)

- Les variables **multilignes** :

```
define VARIABLE = # signe '=' facultatif
expression_1
...
expression_N
endef
```

- Un exemple de regroupement de commandes :

```
define PY_VERS =
@echo -n "Python_ version:_"
@python -V
endef
show_version: ; $(PY_VERS)
```

- On peut écrire de manière équivalente :

```
PY_VERS = @echo -n "Python_ version:_" ; python -V
show_version: ; $(PY_VERS)
```

Les variables

(6/7)

- Les variables **multilignes** :

```
define VARIABLE = # signe '=' facultatif
expression_1
...
expression_N
endef
```

- Un exemple de regroupement de commandes :

```
define PY_VERS =
@echo -n "Python_ version:_"
@python -V
endef
show_version: ; $(PY_VERS)
```

- On peut écrire de manière équivalente :

```
PY_VERS = @echo -n "Python_ version:_" ; python -V
show_version: ; $(PY_VERS)
```

Les variables

(6/7)

- Les variables **multilignes** :

```
define VARIABLE = # signe '=' facultatif
expression_1
...
expression_N
endef
```

- Un exemple de regroupement de commandes :

```
define PY_VERS =
@echo -n "Python_ version:_"
@python -V
endef
show_version: ; $(PY_VERS)
```

- On peut écrire de manière équivalente :

```
PY_VERS = @echo -n "Python_ version:_" ; python -V
show_version: ; $(PY_VERS)
```

Les variables

(7/7)

Le logiciel `make` prédéfinit un certain nombre de variables. En voici quelques unes parmi les plus courantes :

- `$$` \Rightarrow nom de la cible :

```
prog:  $(OBSJ); $(CC) $(OBSJ) -o $$ # '$$' vaut 'prog'
```

- `$<` \Rightarrow nom du premier prérequis :

```
prog: pr_1 pr_2 ... pr_N; ... # '$<' vaut 'pr_1'
```

- `$^` \Rightarrow tous les prérequis séparés par un espace :

```
prog: pr_1 ... pr_N; ... # '$^' vaut 'pr_1_ ... _pr_N'
```

- `$?` \Rightarrow même chose que `$^`, excepté que seuls les prérequis **plus récents** que la cible sont retenus.

- `$*` \Rightarrow nom de la cible sans le suffixe :

```
lib.a: ; @echo $* # '$*' vaut 'lib'
```

Les variables

(7/7)

Le logiciel `make` prédéfinit un certain nombre de variables. En voici quelques unes parmi les plus courantes :

- `$@` \Rightarrow nom de la cible :

```
prog: $(OBSJ); $(CC) $(OBSJ) -o $@ # '$@' vaut 'prog'
```

- `$<` \Rightarrow nom du premier prérequis :

```
prog: pr_1 pr_2 ... pr_N; ... # '$<' vaut 'pr_1'
```

- `$^` \Rightarrow tous les prérequis séparés par un espace :

```
prog: pr_1 ... pr_N; ... # '$^' vaut 'pr_1 pr_2 ... pr_N'
```

- `$?` \Rightarrow même chose que `$^`, excepté que seuls les prérequis **plus récents** que la cible sont retenus.

- `$*` \Rightarrow nom de la cible sans le suffixe :

```
lib.a: ; @echo $* # '$*' vaut 'lib'
```


Les variables

(7/7)

Le logiciel `make` prédéfinit un certain nombre de variables. En voici quelques unes parmi les plus courantes :

- `$@` \Rightarrow nom de la cible :

```
prog: $(OBSJ); $(CC) $(OBSJ) -o $@ # '$@' vaut 'prog'
```

- `$<` \Rightarrow nom du premier prérequis :

```
prog: pr_1 pr_2 ... pr_N; ... # '$<' vaut 'pr_1'
```

- `$^` \Rightarrow tous les prérequis séparés par un espace :

```
prog: pr_1 ... pr_N; ... # '$^' vaut 'pr_1 pr_2 ... pr_N'
```

- `$?` \Rightarrow même chose que `$^`, excepté que seuls les prérequis **plus récents** que la cible sont retenus.

- `$*` \Rightarrow nom de la cible sans le suffixe :

```
lib.a: ; @echo $* # '$*' vaut 'lib'
```

Les variables

(7/7)

Le logiciel `make` prédéfinit un certain nombre de variables. En voici quelques unes parmi les plus courantes :

- `$@` \Rightarrow nom de la cible :

```
prog:  $(OBSJ); $(CC) $(OBSJ) -o $@ # '$@' vaut 'prog'
```

- `$<` \Rightarrow nom du premier prérequis :

```
prog: pr_1 pr_2 ... pr_N; ... # '$<' vaut 'pr_1'
```

- `$^` \Rightarrow tous les prérequis séparés par un espace :

```
prog: pr_1 ... pr_N; ... # '$^' vaut 'pr_1 pr_2 ... pr_N'
```

- `$?` \Rightarrow même chose que `$^`, excepté que seuls les prérequis **plus récents** que la cible sont retenus.

- `$*` \Rightarrow nom de la cible sans le suffixe :

```
lib.a: ; @echo $* # '$*' vaut 'lib'
```

Les variables

(7/7)

Le logiciel `make` prédéfinit un certain nombre de variables. En voici quelques unes parmi les plus courantes :

- `$@` \Rightarrow nom de la cible :

```
prog: $(OBSJ); $(CC) $(OBSJ) -o $@ # '$@' vaut 'prog'
```

- `$<` \Rightarrow nom du premier prérequis :

```
prog: pr_1 pr_2 ... pr_N; ... # '$<' vaut 'pr_1'
```

- `$^` \Rightarrow tous les prérequis séparés par un espace :

```
prog: pr_1 ... pr_N; ... # '$^' vaut 'pr_1 pr_2 ... pr_N'
```

- `$?` \Rightarrow même chose que `$^`, excepté que seuls les prérequis **plus récents** que la cible sont retenus.

- `$*` \Rightarrow nom de la cible sans le suffixe :

```
lib.a: ; @echo $* # '$*' vaut 'lib'
```

Les variables

(7/7)

Le logiciel `make` prédéfinit un certain nombre de variables. En voici quelques unes parmi les plus courantes :

- `$@` \Rightarrow nom de la cible :

```
prog: $(OBSJ); $(CC) $(OBSJ) -o $@ # '$@' vaut 'prog'
```

- `$<` \Rightarrow nom du premier prérequis :

```
prog: pr_1 pr_2 ... pr_N; ... # '$<' vaut 'pr_1'
```

- `$^` \Rightarrow tous les prérequis séparés par un espace :

```
prog: pr_1 ... pr_N; ... # '$^' vaut 'pr_1 pr_2 ... pr_N'
```

- `$?` \Rightarrow même chose que `$^`, excepté que seuls les prérequis **plus récents** que la cible sont retenus.

- `$*` \Rightarrow nom de la cible sans le suffixe :

```
lib.a: ; @echo $* # '$*' vaut 'lib'
```

Les variables (GNU make)

(1/2)

- L'implémentation GNU de `make` propose (beaucoup) d'autres moyens de définir/modifier des variables.
- La fonction `wildcard`. Par exemple, pour définir une variable contenant tous les noms de fichiers du catalogue courant ayant une extension `.c` (analogue à `'ls *.c'`) :

```
SRCS = $(wildcard *.c)
```

Attention : si l'on pose `SRCS = *.c`, `SRCS` est alors la chaîne `"*.c"`.

- Substitution. Si `SRCS` est définie comme précédemment :

```
OBJS = $(SRCS:.c=.o)
```

alors la valeur de `OBJS` est la valeur `SRCS` dans laquelle on a remplacé toutes les occurrences de `.c` par `.o`.

- Et encore bien d'autres fonctions comme `$(subst...)`, `$(filter...)`, `$(patsubst...)`, `$(filter-out...)`, et cætera.

Les variables (GNU make)

(1/2)

- L'implémentation GNU de `make` propose (beaucoup) d'autres moyens de définir/modifier des variables.
- La fonction `wildcard`. Par exemple, pour définir une variable contenant tous les noms de fichiers du catalogue courant ayant une extension `.c` (analogue à `'ls *.c'`) :

```
SRCS = $(wildcard *.c)
```

Attention : si l'on pose `SRCS = *.c`, `SRCS` est alors la chaîne `"*.c"`.

- Substitution. Si `SRCS` est définie comme précédemment :

```
OBJS = $(SRCS:.c=.o)
```

alors la valeur de `OBJS` est la valeur `SRCS` dans laquelle on a remplacé toutes les occurrences de `.c` par `.o`.

- Et encore bien d'autres fonctions comme `$(subst...)`, `$(filter...)`, `$(patsubst...)`, `$(filter-out...)`, et cætera.

Les variables (GNU make)

(1/2)

- L'implémentation GNU de `make` propose (beaucoup) d'autres moyens de définir/modifier des variables.
- La fonction `wildcard`. Par exemple, pour définir une variable contenant tous les noms de fichiers du catalogue courant ayant une extension `.c` (analogue à `'ls *.c'`) :

```
SRCS = $(wildcard *.c)
```

Attention : si l'on pose `SRCS = *.c`, `SRCS` est alors la chaîne `"*.c"`.

- Substitution. Si `SRCS` est définie comme précédemment :

```
OBJS = $(SRCS:.c=.o)
```

alors la valeur de `OBJS` est la valeur `SRCS` dans laquelle on a remplacé toutes les occurrences de `.c` par `.o`.

- Et encore bien d'autres fonctions comme `$(subst...)`, `$(filter...)`, `$(patsubst...)`, `$(filter-out...)`, et cætera.

Les variables (GNU make)

(1/2)

- L'implémentation GNU de `make` propose (beaucoup) d'autres moyens de définir/modifier des variables.
- La fonction `wildcard`. Par exemple, pour définir une variable contenant tous les noms de fichiers du catalogue courant ayant une extension `.c` (analogue à `'ls *.c'`) :

```
SRCS = $(wildcard *.c)
```

Attention : si l'on pose `SRCS = *.c`, `SRCS` est alors la chaîne `"*.c"`.

- Substitution. Si `SRCS` est définie comme précédemment :

```
OBJS = $(SRCS:.c=.o)
```

alors la valeur de `OBJS` est la valeur `SRCS` dans laquelle on a remplacé toutes les occurrences de `.c` par `.o`.

- Et encore bien d'autres fonctions comme `$(subst...)`, `$(filter...)`, `$(patsubst...)`, `$(filter-out...)`, et cætera.


Le premier exemple revisité à nouveau


```


PROG = prog

CC = gcc
CPPFLAGS = -I .
CFLAGS = -Wall -Wpointer-arith # -O3

SRCS = $(wildcard *.c) # liste des fichiers source
OBJS = $(SRCS:.c=.o) # liste des fichiers objet

# on utilise les variables prédéfinies '$^', '$<' et '$@'
$(PROG): $(OBJS)
     $(CC) $^ -o $@

prog.o: $(PROG).c src.h
     $(CC) $(CPPFLAGS) $(CFLAGS) -c $< -o $@


src.o: src.c src.h
     $(CC) $(CPPFLAGS) $(CFLAGS) -c $< -o $@
```


Le premier exemple revisité à nouveau


```
PROG = prog

CC = gcc
CPPFLAGS = -I .
CFLAGS = -Wall -Wpointer-arith # -O3

SRCS = $(wildcard *.c) # liste des fichiers source
OBJS = $(SRCS:.c=.o) # liste des fichiers objet

# on utilise les variables prédéfinies '$^', '$<' et '$@'
$(PROG): $(OBJS)
     $(CC) $^ -o $@

prog.o: $(PROG).c src.h
     $(CC) $(CPPFLAGS) $(CFLAGS) -c $< -o $@

src.o: src.c src.h
     $(CC) $(CPPFLAGS) $(CFLAGS) -c $< -o $@
```

Les cibles génériques

(1/3)

- Le logiciel `make` offre la possibilité de créer des cibles **génériques** qui seront appelées par défaut. Par exemple, pour créer un fichier objet à partir d'un fichier source :

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

- Si l'on reprend l'exemple précédent :

```
$(PROG): $(OBJS)
    $(CC) $^ -o $@
```

```
prog.o: src.h # dépendance non générique de 'prog.o'
```

```
src.o: src.h # dépendance non générique de 'src.o'
```

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

Les cibles génériques

(1/3)

- Le logiciel `make` offre la possibilité de créer des cibles **génériques** qui seront appelées par défaut. Par exemple, pour créer un fichier objet à partir d'un fichier source :

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

- Si l'on reprend l'exemple précédent :

```
$(PROG): $(OBSJ)
    $(CC) $^ -o $@
```

```
prog.o: src.h # dépendance non générique de 'prog.o'
```

```
src.o: src.h # dépendance non générique de 'src.o'
```

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

Les cibles génériques

(2/3)

- Un autre exemple de cible générique avec l'analyseur syntaxique `bison` de GNU (anciennement `yacc`) :

```
YACC = bison
%.tab.c %.tab.h: %.y
$(YACC) -d $<
```

Ici, à partir d'un fichier `parser.y`, pour fixer les idées, deux cibles vont être (re)construites : `parser.tab.c` et `parser.tab.h`.

- Si l'on revient à l'exemple qui nous a servi de fil conducteur, on constate que l'on n'a pas encore automatisé les dépendances.
- En effet, pour chaque fichier source, il faut extraire les fichiers d'en-tête (hors en-têtes standards) et ajouter la ligne correspondante dans le fichier `makefile` :

```
prog.o: src.h # dépendance non générique de 'prog.o'
src.o: src.h # dépendance non générique de 'src.o'
```

Cela peut devenir fastidieux si il y a beaucoup de fichiers source...

Les cibles génériques

(2/3)

- Un autre exemple de cible générique avec l'analyseur syntaxique `bison` de GNU (anciennement `yacc`) :

```
YACC = bison
%.tab.c %.tab.h: %.y
$(YACC) -d $<
```

Ici, à partir d'un fichier `parser.y`, pour fixer les idées, deux cibles vont être (re)construites : `parser.tab.c` et `parser.tab.h`.

- Si l'on revient à l'exemple qui nous a servi de fil conducteur, on constate que l'on n'a pas encore **automatisé les dépendances**.
- En effet, pour chaque fichier source, il faut extraire les fichiers d'en-tête (hors en-têtes standards) et ajouter la ligne correspondante dans le fichier `makefile` :

```
prog.o: src.h # dépendance non générique de 'prog.o'
src.o: src.h # dépendance non générique de 'src.o'
```

Cela peut devenir fastidieux si il y a beaucoup de fichiers source...

Les cibles génériques

(2/3)

- Un autre exemple de cible générique avec l'analyseur syntaxique `bison` de GNU (anciennement `yacc`) :

```
YACC = bison
%.tab.c %.tab.h: %.y
$(YACC) -d $<
```

Ici, à partir d'un fichier `parser.y`, pour fixer les idées, deux cibles vont être (re)construites : `parser.tab.c` et `parser.tab.h`.

- Si l'on revient à l'exemple qui nous a servi de fil conducteur, on constate que l'on n'a pas encore **automatisé les dépendances**.
- En effet, pour chaque fichier source, il faut extraire les fichiers d'en-tête (hors en-têtes standards) et ajouter la ligne correspondante dans le fichier **makefile** :

```
prog.o: src.h # dépendance non générique de 'prog.o'
src.o: src.h # dépendance non générique de 'src.o'
```

Cela peut devenir fastidieux si il y a beaucoup de fichiers source...

Les cibles génériques – gestion des dépendances

(3/3)

- La gestion automatisée des dépendances est réalisée grâce une option du compilateur GNU-CC, l'option `-MM` :

```
$ gcc -MM src.c -o src.d
```

Le fichier `src.d` sera alors constitué de la ligne :

```
src.o: hdr_1.h hdr_2.h ... hdr_n.h
```

- Le fichier `.depend` :

```
DEPS = $(SRCS:.c=.d) # liste des fichiers de dépendance
```

```
depend: .depend
```

```
.depend: $(DEPS)
```

```
cat $^ > $@ # création du fichier '.depend'
```

```
include .depend # inclusion du fichier '.depend'
```

```
%.d: %.c # règle générique pour les fichiers de dépendance
```

```
$(CC) $(CPPFLAGS) -MM $< -o $@
```


Les cibles génériques – gestion des dépendances

(3/3)

- La gestion automatisée des dépendances est réalisée grâce une option du compilateur GNU-CC, l'option `-MM` :

```
$ gcc -MM src.c -o src.d
```

Le fichier `src.d` sera alors constitué de la ligne :

```
src.o: hdr_1.h hdr_2.h ... hdr_n.h
```

- Le fichier `.depend` :

```
DEPS = $(SRCS:.c=.d) # liste des fichiers de dépendance
```

```
depend: .depend
```

```
.depend: $(DEPS)
```

```
cat $^ > $@ # création du fichier '.depend'
```

```
include .depend # inclusion du fichier '.depend'
```

```
%.d: %.c # règle générique pour les fichiers de dépendance
```

```
$(CC) $(CPPFLAGS) -MM $< -o $@
```

La cible spéciale .PHONY

- Prenons l'exemple de la cible `clean`, souvent présente dans les fichiers `makefile` :

```
clean: ; -rm -rf *.o *.d core *.core
```

Cette cible n'a pas de dépendance. Si d'aventure, un fichier de nom `clean` est créé dans le catalogue courant, alors la cible `clean` ne sera **plus jamais exécutée** car le fichier de même nom sera considéré **plus récent** en l'absence de dépendances.

- La cible spéciale `.PHONY` apporte la solution à ce problème :

```
.PHONY: clean very_clean dist
```

Les prérequis de la cible `.PHONY` seront **systématiquement** reconstruits indépendamment de la présence ou non dans le catalogue courant des fichiers `clean`, `very_clean` ou `dist`.

La cible spéciale .PHONY

- Prenons l'exemple de la cible `clean`, souvent présente dans les fichiers `makefile` :

```
clean: ; -rm -rf *.o *.d core *.core
```

Cette cible n'a pas de dépendance. Si d'aventure, un fichier de nom `clean` est créé dans le catalogue courant, alors la cible `clean` ne sera **plus jamais exécutée** car le fichier de même nom sera considéré **plus récent** en l'absence de dépendances.

- La cible spéciale `.PHONY` apporte la solution à ce problème :

```
.PHONY: clean very_clean dist
```

Les prérequis de la cible `.PHONY` seront **systematiquement** reconstruits indépendamment de la présence ou non dans le catalogue courant des fichiers `clean`, `very_clean` ou `dist`.

Les directives conditionnelles

- Ce sont les directives `ifeq`, `ifneq`, `ifdef` et `ifndef`. La syntaxe est la suivante :

```
ifeq condition                ifeq condition
...
endif                          else
...
endif
```

- Un exemple où l'on gère les dépendances différemment :

```
ifeq (.depend, $(wildcard .depend))
include .depend
all: $(PRGS)
else
all: depend
endif
depend:
@echo "Creating \ '.depend' ..."
$(CC) $(CPPFLAGS) -MM $(SRCS) > .$.@
@echo "Done. You have to rerun \ '$(MAKE)' "
```

Les directives conditionnelles

- Ce sont les directives `ifeq`, `ifneq`, `ifdef` et `ifndef`. La syntaxe est la suivante :

```

ifeq condition
...
endif
ifeq condition
...
else
...
endif

```

- Un exemple où l'on gère les dépendances différemment :

```

ifeq (.depend, $(wildcard .depend))
include .depend
all: $(PRGS)
else
all: depend
endif
depend:
@echo "Creating \ '.depend' ..."
$(CC) $(CPPFLAGS) -MM $(SRCS) > .$.@
@echo "Done. You have to rerun \ '$(MAKE)' "

```

Un fichier *makefile* pour gé(né)rer une bibliothèque

(1/4)

on commence par les définitions globales

LIB_NAME = test # le nom de la bibliothèque

INC_DIR = include # le catalogue où se trouvent les en-têtes

LIB_STATIC = lib\$(LIB_NAME).a # la bibliothèque statique

LIB_DYNAMIC = lib\$(LIB_NAME).so # la bibliothèque dynamique

CC = gcc # le compilateur GNU-CC

CPPFLAGS = -I \$(INC_DIR) # options du préprocesseur

CFLAGS = -Wall -Wpointer-arith -fPIC # options de compilation

AR = ar # commande permettant de créer la bib. statique

ARFLAGS = crsU # options de la commande 'ar'

RM = rm -rf # commande de suppression de fichiers/catalogues

MAKE = make # GNU-Make, 'gmake' sur certains OS

Un fichier *makefile* pour gé(né)rer une bibliothèque

(1/4)

on commence par les définitions globales

LIB_NAME = test # le nom de la bibliothèque

INC_DIR = include # le catalogue où se trouvent les en-têtes

LIB_STATIC = lib\$(LIB_NAME).a # la bibliothèque statique

LIB_DYNAMIC = lib\$(LIB_NAME).so # la bibliothèque dynamique

CC = gcc # le compilateur GNU-CC

CPPFLAGS = -I \$(INC_DIR) # options du préprocesseur

CFLAGS = -Wall -Wpointer-arith -fPIC # options de compilation

AR = ar # commande permettant de créer la bib. statique

ARFLAGS = crsU # options de la commande 'ar'

RM = rm -rf # commande de suppression de fichiers/catalogues

MAKE = make # GNU-Make, 'gmake' sur certains OS

Un fichier *makefile* pour gé(né)rer une bibliothèque

(1/4)

on commence par les définitions globales

LIB_NAME = test # le nom de la bibliothèque

INC_DIR = include # le catalogue où se trouvent les en-têtes

LIB_STATIC = lib\$(LIB_NAME).a # la bibliothèque statique

LIB_DYNAMIC = lib\$(LIB_NAME).so # la bibliothèque dynamique

CC = gcc # le compilateur GNU-CC

CPPFLAGS = -I \$(INC_DIR) # options du préprocesseur

CFLAGS = -Wall -Wpointer-arith -fPIC # options de compilation

AR = ar # commande permettant de créer la bib. statique

ARFLAGS = crsU # options de la commande 'ar'

RM = rm -rf # commande de suppression de fichiers/catalogues

MAKE = make # GNU-Make, 'gmake' sur certains OS

Un fichier *makefile* pour gé(né)rer une bibliothèque

(1/4)

on commence par les définitions globales

LIB_NAME = test # le nom de la bibliothèque

INC_DIR = include # le catalogue où se trouvent les en-têtes

LIB_STATIC = lib\$(LIB_NAME).a # la bibliothèque statique

LIB_DYNAMIC = lib\$(LIB_NAME).so # la bibliothèque dynamique

CC = gcc # le compilateur GNU-CC

CPPFLAGS = -I \$(INC_DIR) # options du préprocesseur

CFLAGS = -Wall -Wpointer-arith -fPIC # options de compilation

AR = ar # commande permettant de créer la bib. statique

ARFLAGS = crsU # options de la commande 'ar'

RM = rm -rf # commande de suppression de fichiers/catalogues

MAKE = make # GNU-Make, 'gmake' sur certains OS

Un fichier *makefile* pour gé(né)rer une bibliothèque

(1/4)

```
# on commence par les définitions globales
```

```
LIB_NAME = test # le nom de la bibliothèque
```

```
INC_DIR = include # le catalogue où se trouvent les en-têtes
```

```
LIB_STATIC = lib$(LIB_NAME).a # la bibliothèque statique
```

```
LIB_DYNAMIC = lib$(LIB_NAME).so # la bibliothèque dynamique
```

```
CC = gcc # le compilateur GNU-CC
```

```
CPPFLAGS = -I $(INC_DIR) # options du préprocesseur
```

```
CFLAGS = -Wall -Wpointer-arith -fPIC # options de compilation
```

```
AR = ar # commande permettant de créer la bib. statique
```

```
ARFLAGS = crsU # options de la commande 'ar'
```

```
RM = rm -rf # commande de suppression de fichiers/catalogues
```

```
MAKE = make # GNU-Make, 'gmake' sur certains OS
```

Un fichier *makefile* pour gé(né)rer une bibliothèque

(1/4)

```
# on commence par les définitions globales
```

```
LIB_NAME = test # le nom de la bibliothèque
```

```
INC_DIR = include # le catalogue où se trouvent les en-têtes
```

```
LIB_STATIC = lib$(LIB_NAME).a # la bibliothèque statique
```

```
LIB_DYNAMIC = lib$(LIB_NAME).so # la bibliothèque dynamique
```

```
CC = gcc # le compilateur GNU-CC
```

```
CPPFLAGS = -I $(INC_DIR) # options du préprocesseur
```

```
CFLAGS = -Wall -Wpointer-arith -fPIC # options de compilation
```

```
AR = ar # commande permettant de créer la bib. statique
```

```
ARFLAGS = crsU # options de la commande 'ar'
```

```
RM = rm -rf # commande de suppression de fichiers/catalogues
```

```
MAKE = make # GNU-Make, 'gmake' sur certains OS
```

Un fichier *makefile* pour gé(n)rer une bibliothèque

(2/4)

on va maintenant rechercher les fichiers source

```
SRCS = $(wildcard *.c) # liste des fichiers source
ifeq (, $(SRCS)) # on s'assure que la liste n'est pas vide
$(error No source files found in '.') # erreur fatale
endif
```

```
OBJS = $(SRCS:.c=.o) # liste des fichiers objet
DEPS = $(SRCS:.c=.d) # liste des fichiers de dépendance
```

```
.PHONY: clean very_clean # liste des cibles toujours
                        # reconstruites
```

les principales cibles

```
all: static dynamic # cible par défaut
```

```
static: depend $(LIB_STATIC) # cible pour la bib. statique
```

```
dynamic: depend $(LIB_DYNAMIC) # cible pour la bib. dynamique
```

Un fichier *makefile* pour gé(n)rer une bibliothèque

(2/4)

on va maintenant rechercher les fichiers source

```
SRCS = $(wildcard *.c) # liste des fichiers source
ifeq (, $(SRCS)) # on s'assure que la liste n'est pas vide
$(error No source files found in '.') # erreur fatale
endif
```

```
OBJS = $(SRCS:.c=.o) # liste des fichiers objet
DEPS = $(SRCS:.c=.d) # liste des fichiers de dépendance
```

```
.PHONY: clean very_clean # liste des cibles toujours
                        # reconstruites
```

les principales cibles

```
all: static dynamic # cible par défaut
```

```
static: depend $(LIB_STATIC) # cible pour la bib. statique
```

```
dynamic: depend $(LIB_DYNAMIC) # cible pour la bib. dynamique
```

Un fichier *makefile* pour gé(n)rer une bibliothèque

(2/4)

on va maintenant rechercher les fichiers source

```
SRCS = $(wildcard *.c) # liste des fichiers source
ifeq (, $(SRCS)) # on s'assure que la liste n'est pas vide
$(error No source files found in '.') # erreur fatale
endif
```

```
OBJS = $(SRCS:.c=.o) # liste des fichiers objet
DEPS = $(SRCS:.c=.d) # liste des fichiers de dépendance
```

```
.PHONY: clean very_clean # liste des cibles toujours
                        # reconstruites
```

les principales cibles

```
all: static dynamic # cible par défaut
```

```
static: depend $(LIB_STATIC) # cible pour la bib. statique
```

```
dynamic: depend $(LIB_DYNAMIC) # cible pour la bib. dynamique
```

Un fichier *makefile* pour gé(né)rer une bibliothèque

(2/4)

```
# on va maintenant rechercher les fichiers source
```

```
SRCS = $(wildcard *.c) # liste des fichiers source
ifeq (, $(SRCS)) # on s'assure que la liste n'est pas vide
$(error No source files found in '.') # erreur fatale
endif
```

```
OBJS = $(SRCS:.c=.o) # liste des fichiers objet
DEPS = $(SRCS:.c=.d) # liste des fichiers de dépendance
```

```
.PHONY: clean very_clean # liste des cibles toujours
                        # reconstruites
```

```
# les principales cibles
```

```
all: static dynamic # cible par défaut
```

```
static: depend $(LIB_STATIC) # cible pour la bib. statique
```

```
dynamic: depend $(LIB_DYNAMIC) # cible pour la bib. dynamique
```

Un fichier *makefile* pour gé(n)rer une bibliothèque

(3/4)

```
$(LIB_STATIC): $(OBJS)
    @echo "Creating/Updating static library '$(LIB_STATIC)'..."
    $(AR) $(ARFLAGS) $@ $? # notez l'utilisation de '$?'
    @echo "Done."

$(LIB_DYNAMIC): $(OBJS) # compilés nécessairement avec '-fPIC'
    @echo "Creating dynamic library '$(LIB_DYNAMIC)'..."
    $(CC) -shared $^ -o $@ # éditeur de liens : option '-shared'
    @echo "Done."

depend: .depend
.depend: $(DEPS) # les fichiers de dépendance (voir ci-après)
    @echo "Collecting dependencies in file '.depend'..."
    cat $^ > $@ # création du fichier '.depend'
    @echo "Done."
```

Le fichier *.depend*, automatiquement généré, aura l'allure suivante :

```
src1.o: src1.c
src2.o: src2.c include/src2.h
..... # une entrée pour chaque fichier source
```


Un fichier *makefile* pour gé(n)rer une bibliothèque

(3/4)

```
$(LIB_STATIC): $(OBJS)
    @echo "Creating/Updating static library '$(LIB_STATIC)'..."
    $(AR) $(ARFLAGS) $@ $? # notez l'utilisation de '$?'
    @echo "Done."

$(LIB_DYNAMIC): $(OBJS) # compilés nécessairement avec '-fPIC'
    @echo "Creating dynamic library '$(LIB_DYNAMIC)'..."
    $(CC) -shared $^ -o $@ # éditeur de liens : option '-shared'
    @echo "Done."

depend: .depend
.depend: $(DEPS) # les fichiers de dépendance (voir ci-après)
    @echo "Collecting dependencies in file '$.depend'..."
    cat $^ > $@ # création du fichier '.depend'
    @echo "Done."
```

Le fichier *.depend*, automatiquement généré, aura l'allure suivante :

```
src1.o: src1.c
src2.o: src2.c include/src2.h
..... # une entrée pour chaque fichier source
```

Un fichier *makefile* pour gé(n)rer une bibliothèque

(3/4)

```
$(LIB_STATIC): $(OBJS)
    @echo "Creating/Updating static library '$(LIB_STATIC)'..."
    $(AR) $(ARFLAGS) $@ $? # notez l'utilisation de '$?'
    @echo "Done."

$(LIB_DYNAMIC): $(OBJS) # compilés nécessairement avec '-fPIC'
    @echo "Creating dynamic library '$(LIB_DYNAMIC)'..."
    $(CC) -shared $^ -o $@ # éditeur de liens : option '-shared'
    @echo "Done."

depend: .depend
.depend: $(DEPS) # les fichiers de dépendance (voir ci-après)
    @echo "Collecting dependencies in file '$.depend'..."
    cat $^ > $@ # création du fichier '.depend'
    @echo "Done."
```

Le fichier *.depend*, automatiquement généré, aura l'allure suivante :

```
src1.o: src1.c
src2.o: src2.c include/src2.h
..... # une entrée pour chaque fichier source
```

Un fichier *makefile* pour gé(n)rer une bibliothèque

(3/4)

```
$(LIB_STATIC): $(OBJS)
    @echo "Creating/Updating static library '$(LIB_STATIC)'..."
    $(AR) $(ARFLAGS) $@ $? # notez l'utilisation de '$?'
    @echo "Done."

$(LIB_DYNAMIC): $(OBJS) # compilés nécessairement avec '-fPIC'
    @echo "Creating dynamic library '$(LIB_DYNAMIC)'..."
    $(CC) -shared $^ -o $@ # éditeur de liens : option '-shared'
    @echo "Done."

depend: .depend
.depend: $(DEPS) # les fichiers de dépendance (voir ci-après)
    @echo "Collecting dependencies in file '$.depend'..."
    cat $^ > $@ # création du fichier '.depend'
    @echo "Done."
```

Le fichier **.depend**, automatiquement généré, aura l'allure suivante :

```
src1.o: src1.c
src2.o: src2.c include/src2.h
..... # une entrée pour chaque fichier source
```

Un fichier *makefile* pour gé(n)rer une bibliothèque

(4/4)

```
# on inclut '.depend' s'il existe dans le catalogue courant

ifeq (.depend, $(wildcard .depend))
include .depend
endif

%.d: %.c # règle générique pour les fichiers de dépendance
$(CC) $(CPPFLAGS) -MM $< -o $@

%.o: %.c # règle générique pour les fichiers objet
$(CC) $(CPPFLAGS) $(CFLAGS) -c $< -o $@

clean: # suppression des fichiers intermédiaires (et autres)
-$(RM) $(OBJS) $(DEPS) .depend
-$(RM) *~ \#*\# .\#* core *core

very_clean: # on ne garde que les sources et les en-têtes
@$(MAKE) --no-print-directory clean
-$(RM) $(LIB_STATIC) $(LIB_DYNAMIC)
```

Un fichier *makefile* pour gé(n)rer une bibliothèque

(4/4)

```
# on inclut '.depend' s'il existe dans le catalogue courant

ifeq (.depend, $(wildcard .depend))
include .depend
endif

%.d: %.c # règle générique pour les fichiers de dépendance
$(CC) $(CPPFLAGS) -MM $< -o $@

%.o: %.c # règle générique pour les fichiers objet
$(CC) $(CPPFLAGS) $(CFLAGS) -c $< -o $@

clean: # suppression des fichiers intermédiaires (et autres)
-$(RM) $(OBJS) $(DEPS) .depend
-$(RM) *~ \#*\# .\#* core *core

very_clean: # on ne garde que les sources et les en-têtes
@$(MAKE) --no-print-directory clean
-$(RM) $(LIB_STATIC) $(LIB_DYNAMIC)
```

Un fichier *makefile* pour gé(né)rer une bibliothèque

(4/4)

```
# on inclut '.depend' s'il existe dans le catalogue courant

ifeq (.depend, $(wildcard .depend))
include .depend
endif

%.d: %.c # règle générique pour les fichiers de dépendance
$(CC) $(CPPFLAGS) -MM $< -o $@

%.o: %.c # règle générique pour les fichiers objet
$(CC) $(CPPFLAGS) $(CFLAGS) -c $< -o $@

clean: # suppression des fichiers intermédiaires (et autres)
-$(RM) $(OBJS) $(DEPS) .depend
-$(RM) *~ \#*\# .\#* core *core

very_clean: # on ne garde que les sources et les en-têtes
@$(MAKE) --no-print-directory clean
-$(RM) $(LIB_STATIC) $(LIB_DYNAMIC)
```

Comment utiliser `libtest.a` et `libtest.so`?

(1/2)

- Supposons que l'on ait un programme `prog.c` utilisant la bibliothèque `libtest` et que, pour fixer les idées, cette dernière se trouve dans le sous-catalogue `./testdir` du catalogue courant.

- Pour générer l'exécutable `prog` :

```
$ gcc prog.c -I testdir/include -L testdir -o prog \
    -ltest
```

- La recherche de la bibliothèque (option `-ltest`) se fait par défaut dans les catalogues système (`/lib`, `/usr/lib`, ...) et également dans chaque catalogue spécifié par une option `-L <dir>` de la commande `gcc`, ici l'option `-L testdir`.
- Le nom complet de la bibliothèque recherchée, option `-ltest`, sera `libtest.a` (bib. statique) ou `libtest.so` (bib. dynamique). Notez la présence du **préfixe** `lib` et des **suffixes** `.a` et `.so`.

Comment utiliser `libtest.a` et `libtest.so` ?

(1/2)

- Supposons que l'on ait un programme `prog.c` utilisant la bibliothèque `libtest` et que, pour fixer les idées, cette dernière se trouve dans le sous-catalogue `./testdir` du catalogue courant.
- Pour générer l'exécutable `prog` :

```
$ gcc prog.c -I testdir/include -L testdir -o prog \
    -ltest
```

- La recherche de la bibliothèque (option `-ltest`) se fait par défaut dans les catalogues système (`/lib`, `/usr/lib`, ...) et également dans chaque catalogue spécifié par une option `-L <dir>` de la commande `gcc`, ici l'option `-L testdir`.
- Le nom complet de la bibliothèque recherchée, option `-ltest`, sera `libtest.a` (bib. statique) ou `libtest.so` (bib. dynamique). Notez la présence du **préfixe** `lib` et des **suffixes** `.a` et `.so`.

Comment utiliser `libtest.a` et `libtest.so`?

(1/2)

- Supposons que l'on ait un programme `prog.c` utilisant la bibliothèque `libtest` et que, pour fixer les idées, cette dernière se trouve dans le sous-catalogue `./testdir` du catalogue courant.
- Pour générer l'exécutable `prog` :

```
$ gcc prog.c -I testdir/include -L testdir -o prog \
    -ltest
```

- La recherche de la bibliothèque (option `-ltest`) se fait par défaut dans les catalogues système (`/lib`, `/usr/lib`, ...) et également dans chaque catalogue spécifié par une option `-L <dir>` de la commande `gcc`, ici l'option `-L testdir`.
- Le nom complet de la bibliothèque recherchée, option `-ltest`, sera `libtest.a` (bib. statique) ou `libtest.so` (bib. dynamique). Notez la présence du préfixe `lib` et des suffixes `.a` et `.so`.

Comment utiliser `libtest.a` et `libtest.so` ?

(1/2)

- Supposons que l'on ait un programme `prog.c` utilisant la bibliothèque `libtest` et que, pour fixer les idées, cette dernière se trouve dans le sous-catalogue `./testdir` du catalogue courant.
- Pour générer l'exécutable `prog` :

```
$ gcc prog.c -I testdir/include -L testdir -o prog \
    -ltest
```

- La recherche de la bibliothèque (option `-ltest`) se fait par défaut dans les catalogues système (`/lib`, `/usr/lib`, ...) et également dans chaque catalogue spécifié par une option `-L <dir>` de la commande `gcc`, ici l'option `-L testdir`.
- Le nom complet de la bibliothèque recherchée, option `-ltest`, sera `libtest.a` (bib. statique) ou `libtest.so` (bib. dynamique). Notez la présence du **préfixe** `lib` et des **suffixes** `.a` et `.so`.

Comment utiliser libtest.a et libtest.so?

(2/2)

- Si `gcc` trouve les deux bibliothèques (statique et dynamique), il choisira par défaut la bibliothèque dynamique.
- Pour forcer le choix de la bibliothèque statique :

```
$ gcc prog.c -static -I testdir/include -L testdir \  
-o prog -ltest
```

- Si l'on a choisi la bibliothèque dynamique, on peut rencontrer le problème suivant :

```
$ ./prog  
./prog: error while loading shared libraries: libtest.so:  
cannot open shared object file: No such file or directory
```

- Pour corriger, il faut mettre à jour la variable d'environnement `LD_LIBRARY_PATH`. Par exemple, sous `bash` :

```
$ LD_LIBRARY_PATH='pwd' /testdir  
$ export LD_LIBRARY_PATH
```

Comment utiliser libtest.a et libtest.so?

(2/2)

- Si `gcc` trouve les deux bibliothèques (statique et dynamique), il choisira par défaut la bibliothèque dynamique.
- Pour forcer le choix de la bibliothèque statique :

```
$ gcc prog.c -static -I testdir/include -L testdir \  
-o prog -ltest
```

- Si l'on a choisi la bibliothèque dynamique, on peut rencontrer le problème suivant :

```
$ ./prog  
./prog: error while loading shared libraries: libtest.so:  
cannot open shared object file: No such file or directory
```

- Pour corriger, il faut mettre à jour la variable d'environnement `LD_LIBRARY_PATH`. Par exemple, sous `bash` :

```
$ LD_LIBRARY_PATH='pwd' /testdir  
$ export LD_LIBRARY_PATH
```

Comment utiliser libtest.a et libtest.so?

(2/2)

- Si `gcc` trouve les deux bibliothèques (statique et dynamique), il choisira par défaut la bibliothèque dynamique.
- Pour forcer le choix de la bibliothèque statique :

```
$ gcc prog.c -static -I testdir/include -L testdir \  
-o prog -ltest
```

- Si l'on a choisi la bibliothèque dynamique, on peut rencontrer le problème suivant :

```
$ ./prog  
./prog: error while loading shared libraries: libtest.so:  
cannot open shared object file: No such file or directory
```

- Pour corriger, il faut mettre à jour la variable d'environnement `LD_LIBRARY_PATH`. Par exemple, sous `bash` :

```
$ LD_LIBRARY_PATH='pwd' /testdir  
$ export LD_LIBRARY_PATH
```

Comment utiliser libtest.a et libtest.so?

(2/2)

- Si `gcc` trouve les deux bibliothèques (statique et dynamique), il choisira par défaut la bibliothèque dynamique.
- Pour forcer le choix de la bibliothèque statique :

```
$ gcc prog.c -static -I testdir/include -L testdir \  
    -o prog -ltest
```

- Si l'on a choisi la bibliothèque dynamique, on peut rencontrer le problème suivant :

```
$ ./prog  
./prog: error while loading shared libraries: libtest.so:  
cannot open shared object file: No such file or directory
```

- Pour corriger, il faut mettre à jour la variable d'environnement `LD_LIBRARY_PATH`. Par exemple, sous `bash` :

```
$ LD_LIBRARY_PATH='pwd' /testdir  
$ export LD_LIBRARY_PATH
```

Pourquoi a-t-on besoin d'une telle bibliothèque ?

(1/3)

- Considérons le cas d'une fonction `func` qui ouvre un fichier, en analyse le contenu et retourne un tableau d'entiers alloué dynamiquement et calculé en fonction du contenu.
- Quel prototype pour la fonction `func` ? Si on choisit comme prototype `int *func(const char *)`, on ne peut pas distinguer entre les différentes erreurs pouvant survenir :

```
int *func(const char *fname) {
    int fd, nb, *ret;
    fd = open(fname, O_RDONLY);
    if (fd < 0)
        return NULL;
    ...
    ret = malloc(nb * sizeof(int));
    if (!ret)
        return NULL; /* on retourne NULL comme au dessus */
    ...
}
```

Pourquoi a-t-on besoin d'une telle bibliothèque ?

(1/3)

- Considérons le cas d'une fonction `func` qui ouvre un fichier, en analyse le contenu et retourne un tableau d'entiers alloué dynamiquement et calculé en fonction du contenu.
- Quel prototype pour la fonction `func` ? Si on choisit comme prototype `int *func(const char *)`, on ne peut pas **distinguer** entre les **différentes erreurs** pouvant survenir :

```
int *func(const char *fname) {
    int fd, nb, *ret;
    fd = open(fname, O_RDONLY);
    if (fd < 0)
        return NULL;
    ...
    ret = malloc(nb * sizeof(int));
    if (!ret)
        return NULL; /* on retourne NULL comme au dessus */
    ...
}
```


Pourquoi a-t-on besoin d'une telle bibliothèque ?

(2/3)

- Un prototype plus adapté serait :

```
int func(const char *fname, int **res) {  
    int fd, nb;  
    fd = open(fname, O_RDONLY);  
    if (fd < 0)  
        return -1; /* code d'erreur */  
    ...  
    *res = malloc(nb * sizeof(int));  
    if (!*res)  
        return -2; /* code d'erreur */  
    ...  
}
```

- Cela revient donc à renvoyer un **code d'erreur** spécifique pour chaque type d'erreur rencontré.
- C'est une manière tout à fait correcte de gérer les erreurs mais cette gestion peut devenir **malaisée** si l'on a beaucoup de fonctions et/ou de codes d'erreur.

Pourquoi a-t-on besoin d'une telle bibliothèque ?

(2/3)

- Un prototype plus adapté serait :

```
int func(const char *fname, int **res) {  
    int fd, nb;  
    fd = open(fname, O_RDONLY);  
    if (fd < 0)  
        return -1; /* code d'erreur */  
    ...  
    *res = malloc(nb * sizeof(int));  
    if (!*res)  
        return -2; /* code d'erreur */  
    ...  
}
```

- Cela revient donc à renvoyer un **code d'erreur** spécifique pour chaque type d'erreur rencontré.
- C'est une manière tout à fait correcte de gérer les erreurs mais cette gestion peut devenir **malaisée** si l'on a beaucoup de fonctions et/ou de codes d'erreur.

Pourquoi a-t-on besoin d'une telle bibliothèque ?

(2/3)

- Un prototype plus adapté serait :

```
int func(const char *fname, int **res) {  
    int fd, nb;  
    fd = open(fname, O_RDONLY);  
    if (fd < 0)  
        return -1; /* code d'erreur */  
    ...  
    *res = malloc(nb * sizeof(int));  
    if (!*res)  
        return -2; /* code d'erreur */  
    ...  
}
```

- Cela revient donc à renvoyer un **code d'erreur** spécifique pour chaque type d'erreur rencontré.
- C'est une manière tout à fait correcte de gérer les erreurs mais cette gestion peut devenir **malaisée** si l'on a beaucoup de fonctions et/ou de codes d'erreur.

Pourquoi a-t-on besoin d'une telle bibliothèque ?

(3/3)

- Si l'on revient au code d'erreur unique :

```
int *func(const char *fname) {
    int fd, nb, *ret;
    fd = open(fname, O_RDONLY);
    if (fd < 0) {
        fprintf(stderr, "can't open %s\n", fname);
        return NULL;
    }
    ...
    ret = malloc(nb * sizeof(int));
    if (!ret) {
        fprintf(stderr, "out of memory\n")
        return NULL;
    }
    ...
}
```

- Problème : les erreurs sont traitées immédiatement, ce qui ne laisse aucune liberté à l'utilisateur de votre bibliothèque.

Pourquoi a-t-on besoin d'une telle bibliothèque ?

(3/3)

- Si l'on revient au code d'erreur unique :

```
int *func(const char *fname) {
    int fd, nb, *ret;
    fd = open(fname, O_RDONLY);
    if (fd < 0) {
        fprintf(stderr, "can't open %s\n", fname);
        return NULL;
    }
    ...
    ret = malloc(nb * sizeof(int));
    if (!ret) {
        fprintf(stderr, "out of memory\n")
        return NULL;
    }
    ...
}
```

- **Problème** : les erreurs sont traitées immédiatement, ce qui ne laisse aucune liberté à l'utilisateur de votre bibliothèque.

Une première ébauche

(1/2)

- Dans l'exemple précédent, on aurait besoin, d'une part, d'une fonction `error_sys_set` qui sauve le message d'erreur et, d'autre part, d'une fonction `error_what` qui permette de récupérer ce message.
- Dans le code de la fonction `func` :

```
ret = malloc(nb * sizeof(int));
if (!ret) {
    error_sys_set(__func__, "malloc");
    return NULL;
}
```

- Dans le code de l'utilisateur :

```
int *ret;

ret = func("file");
if (!ret) {
    fprintf(stderr, "fatal error: %s\n", error_what());
    exit(1);
}
```

Une première ébauche

(1/2)

- Dans l'exemple précédent, on aurait besoin, d'une part, d'une fonction `error_sys_set` qui sauve le message d'erreur et, d'autre part, d'une fonction `error_what` qui permette de récupérer ce message.
- Dans le code de la fonction `func` :

```
ret = malloc(nb * sizeof(int));  
if (!ret) {  
    error_sys_set(__func__, "malloc");  
    return NULL;  
}
```

- Dans le code de l'utilisateur :

```
int *ret;  
  
ret = func("file");  
if (!ret) {  
    fprintf(stderr, "fatal error: %s\n", error_what());  
    exit(1);  
}
```

Une première ébauche

(1/2)

- Dans l'exemple précédent, on aurait besoin, d'une part, d'une fonction `error_sys_set` qui sauve le message d'erreur et, d'autre part, d'une fonction `error_what` qui permette de récupérer ce message.
- Dans le code de la fonction `func` :

```
ret = malloc(nb * sizeof(int));
if (!ret) {
    error_sys_set(__func__, "malloc");
    return NULL;
}
```

- Dans le code de l'utilisateur :

```
int *ret;

ret = func("file");
if (!ret) {
    fprintf(stderr, "fatal error: %s\n", error_what());
    exit(1);
}
```


Une première ébauche

(2/2)

- Les considérations précédentes induisent le fichier d'en-tête `error.h` suivant :

```
#ifndef ERROR_H
#define ERROR_H

void error_sys_set(const char *, const char *);
const char *error_what(void);

#endif /* ERROR_H */
```

- On pourrait également rajouter la fonction `error_print` qui imprime le message d'erreur sur la sortie `stderr`. Son prototype serait :

```
/* à rajouter dans error.h */

void error_print(void);
```

Une première ébauche

(2/2)

- Les considérations précédentes induisent le fichier d'en-tête `error.h` suivant :

```
#ifndef ERROR_H
#define ERROR_H

void error_sys_set(const char *, const char *);
const char *error_what(void);

#endif /* ERROR_H */
```

- On pourrait également rajouter la fonction `error_print` qui imprime le message d'erreur sur la sortie `stderr`. Son prototype serait :

```
/* à rajouter dans error.h */

void error_print(void);
```

Le code des fonctions – fichier `error.c`

(1/4)

```
/* les inclusions nécessaires */
#include <errno.h> /* variable errno */
#include <string.h> /* prototype de strerror() */
#include <error.h> /* notre fichier d'en-tête */

/* le tampon contenant le message d'erreur */
#define BUF_ERR_SIZE 256
static char buf_err[BUF_ERR_SIZE];

/* la fonction error_sys_set */
void
error_sys_set(const char *func, const char *sys_call)
{
    (void) snprintf(
        buf_err, BUF_ERR_SIZE, "%s(): %s(): (%d, %s)",
        func, sys_call, errno, strerror(errno)
    );
}
```

Le code des fonctions – fichier error.c

(1/4)

```
/* les inclusions nécessaires */
#include <errno.h> /* variable errno */
#include <string.h> /* prototype de strerror() */
#include <error.h> /* notre fichier d'en-tête */

/* le tampon contenant le message d'erreur */
#define BUF_ERR_SIZE 256
static char buf_err[BUF_ERR_SIZE];

/* la fonction error_sys_set */
void
error_sys_set(const char *func, const char *sys_call)
{
    (void) snprintf(
        buf_err, BUF_ERR_SIZE, "%s(): %s(): (%d, %s)",
        func, sys_call, errno, strerror(errno)
    );
}
```

Le code des fonctions – fichier error.c

(1/4)

```
/* les inclusions nécessaires */
#include <errno.h> /* variable errno */
#include <string.h> /* prototype de strerror() */
#include <error.h> /* notre fichier d'en-tête */

/* le tampon contenant le message d'erreur */
#define BUF_ERR_SIZE 256
static char buf_err[BUF_ERR_SIZE];

/* la fonction error_sys_set */
void
error_sys_set(const char *func, const char *sys_call)
{
    (void) snprintf(
        buf_err, BUF_ERR_SIZE, "%s():_%s():_(%d,_%s)",
        func, sys_call, errno, strerror(errno)
    );
}
```

Le code des fonctions – fichier error.c

(1/4)

```
/* les inclusions nécessaires */
#include <errno.h> /* variable errno */
#include <string.h> /* prototype de strerror() */
#include <error.h> /* notre fichier d'en-tête */

/* le tampon contenant le message d'erreur */
#define BUF_ERR_SIZE 256
static char buf_err[BUF_ERR_SIZE];

/* la fonction error_sys_set */
void
error_sys_set(const char *func, const char *sys_call)
{
    (void) snprintf(
        buf_err, BUF_ERR_SIZE, "%s(): %s(): (%d, %s)",
        func, sys_call, errno, strerror(errno)
    );
}
```

Le code des fonctions – fichier `error.c`

(2/4)

Le code des fonctions d'impression est particulièrement simple :

```
/* la fonction error_what */
const char *
error_what(void) {
    return buf_err
}

/* la fonction error_print */
void
error_print(void)
{
    (void) fprintf(stderr, "%s\n", buf_err);
}
```

Attention : le tampon `buf_err` est écrasé à chaque appel de la fonction `error_sys_set`. Ceci doit être signalé dans la documentation fournie à l'utilisateur de la bibliothèque.

Le code des fonctions – fichier `error.c`

(2/4)

Le code des fonctions d'impression est particulièrement simple :

```
/* la fonction error_what */
const char *
error_what(void) {
    return buf_err
}

/* la fonction error_print */
void
error_print(void)
{
    (void) fprintf(stderr, "%s\n", buf_err);
}
```

Attention : le tampon `buf_err` est écrasé à chaque appel de la fonction `error_sys_set`. Ceci doit être signalé dans la documentation fournie à l'utilisateur de la bibliothèque.

Le code des fonctions – fichier `error.c`

(2/4)

Le code des fonctions d'impression est particulièrement simple :

```
/* la fonction error_what */
const char *
error_what(void) {
    return buf_err
}

/* la fonction error_print */
void
error_print(void)
{
    (void) fprintf(stderr, "%s\n", buf_err);
}
```

Attention : le tampon `buf_err` est écrasé à chaque appel de la fonction `error_sys_set`. Ceci doit être signalé dans la documentation fournie à l'utilisateur de la bibliothèque.

Le code des fonctions – fichier `error.c`

(2/4)

Le code des fonctions d'impression est particulièrement simple :

```
/* la fonction error_what */
const char *
error_what(void) {
    return buf_err
}

/* la fonction error_print */
void
error_print(void)
{
    (void) fprintf(stderr, "%s\n", buf_err);
}
```

Attention : le tampon `buf_err` est écrasé à chaque appel de la fonction `error_sys_set`. Ceci doit être signalé dans la documentation fournie à l'utilisateur de la bibliothèque.

Une fonction généraliste : `error_set`

(3/4)

- On a jusqu'à présent traité seulement le cas d'une erreur survenue lors d'un appel à une primitive système (comme `malloc` ou `open`).
- On aurait besoin d'une fonction plus générale, analogue à la primitive `printf`. On aimerait, par exemple pouvoir écrire :

```
error_set(__func__, "can't open %s: error code: %d\n",  
         fname, errno);
```

- Une telle fonction aurait ainsi pour prototype :

```
void error_set(const char *, const char *, ...);
```

Le symbole « ... » (en anglais *ellipsis*) indique que la fonction possède un nombre variable d'arguments.

- Bien évidemment, il serait souhaitable que les formats utilisés par les fonctions `error_set` et `printf` soient identiques.
- Pour satisfaire toutes ces exigences, on va utiliser la primitive de la bibliothèque standard `vsnprintf`.

Une fonction généraliste : `error_set`

(3/4)

- On a jusqu'à présent traité seulement le cas d'une erreur survenue lors d'un appel à une primitive système (comme `malloc` ou `open`).
- On aurait besoin d'une fonction plus générale, analogue à la primitive `printf`. On aimerait, par exemple pouvoir écrire :

```
error_set(__func__, "can't open%s: error code: %d\n",  
          fname, errno);
```

- Une telle fonction aurait ainsi pour prototype :

```
void error_set(const char *, const char *, ...);
```

Le symbole « ... » (en anglais *ellipsis*) indique que la fonction possède un nombre variable d'arguments.

- Bien évidemment, il serait souhaitable que les formats utilisés par les fonctions `error_set` et `printf` soient identiques.
- Pour satisfaire toutes ces exigences, on va utiliser la primitive de la bibliothèque standard `vsnprintf`.

Une fonction généraliste : `error_set`

(3/4)

- On a jusqu'à présent traité seulement le cas d'une erreur survenue lors d'un appel à une primitive système (comme `malloc` ou `open`).
- On aurait besoin d'une fonction plus générale, analogue à la primitive `printf`. On aimerait, par exemple pouvoir écrire :

```
error_set(__func__, "can't open%s: error code: %d\n",  
         fname, errno);
```

- Une telle fonction aurait ainsi pour prototype :

```
void error_set(const char *, const char *, ...);
```

Le symbole « ... » (en anglais *ellipsis*) indique que la fonction possède un nombre variable d'arguments.

- Bien évidemment, il serait souhaitable que les formats utilisés par les fonctions `error_set` et `printf` soient identiques.
- Pour satisfaire toutes ces exigences, on va utiliser la primitive de la bibliothèque standard `vsnprintf`.

Une fonction généraliste : `error_set`

(3/4)

- On a jusqu'à présent traité seulement le cas d'une erreur survenue lors d'un appel à une primitive système (comme `malloc` ou `open`).
- On aurait besoin d'une fonction plus générale, analogue à la primitive `printf`. On aimerait, par exemple pouvoir écrire :

```
error_set(__func__, "can't open%s: error code: %d\n",  
         fname, errno);
```

- Une telle fonction aurait ainsi pour prototype :

```
void error_set(const char *, const char *, ...);
```

Le symbole « ... » (en anglais *ellipsis*) indique que la fonction possède un nombre variable d'arguments.

- Bien évidemment, il serait souhaitable que les formats utilisés par les fonctions `error_set` et `printf` soient identiques.
- Pour satisfaire toutes ces exigences, on va utiliser la primitive de la bibliothèque standard `vsnprintf`.

Une fonction généraliste : `error_set`

(3/4)

- On a jusqu'à présent traité seulement le cas d'une erreur survenue lors d'un appel à une primitive système (comme `malloc` ou `open`).
- On aurait besoin d'une fonction plus générale, analogue à la primitive `printf`. On aimerait, par exemple pouvoir écrire :

```
error_set(__func__, "can't open %s: error code: %d\n",  
         fname, errno);
```

- Une telle fonction aurait ainsi pour prototype :

```
void error_set(const char *, const char *, ...);
```

Le symbole « ... » (en anglais *ellipsis*) indique que la fonction possède un nombre variable d'arguments.

- Bien évidemment, il serait souhaitable que les formats utilisés par les fonctions `error_set` et `printf` soient identiques.
- Pour satisfaire toutes ces exigences, on va utiliser la primitive de la bibliothèque standard `vsnprintf`.

Le code des fonctions – fichier `error.c`

(4/4)

```
/* la fonction error_set */

#include <stdarg.h> /* nécessaire pour `...' */

void
error_set(const char *func, const char *fmt, ...)
{
    va_list ap; /* type défini dans `stdarg.h' */
    size_t offset = strlen(func) + 4;

    (void) snprintf(buf_err, BUF_ERR_SIZE, "%s():_", func);
    va_start(ap, fmt); /* macro définie dans `stdarg.h' */
    (void) vsnprintf(
        buf_err + offset, BUF_ERR_SIZE - offset, fmt, ap);
    va_end(ap); /* macro définie dans `stdarg.h' */
}
```

Une bibliothèque « thread safe »

(1/4)

- La bibliothèque développée précédemment n'est pas « thread safe ».
- En effet, toutes les fonctions de cette bibliothèque font usage du tampon `buf_err` déclaré (dans la classe de variables `static`) dans le fichier source `error.c`.
- Ainsi, si cette bibliothèque est utilisée par un programme comportant plusieurs *threads*, il risque d'y avoir des conflits d'accès par les différentes *threads* au tampon `buf_err`.
- Un moyen simple de pallier ce conflit est d'ajouter un paramètre à toutes les fonctions de la bibliothèque, ce paramètre étant tout simplement un tampon destiné à recueillir le message d'erreur.
- Chaque *thread* pourra alors disposer de son propre tampon.
- Il faudra également remplacer dans le code la fonction `strerror`, qui n'est pas « thread safe », par la fonction `strerror_r` qui elle l'est.

Une bibliothèque « thread safe »

(1/4)

- La bibliothèque développée précédemment n'est pas « thread safe ».
- En effet, toutes les fonctions de cette bibliothèque font usage du tampon `buf_err` déclaré (dans la classe de variables `static`) dans le fichier source `error.c`.
- Ainsi, si cette bibliothèque est utilisée par un programme comportant plusieurs *threads*, il risque d'y avoir des conflits d'accès par les différentes *threads* au tampon `buf_err`.
- Un moyen simple de pallier ce conflit est d'ajouter un paramètre à toutes les fonctions de la bibliothèque, ce paramètre étant tout simplement un tampon destiné à recueillir le message d'erreur.
- Chaque *thread* pourra alors disposer de son propre tampon.
- Il faudra également remplacer dans le code la fonction `strerror`, qui n'est pas « thread safe », par la fonction `strerror_r` qui elle l'est.

Une bibliothèque « thread safe »

(1/4)

- La bibliothèque développée précédemment n'est pas « thread safe ».
- En effet, toutes les fonctions de cette bibliothèque font usage du tampon `buf_err` déclaré (dans la classe de variables `static`) dans le fichier source `error.c`.
- Ainsi, si cette bibliothèque est utilisée par un programme comportant plusieurs *threads*, il risque d'y avoir des conflits d'accès par les différentes *threads* au tampon `buf_err`.
- Un moyen simple de pallier ce conflit est d'ajouter un paramètre à toutes les fonctions de la bibliothèque, ce paramètre étant tout simplement un tampon destiné à recueillir le message d'erreur.
- Chaque *thread* pourra alors disposer de son propre tampon.
- Il faudra également remplacer dans le code la fonction `strerror`, qui n'est pas « thread safe », par la fonction `strerror_r` qui elle l'est.

Une bibliothèque « thread safe »

(1/4)

- La bibliothèque développée précédemment n'est pas « thread safe ».
- En effet, toutes les fonctions de cette bibliothèque font usage du tampon `buf_err` déclaré (dans la classe de variables `static`) dans le fichier source `error.c`.
- Ainsi, si cette bibliothèque est utilisée par un programme comportant plusieurs *threads*, il risque d'y avoir des conflits d'accès par les différentes *threads* au tampon `buf_err`.
- Un moyen simple de pallier ce conflit est d'**ajouter un paramètre** à toutes les fonctions de la bibliothèque, ce paramètre étant tout simplement un tampon destiné à recueillir le message d'erreur.
- Chaque *thread* pourra alors disposer de son propre tampon.
- Il faudra également remplacer dans le code la fonction `strerror`, qui n'est pas « thread safe », par la fonction `strerror_r` qui elle l'est.

Une bibliothèque « thread safe »

(1/4)

- La bibliothèque développée précédemment n'est pas « thread safe ».
- En effet, toutes les fonctions de cette bibliothèque font usage du tampon `buf_err` déclaré (dans la classe de variables `static`) dans le fichier source `error.c`.
- Ainsi, si cette bibliothèque est utilisée par un programme comportant plusieurs *threads*, il risque d'y avoir des conflits d'accès par les différentes *threads* au tampon `buf_err`.
- Un moyen simple de pallier ce conflit est d'**ajouter un paramètre** à toutes les fonctions de la bibliothèque, ce paramètre étant tout simplement un tampon destiné à recueillir le message d'erreur.
- Chaque *thread* pourra alors disposer de son propre tampon.
- Il faudra également remplacer dans le code la fonction `strerror`, qui n'est pas « thread safe », par la fonction `strerror_r` qui elle l'est.

Une bibliothèque « thread safe »

(1/4)

- La bibliothèque développée précédemment n'est pas « thread safe ».
- En effet, toutes les fonctions de cette bibliothèque font usage du tampon `buf_err` déclaré (dans la classe de variables `static`) dans le fichier source `error.c`.
- Ainsi, si cette bibliothèque est utilisée par un programme comportant plusieurs *threads*, il risque d'y avoir des conflits d'accès par les différentes *threads* au tampon `buf_err`.
- Un moyen simple de pallier ce conflit est d'**ajouter un paramètre** à toutes les fonctions de la bibliothèque, ce paramètre étant tout simplement un tampon destiné à recueillir le message d'erreur.
- Chaque *thread* pourra alors disposer de son propre tampon.
- Il faudra également remplacer dans le code la fonction `strerror`, qui n'est pas « thread safe », par la fonction `strerror_r` qui elle l'est.

Une bibliothèque « thread safe »

(2/4)

Le nouveau fichier d'en-tête, `error_r.h` aurait l'allure suivante :

```
#ifndef ERROR_R_H
#define ERROR_R_H

#define ERROR_BUF_SIZE 256

/* `error_r_t' sera le type du nouveau paramètre */
typedef char error_r_t[ERROR_BUF_SIZE];

void error_r_set(error_r_t, const char *, const char *,...);
void error_r_sys_set(error_r_t, const char *, const char *);
const char *error_r_what(const error_r_t);
void error_r_print(const error_r_t);

#endif /* ERROR_R_H */
```

Une bibliothèque « thread safe »

(3/4)

La nouvelle fonction `error_r_set` présente peu de différences avec la fonction `error_set` :

```
void
error_r_set(
    error_r_t err, const char *func, const char *fmt, ...)
{
    va_list ap;
    size_t offset = strlen(func) + 4;

    (void) snprintf(err, sizeof(error_r_t), "%s():_", func);
    va_start(ap, fmt);
    (void) vsnprintf(
        err + offset, sizeof(error_r_t) - offset, fmt, ap
    );
    va_end(ap);
}
```

Une bibliothèque « thread safe »

(4/4)

La fonction `error_r_sys_set`, quant à elle, s'écrit :

```
void
error_r_sys_set(
    error_r_t err, const char *func, const char *func_sys)
{
    error_r_t err_sys;

    (void) snprintf(
        err, sizeof(error_r_t), "%s():_%s():_(%d,_%s)", func,
        func_sys, errno,
        strerror_r(errno, err_sys, sizeof(error_r_t))
    );
}
```

La fonction `strerror_r` utilisée ici est une fonction spécifique à GNU. Il faudra donc compiler le code avec l'option `D_GNU_SOURCE`.

Une bibliothèque « thread safe »

(4/4)

La fonction `error_r_sys_set`, quant à elle, s'écrit :

```
void
error_r_sys_set(
    error_r_t err, const char *func, const char *func_sys)
{
    error_r_t err_sys;

    (void) snprintf(
        err, sizeof(error_r_t), "%s():_%s():_(%d,_%s)", func,
        func_sys, errno,
        strerror_r(errno, err_sys, sizeof(error_r_t))
    );
}
```

La fonction `strerror_r` utilisée ici est une fonction spécifique à GNU. Il faudra donc compiler le code avec l'option `D_GNU_SOURCE`.

Le fichier **makefile** de la bibliothèque

(1/7)

- On va affiner quelque peu le fichier **makefile** que l'on a vu précédemment dans le cas d'une bibliothèque générique.
- Auparavant, seuls les fichiers d'en-tête étaient dans un catalogue séparé, le catalogue `$(INC_DIR)`.
- On va rajouter deux catalogues : le catalogue `$(SRC_DIR)` pour les fichiers sources et le catalogue `$(TMP_DIR)` pour les fichiers temporaires (`*.o`, `*.d` et `.depend`).
- Ce **découpage** devient nécessaire dès que les fichiers sources (et/ou autres) sont suffisamment nombreux. On peut même envisager, ce que l'on fera pas ici, de découper les catalogues en sous-catalogues.
- Enfin, on ajoutera la cible `dist` permettant de créer une archive compressée (commande `tar zcf`) des sources de la bibliothèque de gestion d'erreurs.

Le fichier **makefile** de la bibliothèque

(1/7)

- On va affiner quelque peu le fichier **makefile** que l'on a vu précédemment dans le cas d'une bibliothèque générique.
- Auparavant, seuls les fichiers d'en-tête étaient dans un catalogue séparé, le catalogue `$(INC_DIR)`.
- On va rajouter deux catalogues : le catalogue `$(SRC_DIR)` pour les fichiers sources et le catalogue `$(TMP_DIR)` pour les fichiers temporaires (`*.o`, `*.d` et `.depend`).
- Ce **découpage** devient nécessaire dès que les fichiers sources (et/ou autres) sont suffisamment nombreux. On peut même envisager, ce que l'on fera pas ici, de découper les catalogues en sous-catalogues.
- Enfin, on ajoutera la cible `dist` permettant de créer une archive compressée (commande `tar zcf`) des sources de la bibliothèque de gestion d'erreurs.

Le fichier **makefile** de la bibliothèque

(1/7)

- On va affiner quelque peu le fichier **makefile** que l'on a vu précédemment dans le cas d'une bibliothèque générique.
- Auparavant, seuls les fichiers d'en-tête étaient dans un catalogue séparé, le catalogue $\$(\text{INC_DIR})$.
- On va rajouter deux catalogues : le catalogue $\$(\text{SRC_DIR})$ pour les fichiers sources et le catalogue $\$(\text{TMP_DIR})$ pour les fichiers temporaires (`*.o`, `*.d` et `.depend`).
- Ce **découpage** devient nécessaire dès que les fichiers sources (et/ou autres) sont suffisamment nombreux. On peut même envisager, ce que l'on fera pas ici, de découper les catalogues en sous-catalogues.
- Enfin, on ajoutera la cible `dist` permettant de créer une archive compressée (commande `tar zcf`) des sources de la bibliothèque de gestion d'erreurs.

Le fichier **makefile** de la bibliothèque

(1/7)

- On va affiner quelque peu le fichier **makefile** que l'on a vu précédemment dans le cas d'une bibliothèque générique.
- Auparavant, seuls les fichiers d'en-tête étaient dans un catalogue séparé, le catalogue `$(INC_DIR)`.
- On va rajouter deux catalogues : le catalogue `$(SRC_DIR)` pour les fichiers sources et le catalogue `$(TMP_DIR)` pour les fichiers temporaires (`*.o`, `*.d` et `.depend`).
- Ce **découpage** devient nécessaire dès que les fichiers sources (et/ou autres) sont suffisamment nombreux. On peut même envisager, ce que l'on fera pas ici, de découper les catalogues en sous-catalogues.
- Enfin, on ajoutera la cible `dist` permettant de créer une archive compressée (commande `tar zcf`) des sources de la bibliothèque de gestion d'erreurs.

Le fichier **makefile** de la bibliothèque

(1/7)

- On va affiner quelque peu le fichier **makefile** que l'on a vu précédemment dans le cas d'une bibliothèque générique.
- Auparavant, seuls les fichiers d'en-tête étaient dans un catalogue séparé, le catalogue $\$(\text{INC_DIR})$.
- On va rajouter deux catalogues : le catalogue $\$(\text{SRC_DIR})$ pour les fichiers sources et le catalogue $\$(\text{TMP_DIR})$ pour les fichiers temporaires (`*.o`, `*.d` et `.depend`).
- Ce **découpage** devient nécessaire dès que les fichiers sources (et/ou autres) sont suffisamment nombreux. On peut même envisager, ce que l'on fera pas ici, de découper les catalogues en sous-catalogues.
- Enfin, on ajoutera la cible `dist` permettant de créer une archive compressée (commande `tar zcf`) des sources de la bibliothèque de gestion d'erreurs.

Le fichier **makefile** de la bibliothèque

(2/7)

```
LIB_NAME = error
LIB_VERS = 1.0

INC_DIR = include
SRC_DIR = src
TMP_DIR = tmp
DIST_DIR = dist

OPTIMIZE = 0 # mettre à 1 pour optimiser le code
# à partir d'ici, il n'y a plus rien à paramétrer
LIB_STATIC = lib$(LIB_NAME).a
LIB_DYNAMIC = lib$(LIB_NAME).so

CC = gcc
CPPFLAGS = -I $(INC_DIR)
CFLAGS = -Wall -Wpointer-arith -fPIC -D_GNU_SOURCE
ifeq ($(OPTIMIZE), 1)
CFLAGS += -O3
endif
```


Le fichier **makefile** de la bibliothèque

(3/7)

```
AR = ar
ARFLAGS = crsU

RM = rm -rf
MKDIR = mkdir -p
LN = ln -v

SRCS = $(wildcard $(SRC_DIR)/*.c)
ifeq (, $(SRCS))
$(error No source files found in '$(SRC_DIR)')
endif

HDRS = $(wildcard $(INC_DIR)/*.h)

OBJS = $(subst $(SRC_DIR), $(TMP_DIR), $(SRCS:.c=.o))
DEPS = $(subst $(SRC_DIR), $(TMP_DIR), $(SRCS:.c=.d))

DIST_NAME = lib$(LIB_NAME)-$(LIB_VERS)
```

Le fichier **makefile** de la bibliothèque

(4/7)

```
.PHONY: clean very_clean dist

all: static dynamic

static: depend $(LIB_STATIC)

dynamic: depend $(LIB_DYNAMIC)

$(LIB_STATIC): $(OBJS)
    @echo \
        "Creating/Updating static library \'$(LIB_STATIC)\'..."
    $(AR) $(ARFLAGS) $@ $?
    @echo "Done."

$(LIB_DYNAMIC): $(OBJS)
    @echo "Creating dynamic library \'$(LIB_DYNAMIC)\'..."
    $(CC) -shared $^ -o $@
    @echo "Done."
```

Le fichier **makefile** de la bibliothèque

(5/7)

```
depend: $(TMP_DIR)/.depend
$(TMP_DIR)/.depend: $(DEPS)
    @echo "Collecting dependencies in file \"$(TMP_DIR)/.depend"
    cat $^ > $@
    @echo "Done."

ifeq ($(TMP_DIR)/.depend, $(wildcard $(TMP_DIR)/.depend))
include $(TMP_DIR)/.depend
endif

$(TMP_DIR)/%.d: $(SRC_DIR)/%.c
    $(CC) $(CPPFLAGS) -MM $< -MT '$$(TMP_DIR)/$*.o -o $@
```

On remarquera que, dans la dernière cible, on a rajouté l'option `-MT`. Effet, par défaut, on produit, par exemple, l'entrée incorrecte :

```
error.o: src/error.c include/error.h
```

Alors qu'avec cette option, l'entrée produite, maintenant correcte, est :

```
$(TMP_DIR)/error.o: src/error.c include/error.h
```

Le fichier **makefile** de la bibliothèque

(5/7)

```
depend: $(TMP_DIR)/.depend
$(TMP_DIR)/.depend: $(DEPS)
    @echo "Collecting dependencies in file \"$(TMP_DIR)/.depend\"
    cat $^ > $@
    @echo "Done."

ifeq ($(TMP_DIR)/.depend, $(wildcard $(TMP_DIR)/.depend))
include $(TMP_DIR)/.depend
endif

$(TMP_DIR)/%.d: $(SRC_DIR)/%.c
    $(CC) $(CPPFLAGS) -MM $< -MT '$$(TMP_DIR)/$*.o -o $@
```

On remarquera que, dans la dernière cible, on a rajouté l'option **-MT**. Effet, par défaut, on produit, par exemple, l'entrée incorrecte :

```
error.o: src/error.c include/error.h
```

Alors qu'avec cette option, l'entrée produite, maintenant correcte, est :

```
$(TMP_DIR)/error.o: src/error.c include/error.h
```

Le fichier **makefile** de la bibliothèque

(5/7)

```
depend: $(TMP_DIR)/.depend
$(TMP_DIR)/.depend: $(DEPS)
    @echo "Collecting dependencies in file \"$(TMP_DIR)/.depend"
    cat $^ > $@
    @echo "Done."

ifeq ($(TMP_DIR)/.depend, $(wildcard $(TMP_DIR)/.depend))
include $(TMP_DIR)/.depend
endif

$(TMP_DIR)/%.d: $(SRC_DIR)/%.c
    $(CC) $(CPPFLAGS) -MM $< -MT '$$(TMP_DIR)/$*.o -o $@
```

On remarquera que, dans la dernière cible, on a rajouté l'option **-MT**. Effet, par défaut, on produit, par exemple, l'entrée incorrecte :

```
error.o: src/error.c include/error.h
```

Alors qu'avec cette option, l'entrée produite, maintenant correcte, est :

```
$(TMP_DIR)/error.o: src/error.c include/error.h
```

Le fichier **makefile** de la bibliothèque

(6/7)

```
$(TMP_DIR)/%.o: $(SRC_DIR)/%.c
    $(CC) $(CPPFLAGS) $(CFLAGS) -c $< -o $@

clean:
    -$(RM) $(OBJS) $(DEPS) $(TMP_DIR)/.depend
    -$(foreach dir, . $(SRC_DIR) $(INC_DIR) $(TMP_DIR), \
        $(RM) $(dir)/*~ $(dir)/\#*\# $(dir)/.\#* $(dir)/core \
            $(dir)/*core; \
    )

very_clean: clean
    -$(RM) $(LIB_STATIC) $(LIB_DYNAMIC) $(DIST_DIR)
```

Ici, dans la cible `clean`, on prend également soin de supprimer les fichiers « autres » dans les catalogues `$(SRC_DIR)`, `$(INC_DIR)` et `$(TMP_DIR)`, en plus du catalogue courant.

Le fichier **makefile** de la bibliothèque

(6/7)

```
$(TMP_DIR)/%.o: $(SRC_DIR)/%.c
    $(CC) $(CPPFLAGS) $(CFLAGS) -c $< -o $@

clean:
    -$(RM) $(OBJS) $(DEPS) $(TMP_DIR)/.depend
    -$(foreach dir, . $(SRC_DIR) $(INC_DIR) $(TMP_DIR), \
        $(RM) $(dir)/*~ $(dir)/\#*\# $(dir)/.\#* $(dir)/core \
            $(dir)/*core; \
    )

very_clean: clean
    -$(RM) $(LIB_STATIC) $(LIB_DYNAMIC) $(DIST_DIR)
```

Ici, dans la cible **clean**, on prend également soin de supprimer les fichiers « autres » dans les catalogues **\$(SRC_DIR)**, **\$(INC_DIR)** et **\$(TMP_DIR)**, en plus du catalogue courant.

Le fichier **makefile** de la bibliothèque

(7/7)

```
# la cible 'dist' pour créer une archive tar compressée
# des sources de la bibliothèque de gestion d'erreurs.
define save_dir =
@echo "->creating$(DIST_DIR)/$(DIST_NAME)$(2) "
@$(MKDIR) $(DIST_DIR)/$(DIST_NAME)$(2)
@echo "->linkingfiles to$(DIST_DIR)/$(DIST_NAME)$(2) "
@$(LN) $(1) $(DIST_DIR)/$(DIST_NAME)$(2)
endef

dist:
@-$(RM) $(DIST_DIR)/$(DIST_NAME)
$(call save_dir, Makefile)
$(call save_dir, $(SRCS),/$(SRC_DIR))
$(call save_dir, $(HDRS),/$(INC_DIR))
@$(MKDIR) $(DIST_DIR)/$(DIST_NAME)/$(TMP_DIR)
@echo "->makingcompressedarchive"
(cd $(DIST_DIR); tar zcf $(DIST_NAME).tar.gz $(DIST_NAME))
@echo "->removing$(DIST_DIR)/$(DIST_NAME)"
@-$(RM) $(DIST_DIR)/$(DIST_NAME)
```


Présentation – Principe de fonctionnement

(1/2)

- Nous avons vu précédemment comment le logiciel `make` permettait d'automatiser la gestion d'un projet (compilation, dépendances, ...).
- Cependant, `make` ne permet pas de prendre en compte les problèmes liés à la **portabilité** comme la présence de telle ou telle bibliothèque, ou la valeur d'une macro dans un fichier en-tête du système, ou encore la nature du compilateur C (est-ce le compilateur GNU CC?).
- Le logiciel `autoconf` a été conçu pour résoudre ce type de problème.
- Le principe est le suivant : le concepteur du projet va éditer un fichier, généralement appelé `configure.ac`, contenant la liste des options et spécificités du **système d'exploitation** et de la **machine d'installation** que l'on veut tester.
- L'exécution de `autoconf` va transformer le fichier `configure.ac` en un *shell script* (`/bin/sh`) de nom : `configure`.

Présentation – Principe de fonctionnement

(1/2)

- Nous avons vu précédemment comment le logiciel `make` permettait d'automatiser la gestion d'un projet (compilation, dépendances, ...).
- Cependant, `make` ne permet pas de prendre en compte les problèmes liés à la **portabilité** comme la présence de telle ou telle bibliothèque, ou la valeur d'une macro dans un fichier en-tête du système, ou encore la nature du compilateur C (est-ce le compilateur GNU CC?).
- Le logiciel `autoconf` a été conçu pour résoudre ce type de problème.
- Le principe est le suivant : le concepteur du projet va éditer un fichier, généralement appelé `configure.ac`, contenant la liste des options et spécificités du **système d'exploitation** et de la **machine d'installation** que l'on veut tester.
- L'exécution de `autoconf` va transformer le fichier `configure.ac` en un *shell script* (`/bin/sh`) de nom : `configure`.

Présentation – Principe de fonctionnement

(1/2)

- Nous avons vu précédemment comment le logiciel `make` permettait d'automatiser la gestion d'un projet (compilation, dépendances, ...).
- Cependant, `make` ne permet pas de prendre en compte les problèmes liés à la **portabilité** comme la présence de telle ou telle bibliothèque, ou la valeur d'une macro dans un fichier en-tête du système, ou encore la nature du compilateur C (est-ce le compilateur GNU CC?).
- Le logiciel `autoconf` a été conçu pour résoudre ce type de problème.
- Le principe est le suivant : le concepteur du projet va éditer un fichier, généralement appelé `configure.ac`, contenant la liste des options et spécificités du **système d'exploitation** et de la **machine d'installation** que l'on veut tester.
- L'exécution de `autoconf` va transformer le fichier `configure.ac` en un *shell script* (`/bin/sh`) de nom : `configure`.

Présentation – Principe de fonctionnement

(1/2)

- Nous avons vu précédemment comment le logiciel `make` permettait d'automatiser la gestion d'un projet (compilation, dépendances, ...).
- Cependant, `make` ne permet pas de prendre en compte les problèmes liés à la **portabilité** comme la présence de telle ou telle bibliothèque, ou la valeur d'une macro dans un fichier en-tête du système, ou encore la nature du compilateur C (est-ce le compilateur GNU CC?).
- Le logiciel `autoconf` a été conçu pour résoudre ce type de problème.
- Le principe est le suivant : le concepteur du projet va éditer un fichier, généralement appelé `configure.ac`, contenant la liste des options et spécificités du **système d'exploitation** et de la **machine d'installation** que l'on veut tester.
- L'exécution de `autoconf` va transformer le fichier `configure.ac` en un *shell script* (`/bin/sh`) de nom : `configure`.

Présentation – Principe de fonctionnement

(1/2)

- Nous avons vu précédemment comment le logiciel `make` permettait d'automatiser la gestion d'un projet (compilation, dépendances, ...).
- Cependant, `make` ne permet pas de prendre en compte les problèmes liés à la **portabilité** comme la présence de telle ou telle bibliothèque, ou la valeur d'une macro dans un fichier en-tête du système, ou encore la nature du compilateur C (est-ce le compilateur GNU CC?).
- Le logiciel `autoconf` a été conçu pour résoudre ce type de problème.
- Le principe est le suivant : le concepteur du projet va éditer un fichier, généralement appelé `configure.ac`, contenant la liste des options et spécificités du **système d'exploitation** et de la **machine d'installation** que l'on veut tester.
- L'exécution de `autoconf` va transformer le fichier `configure.ac` en un *shell script* (`/bin/sh`) de nom : `configure`.

Présentation – Principe de fonctionnement

(2/2)

- Le script `configure` va à partir de fichiers modèles (`*.in`) créer les fichiers adaptés au système d'exploitation et à la machine d'accueil. Parmi ces fichiers modèles, il y aura le plus souvent `Makefile.in`.
- C'est ce script qui sera distribué par le concepteur du projet. Il devra donc gérer toutes les variantes de *Bourne shell* que l'on rencontre sur les différents systèmes d'exploitation (ce qui explique en partie sa taille : ≈ 3700 lignes pour un fichier `configure.ac` minimal).
- Le fichier `configure.ac`, qui, lui, peut être absent de la distribution source, est écrit en langage `m4`.
- Le langage `m4` est un outil de traitement de macros universel. Il a été développé à l'origine par *B. Kernighan* et *D. Ritchie* (qui sont également les concepteurs du langage `C`) en 1977.
- Les macros définies par `autoconf` sont en grand nombre et certaines très spécialisées. Nous n'aborderons donc que les plus fréquemment utilisées.

Présentation – Principe de fonctionnement

(2/2)

- Le script `configure` va à partir de fichiers modèles (`*.in`) créer les fichiers adaptés au système d'exploitation et à la machine d'accueil. Parmi ces fichiers modèles, il y aura le plus souvent `Makefile.in`.
- C'est ce script qui sera distribué par le concepteur du projet. Il devra donc gérer toutes les variantes de *Bourne shell* que l'on rencontre sur les différents systèmes d'exploitation (ce qui explique en partie sa taille : ≈ 3700 lignes pour un fichier `configure.ac` minimal).
- Le fichier `configure.ac`, qui, lui, peut être absent de la distribution source, est écrit en langage `m4`.
- Le langage `m4` est un outil de traitement de macros universel. Il a été développé à l'origine par *B. Kernighan* et *D. Ritchie* (qui sont également les concepteurs du langage `C`) en 1977.
- Les macros définies par `autoconf` sont en grand nombre et certaines très spécialisées. Nous n'aborderons donc que les plus fréquemment utilisées.

Présentation – Principe de fonctionnement

(2/2)

- Le script `configure` va à partir de fichiers modèles (`*.in`) créer les fichiers adaptés au système d'exploitation et à la machine d'accueil. Parmi ces fichiers modèles, il y aura le plus souvent `Makefile.in`.
- C'est ce script qui sera distribué par le concepteur du projet. Il devra donc gérer toutes les variantes de *Bourne shell* que l'on rencontre sur les différents systèmes d'exploitation (ce qui explique en partie sa taille : ≈ 3700 lignes pour un fichier `configure.ac` minimal).
- Le fichier `configure.ac`, qui, lui, peut être absent de la distribution source, est écrit en langage `m4`.
- Le langage `m4` est un outil de traitement de macros universel. Il a été développé à l'origine par *B. Kernighan* et *D. Ritchie* (qui sont également les concepteurs du langage `C`) en 1977.
- Les macros définies par `autoconf` sont en grand nombre et certaines très spécialisées. Nous n'aborderons donc que les plus fréquemment utilisées.

Présentation – Principe de fonctionnement

(2/2)

- Le script `configure` va à partir de fichiers modèles (`*.in`) créer les fichiers adaptés au système d'exploitation et à la machine d'accueil. Parmi ces fichiers modèles, il y aura le plus souvent `Makefile.in`.
- C'est ce script qui sera distribué par le concepteur du projet. Il devra donc gérer toutes les variantes de *Bourne shell* que l'on rencontre sur les différents systèmes d'exploitation (ce qui explique en partie sa taille : ≈ 3700 lignes pour un fichier `configure.ac` minimal).
- Le fichier `configure.ac`, qui, lui, peut être absent de la distribution source, est écrit en langage `m4`.
- Le langage `m4` est un outil de traitement de macros universel. Il a été développé à l'origine par *B. Kernighan* et *D. Ritchie* (qui sont également les concepteurs du langage `C`) en 1977.
- Les macros définies par `autoconf` sont en grand nombre et certaines très spécialisées. Nous n'aborderons donc que les plus fréquemment utilisées.

Présentation – Principe de fonctionnement

(2/2)

- Le script `configure` va à partir de fichiers modèles (`*.in`) créer les fichiers adaptés au système d'exploitation et à la machine d'accueil. Parmi ces fichiers modèles, il y aura le plus souvent `Makefile.in`.
- C'est ce script qui sera distribué par le concepteur du projet. Il devra donc gérer toutes les variantes de *Bourne shell* que l'on rencontre sur les différents systèmes d'exploitation (ce qui explique en partie sa taille : ≈ 3700 lignes pour un fichier `configure.ac` minimal).
- Le fichier `configure.ac`, qui, lui, peut être absent de la distribution source, est écrit en langage `m4`.
- Le langage `m4` est un outil de traitement de macros universel. Il a été développé à l'origine par *B. Kernighan* et *D. Ritchie* (qui sont également les concepteurs du langage `C`) en 1977.
- Les macros définies par `autoconf` sont en grand nombre et certaines très spécialisées. Nous n'aborderons donc que les plus fréquemment utilisées.

L'incontournable « *Hello World!* »

```
dnl le fichier 'configure.ac'
```

```
AC_INIT(hw, 0.1) dnl un autre commentaire  
AC_MSG_NOTICE([Hello, \world!])
```

```
$ autoconf      # on génère 'configure'  
$ ./configure  # on exécute 'configure'  
configure: Hello World!  
$
```

- La macro `AC_INIT` doit obligatoirement être présente dans le fichier `configure.ac`. Son 1er argument est le nom du projet et son 2ème argument est la version du projet. Elle peut prendre également d'autres arguments optionnels.
- Les arguments des macros peuvent être marqués (*quoted*) par '[' et ']', auquel cas rien ne sera interprété par m4 entre ces deux caractères.

L'incontournable « *Hello World* ! »

```
dn1 le fichier 'configure.ac'
```

```
AC_INIT(hw, 0.1) dn1 un autre commentaire
AC_MSG_NOTICE([Hello, \world!])
```

```
$ autoconf      # on génère 'configure'
$ ./configure  # on exécute 'configure'
configure: Hello World!
$
```

- La macro `AC_INIT` doit obligatoirement être présente dans le fichier `configure.ac`. Son 1er argument est le **nom** du projet et son 2ème argument est la **version** du projet. Elle peut prendre également d'autres arguments optionnels.
- Les arguments des macros peuvent être marqués (*quoted*) par '[' et ']', auquel cas rien ne sera interprété par `m4` entre ces deux caractères.

L'incontournable « *Hello World!* »

```
dn1 le fichier 'configure.ac'
```

```
AC_INIT(hw, 0.1) dn1 un autre commentaire
AC_MSG_NOTICE([Hello, \world!])
```

```
$ autoconf      # on génère 'configure'
$ ./configure   # on exécute 'configure'
configure: Hello World!
$
```

- La macro `AC_INIT` doit obligatoirement être présente dans le fichier `configure.ac`. Son 1er argument est le **nom** du projet et son 2ème argument est la **version** du projet. Elle peut prendre également d'autres arguments optionnels.
- Les arguments des macros peuvent être marqués (*quoted*) par '[' et ']', auquel cas rien ne sera interprété par `m4` entre ces deux caractères.

L'incontournable « *Hello World* ! »

```
dn1 le fichier 'configure.ac'
```

```
AC_INIT(hw, 0.1) dn1 un autre commentaire
AC_MSG_NOTICE([Hello, \world!])
```

```
$ autoconf      # on génère 'configure'
$ ./configure   # on exécute 'configure'
configure: Hello World!
$
```

- La macro `AC_INIT` doit obligatoirement être présente dans le fichier `configure.ac`. Son 1er argument est le **nom** du projet et son 2ème argument est la **version** du projet. Elle peut prendre également d'autres arguments optionnels.
- Les arguments des macros peuvent être marqués (*quoted*) par '[' et ']', auquel cas rien ne sera interprété par **m4** entre ces deux caractères.

Quelques éléments de syntaxe

- Toutes les macros **m4** d'**autoconf** ont un nom commençant par **AC_**, **AS_** ou **AT_**.
- On a vu précédemment que l'on pouvait marquer les arguments d'une macro avec les caractères '[' et ']'. Ces caractères ne peuvent être « échappés ».
- Une solution est de doubler ces caractères : **m4** transformera la chaîne `[[blah,␣blah]]` en la chaîne `[blah,␣blah]` en général.
- Parfois, cela dépend des macros, il faudra tripler les caractères '[' et ']'. C'est par exemple le cas avec la macro **AC_MSG_NOTICE** :

```
AC_MSG_NOTICE([[[Hello,␣world!]]])  
$ ./configure  
configure: [Hello World!]
```

Alors qu'avec un simple doublement :

```
AC_MSG_NOTICE([[Hello,␣world!]])  
$ ./configure  
configure: Hello World!
```

Quelques éléments de syntaxe

- Toutes les macros **m4** d'**autoconf** ont un nom commençant par **AC_**, **AS_** ou **AT_**.
- On a vu précédemment que l'on pouvait marquer les arguments d'une macro avec les caractères '[' et ']'. Ces caractères ne peuvent être « échappés ».
- Une solution est de doubler ces caractères : **m4** transformera la chaîne `[[blah,␣blah]]` en la chaîne `[blah,␣blah]` en général.
- Parfois, cela dépend des macros, il faudra tripler les caractères '[' et ']'. C'est par exemple le cas avec la macro **AC_MSG_NOTICE** :

```
AC_MSG_NOTICE([[[Hello,␣world!]]])  
$ ./configure  
configure: [Hello World!]
```

Alors qu'avec un simple doublement :

```
AC_MSG_NOTICE([Hello,␣world!])  
$ ./configure  
configure: Hello World!
```


Quelques éléments de syntaxe

- Toutes les macros **m4** d'**autoconf** ont un nom commençant par **AC_**, **AS_** ou **AT_**.
- On a vu précédemment que l'on pouvait marquer les arguments d'une macro avec les caractères '[' et ']'. Ces caractères ne peuvent être « échappés ».
- Une solution est de doubler ces caractères : **m4** transformera la chaîne **[[blah,␣blah]]** en la chaîne **[blah,␣blah]** **en général**.
- Parfois, cela dépend des macros, il faudra tripler les caractères '[' et ']'. C'est par exemple le cas avec la macro **AC_MSG_NOTICE** :

```
AC_MSG_NOTICE([[[Hello,␣world!]]])  
$ ./configure  
configure: [Hello World!]
```

Alors qu'avec un simple doublement :

```
AC_MSG_NOTICE([[Hello,␣world!]])  
$ ./configure  
configure: Hello World!
```

Quelques éléments de syntaxe

- Toutes les macros **m4** d'**autoconf** ont un nom commençant par **AC_**, **AS_** ou **AT_**.
- On a vu précédemment que l'on pouvait marquer les arguments d'une macro avec les caractères '[' et ']'. Ces caractères ne peuvent être « échappés ».
- Une solution est de doubler ces caractères : **m4** transformera la chaîne `[[blah, \blah]]` en la chaîne `[blah, \blah]` **en général**.
- Parfois, cela dépend des macros, il faudra tripler les caractères '[' et ']'. C'est par exemple le cas avec la macro **AC_MSG_NOTICE** :

```
AC_MSG_NOTICE([[[Hello, \world!]]])
$ ./configure
configure: [Hello World!]
```

Alors qu'avec un simple doublement :

```
AC_MSG_NOTICE([[[Hello, \world!]])
$ ./configure
configure: Hello World!
```

Un premier exemple

(1/2)

```
dnl fichier 'configure.ac'      # fichier 'Makefile.in'

dnl nom, version                PROJ_NAME = @PACKAGE_NAME@
AC_INIT(projet, 1.0)           PROJ_VERS = @PACKAGE_VERSION@
                                CC = @CC@
dnl langage C                   CFLAGS = -Wall -O3
AC_LANG(C)

                                prog: prog.o
                                $(CC) -o $@ $~

dnl compilateur C              .PHONY: clean
AC_PROG_CC                     clean:

dnl on génère 'Makefile'       -rm -rf *.o
AC_OUTPUT(Makefile)
```

On génère tout d'abord le script `configure`, puis on exécute ce script :

```
$ autoconf; ./configure
checking for gcc... gcc
...
config.status: creating Makefile
```

Un premier exemple

(1/2)

```
dn1 fichier 'configure.ac'      # fichier 'Makefile.in'

dn1 nom, version                PROJ_NAME = @PACKAGE_NAME@
AC_INIT(projet, 1.0)            PROJ_VERS = @PACKAGE_VERSION@
                                CC = @CC@
dn1 langage C                   CFLAGS = -Wall -O3
AC_LANG(C)

                                prog: prog.o
dn1 compilateur C               $(CC) -o $@ $~
AC_PROG_CC

                                .PHONY: clean
dn1 on génère 'Makefile'        clean:
AC_OUTPUT(Makefile)             -rm -rf *.o
```

On génère tout d'abord le script `configure`, puis on exécute ce script :

```
$ autoconf; ./configure
checking for gcc... gcc
...
config.status: creating Makefile
```

Un premier exemple

(2/2)

fichier 'Makefile.in'

```
PROJ_NAME = @PACKAGE_NAME@
PROJ_VERS = @PACKAGE_VERSION@
CC = @CC@
CFLAGS = -Wall -O3
```

```
prog: prog.o
    $(CC) -o $@ $^
```

```
.PHONY: clean
clean:
    -rm -rf *.o
```

fichier 'Makefile'

```
PROJ_NAME = projet
PROJ_VERS = 1.0
CC = gcc
CFLAGS = -Wall -O3
```

```
prog: prog.o
    $(CC) -o $@ $^
```

```
.PHONY: clean
clean:
    -rm -rf *.o
```

L'exécution du script `configure` a créé le fichier `Makefile` qui n'est autre qu'une copie du fichier `Makefile.in` dans laquelle on a remplacé les chaînes entre '@' par les valeurs générées par `autoconf` via `configure`. Par exemple, `@PACKAGE_NAME@`, générée par la macro `AC_INIT`, a été remplacé par `projet` (1er argument de `AC_INIT`).

Un premier exemple

(2/2)

fichier 'Makefile.in'

```
PROJ_NAME = @PACKAGE_NAME@
PROJ_VERS = @PACKAGE_VERSION@
CC = @CC@
CFLAGS = -Wall -O3
```

```
prog: prog.o
    $(CC) -o $@ $^
```

```
.PHONY: clean
clean:
    -rm -rf *.o
```

fichier 'Makefile'

```
PROJ_NAME = projet
PROJ_VERS = 1.0
CC = gcc
CFLAGS = -Wall -O3
```

```
prog: prog.o
    $(CC) -o $@ $^
```

```
.PHONY: clean
clean:
    -rm -rf *.o
```

L'exécution du script `configure` a créé le fichier `Makefile` qui n'est autre qu'une copie du fichier `Makefile.in` dans laquelle on a remplacé les chaînes entre '@' par les valeurs générées par `autoconf` via `configure`. Par exemple, `@PACKAGE_NAME@`, générée par la macro `AC_INIT`, a été remplacé par `projet` (1er argument de `AC_INIT`).

Le Bourne shell

(1/2)

- On l'a vu précédemment, `autoconf` produit le *script shell* `configure` à partir du fichier de configuration `configure.ac`.
- Il n'est donc pas surprenant que, outre les macros, le fichier `configure.ac` puisse également contenir du code *shell*.
- Par exemple, si l'on veut s'assurer que l'on dispose bien du compilateur GNU CC :

```
AC_PROG_CC
if test $GCC != "yes"; then
    AC_MSG_FAILURE([ 'GNU CC' est manquant ])
fi
```

- **Attention** : afin de **préserver la portabilité**, il faudra veiller à programmer en *Bourne shell*. Par conséquent, il faudra éviter d'utiliser les extensions ou spécificités du *shell* du système d'exploitation sur lequel on développe. Voir à ce propos le lien [Portable Shell Programming](#)

Le Bourne shell

(1/2)

- On l'a vu précédemment, `autoconf` produit le *script shell* `configure` à partir du fichier de configuration `configure.ac`.
- Il n'est donc pas surprenant que, outre les macros, le fichier `configure.ac` puisse également contenir du code *shell*.
- Par exemple, si l'on veut s'assurer que l'on dispose bien du compilateur GNU CC :

```
AC_PROG_CC
if test $GCC != "yes"; then
    AC_MSG_FAILURE(['GNU CC est manquant'])
fi
```

- **Attention** : afin de **préserver la portabilité**, il faudra veiller à programmer en *Bourne shell*. Par conséquent, il faudra éviter d'utiliser les extensions ou spécificités du *shell* du système d'exploitation sur lequel on développe. Voir à ce propos le lien [Portable Shell Programming](#)

Le Bourne shell

(1/2)

- On l'a vu précédemment, `autoconf` produit le *script shell* `configure` à partir du fichier de configuration `configure.ac`.
- Il n'est donc pas surprenant que, outre les macros, le fichier `configure.ac` puisse également contenir du code *shell*.
- Par exemple, si l'on veut s'assurer que l'on dispose bien du compilateur GNU CC :

```
AC_PROG_CC
if test $GCC != "yes"; then
    AC_MSG_FAILURE(['GNU CC', 'est', 'manquant'])
fi
```

- **Attention** : afin de **préserver la portabilité**, il faudra veiller à programmer en *Bourne shell*. Par conséquent, il faudra éviter d'utiliser les extensions ou spécificités du *shell* du système d'exploitation sur lequel on développe. Voir à ce propos le lien [Portable Shell Programming](#)

Le Bourne shell

(1/2)

- On l'a vu précédemment, `autoconf` produit le *script shell* `configure` à partir du fichier de configuration `configure.ac`.
- Il n'est donc pas surprenant que, outre les macros, le fichier `configure.ac` puisse également contenir du code *shell*.
- Par exemple, si l'on veut s'assurer que l'on dispose bien du compilateur GNU CC :

```
AC_PROG_CC
if test $GCC != "yes"; then
    AC_MSG_FAILURE(['GNU CC est manquant'])
fi
```

- **Attention** : afin de **préserver la portabilité**, il faudra veiller à programmer en *Bourne shell*. Par conséquent, il faudra éviter d'utiliser les extensions ou spécificités du *shell* du système d'exploitation sur lequel on développe. Voir à ce propos le lien [Portable Shell Programming](#)

Le Bourne shell

(2/2)

Un autre exemple, plus complexe, où l'on essaie de déterminer si on dispose de la commande GNU make :

```
ac_make_command=""
for a in "$MAKE" make gmake gnumake; do
    test -z "$a" && continue
    if ( sh -c "$a --version" 2> /dev/null | grep GNU > \
        /dev/null ); then
        ac_make_command=$a;
        break;
    fi
done
if test -z $ac_make_command; then
    AC_MSG_FAILURE(['GNU_ make 'est_manquant])
fi
AC_SUBST(MAKE)
MAKE=$ac_make_command
```

On a utilisé ici la macro `AC_SUBST` afin d'exporter une variable du *shell*.

Le Bourne shell

(2/2)

Un autre exemple, plus complexe, où l'on essaie de déterminer si on dispose de la commande GNU make :

```
ac_make_command=""
for a in "$MAKE" make gmake gnumake; do
    test -z "$a" && continue
    if ( sh -c "$a --version" 2> /dev/null | grep GNU > \
        /dev/null ); then
        ac_make_command=$a;
        break;
    fi
done
if test -z $ac_make_command; then
    AC_MSG_FAILURE(['GNU_␣make'␣est␣manquant])
fi
AC_SUBST(MAKE)
MAKE=$ac_make_command
```

On a utilisé ici la macro `AC_SUBST` afin d'exporter une variable du *shell*.

Le Bourne shell

(2/2)

Un autre exemple, plus complexe, où l'on essaie de déterminer si on dispose de la commande GNU make :

```
ac_make_command=""
for a in "$MAKE" make gmake gnumake; do
    test -z "$a" && continue
    if ( sh -c "$a --version" 2> /dev/null | grep GNU > \
        /dev/null ); then
        ac_make_command=$a;
        break;
    fi
done
if test -z $ac_make_command; then
    AC_MSG_FAILURE(['GNU_␣make'␣est␣manquant])
fi
AC_SUBST(MAKE)
MAKE=$ac_make_command
```

On a utilisé ici la macro `AC_SUBST` afin d'exporter une variable du *shell*.

Le fichier en-tête générique

(1/5)

- La macro `AC_CONFIG_HEADERS` permet de créer un fichier en-tête contenant un certain nombre de variables générées par `autoconf`.
- Elle s'utilise comme suit :

```
AC_CONFIG_HEADERS(config.h)
AH_TOP([      dnl macro 'autoheader'
#ifndef CONFIG_H
#define CONFIG_H
])
AH_BOTTOM([  dnl macro 'autoheader'
#endif /* CONFIG_H */
])
```

- Cette macro est traditionnellement placée juste après la macro `AC_INIT`. Supposons maintenant que cette dernière ait été initialisée comme suit :

```
AC_INIT(projet, 0.1, ylg@irif.fr, projet-0.1.tgz)
```

Le fichier en-tête générique

(1/5)

- La macro `AC_CONFIG_HEADERS` permet de créer un fichier en-tête contenant un certain nombre de variables générées par `autoconf`.
- Elle s'utilise comme suit :

```
AC_CONFIG_HEADERS(config.h)
AH_TOP([      dnl macro 'autoheader'
#ifndef CONFIG_H
#define CONFIG_H
])
AH_BOTTOM([  dnl macro 'autoheader'
#endif /* CONFIG_H */
])
```

- Cette macro est traditionnellement placée juste après la macro `AC_INIT`. Supposons maintenant que cette dernière ait été initialisée comme suit :

```
AC_INIT(projet, 0.1, ylg@irif.fr, projet-0.1.tgz)
```

Le fichier en-tête générique

(1/5)

- La macro `AC_CONFIG_HEADERS` permet de créer un fichier en-tête contenant un certain nombre de variables générées par `autoconf`.
- Elle s'utilise comme suit :

```
AC_CONFIG_HEADERS(config.h)
AH_TOP([      dnl macro 'autoheader'
#ifndef CONFIG_H
#define CONFIG_H
])
AH_BOTTOM([  dnl macro 'autoheader'
#endif /* CONFIG_H */
])
```

- Cette macro est traditionnellement placée juste après la macro `AC_INIT`. Supposons maintenant que cette dernière ait été initialisée comme suit :

```
AC_INIT(projet, 0.1, ylg@irif.fr, projet-0.1.tgz)
```


Le fichier en-tête générique

(2/5)

Le fichier `config.h` ainsi généré aura l'allure suivante :

```
/* config.h. Generated from config.h.in by configure. */
/* config.h.in. Generated from configure.ac by autoheader. */

#ifndef CONFIG_H
#define CONFIG_H

/* Define to the address where bug reports for this package should be sent. */
#define PACKAGE_BUGREPORT "ylg@irif.fr"

/* Define to the full name of this package. */
#define PACKAGE_NAME "projet"

/* Define to the full name and version of this package. */
#define PACKAGE_STRING "projet_0.1"

/* Define to the one symbol short name of this package. */
#define PACKAGE_TARNAME "projet-0.1.tgz"

/* Define to the home page for this package. */
#define PACKAGE_URL ""

/* Define to the version of this package. */
#define PACKAGE_VERSION "0.1"

#endif /* CONFIG_H */
```

Le fichier en-tête générique

(3/5)

- On peut également définir ses propres variables :

```
AC_DEFINE(  
    DEBUG, 1, [Define to 1 for debug mode]  
)
```

- Ceci aura pour effet d'ajouter les deux lignes suivantes dans le fichier généré `config.h` :

```
/* Define to 1 for debug mode */  
#define DEBUG 1
```

- Attention** : la macro `AC_DEFINE` ne peut s'utiliser que conjointement avec la macro `AC_CONFIG_HEADERS`.
- Nous allons voir maintenant un usage moins immédiat de cette macro, usage qui nous permettra de découvrir de nouvelles macros.

Le fichier en-tête générique

(3/5)

- On peut également définir ses propres variables :

```
AC_DEFINE(  
    DEBUG, 1, [Define to 1 for debug mode]  
)
```

- Ceci aura pour effet d'ajouter les deux lignes suivantes dans le fichier généré `config.h` :

```
/* Define to 1 for debug mode */  
#define DEBUG 1
```

- **Attention** : la macro `AC_DEFINE` ne peut s'utiliser que conjointement avec la macro `AC_CONFIG_HEADERS`.
- Nous allons voir maintenant un usage moins immédiat de cette macro, usage qui nous permettra de découvrir de nouvelles macros.

Le fichier en-tête générique

(3/5)

- On peut également définir ses propres variables :

```
AC_DEFINE(  
    DEBUG, 1, [Define to 1 for debug mode]  
)
```

- Ceci aura pour effet d'ajouter les deux lignes suivantes dans le fichier généré `config.h` :

```
/* Define to 1 for debug mode */  
#define DEBUG 1
```

- Attention** : la macro `AC_DEFINE` ne peut s'utiliser que conjointement avec la macro `AC_CONFIG_HEADERS`.
- Nous allons voir maintenant un usage moins immédiat de cette macro, usage qui nous permettra de découvrir de nouvelles macros.

Le fichier en-tête générique

(3/5)

- On peut également définir ses propres variables :

```
AC_DEFINE(  
    DEBUG, 1, [Define to 1 for debug mode]  
)
```

- Ceci aura pour effet d'ajouter les deux lignes suivantes dans le fichier généré `config.h` :

```
/* Define to 1 for debug mode */  
#define DEBUG 1
```

- Attention** : la macro `AC_DEFINE` ne peut s'utiliser que conjointement avec la macro `AC_CONFIG_HEADERS`.
- Nous allons voir maintenant un usage moins immédiat de cette macro, usage qui nous permettra de découvrir de nouvelles macros.

Le fichier en-tête générique

(4/5)

- Supposons que notre projet ait besoin de savoir si l'option de l'API *socket* `SO_REUSEPORT` est disponible sur le système d'exploitation.
- Une solution serait d'essayer de compiler un petit programme C adéquat et d'ensuite, si la compilation a réussi, le signaler dans le fichier `config.h`, par exemple sous la forme suivante :

```
/* Define to 1 when have the socket option `SO_REUSEPORT' */  
#define HAVE_SO_REUSEPORT 1
```

- La macro `autoconf` suivante permet justement de faire cela :

```
AC_TRY_COMPILE(  
  en-têtes, corps-de-la-fonction,  
  [action-si-compilation-réussie], dnl arg. optionnel  
  [action-si-échec-compilation]    dnl arg. optionnel  
)
```

Le langage dans lequel seront écrits les arguments « en-têtes » et « corps-de-la-fonction » est le langage déclaré avec la directive `AC_LANG` (ou `AC_LANG_PUSH`).

Le fichier en-tête générique

(4/5)

- Supposons que notre projet ait besoin de savoir si l'option de l'API *socket* `SO_REUSEPORT` est disponible sur le système d'exploitation.
- Une solution serait d'essayer de compiler un petit programme C adéquat et d'ensuite, si la compilation a réussi, le signaler dans le fichier `config.h`, par exemple sous la forme suivante :

```
/* Define to 1 when have the socket option `SO_REUSEPORT' */  
#define HAVE_SO_REUSEPORT 1
```

- La macro `autoconf` suivante permet justement de faire cela :

```
AC_TRY_COMPILE(  
  en-têtes, corps-de-la-fonction,  
  [action-si-compilation-réussie], dnl arg. optionnel  
  [action-si-échec-compilation]    dnl arg. optionnel  
)
```

Le langage dans lequel seront écrits les arguments « en-têtes » et « corps-de-la-fonction » est le langage déclaré avec la directive `AC_LANG` (ou `AC_LANG_PUSH`).

Le fichier en-tête générique

(4/5)

- Supposons que notre projet ait besoin de savoir si l'option de l'API *socket* `SO_REUSEPORT` est disponible sur le système d'exploitation.
- Une solution serait d'essayer de compiler un petit programme C adéquat et d'ensuite, si la compilation a réussi, le signaler dans le fichier `config.h`, par exemple sous la forme suivante :

```
/* Define to 1 when have the socket option `SO_REUSEPORT' */  
#define HAVE_SO_REUSEPORT 1
```

- La macro `autoconf` suivante permet justement de faire cela :

```
AC_TRY_COMPILE(  
  en-têtes, corps-de-la-fonction,  
  [action-si-compilation-réussie], dnl arg. optionnel  
  [action-si-échec-compilation]    dnl arg. optionnel  
)
```

Le langage dans lequel seront écrits les arguments « en-têtes » et « corps-de-la-fonction » est le langage déclaré avec la directive `AC_LANG` (ou `AC_LANG_PUSH`).

Le fichier en-tête générique

(5/5)

```
AC_MSG_CHECKING(for socket option 'SO_REUSEPORT')
AC_TRY_COMPILE(
[
    #include <sys/types.h>
    #include <sys/socket.h>
],
[
    int val = 1, sock = socket(AF_INET, SOCK_DGRAM, 0);
    setsockopt(
        sock, SOL_SOCKET, SO_REUSEPORT, &val, sizeof(val));
],
[
    AC_DEFINE(HAVE_SO_REUSEPORT, 1,
        [Define to 1 when you have option 'SO_REUSEPORT'])
    AC_MSG_RESULT(yes)
],
    AC_MSG_RESULT(no)
)
```

Création de fichiers

- On a souvent besoin de créer, à partir de modèles, d'autres fichiers que le fichier `Makefile`.
- On procédera, par exemple, de la manière suivante :

```
AC_CONFIG_FILES(Makefile, README)
AC_OUTPUT dnl dernière ligne de 'configure.ac'
```

- Le fichier modèle `README.in` a, par exemple, l'allure suivante :

```
Manuel du logiciel @PACKAGE_NAME@ - version @PACKAGE_VERSION@
=====

1. Installation

tar xzf @PACKAGE_TARNAME@
.....
```

- **Note** : la macro `AC_OUTPUT`, ici sans arguments, doit apparaître en toute fin du fichier `configure.ac`.

Création de fichiers

- On a souvent besoin de créer, à partir de modèles, d'autres fichiers que le fichier `Makefile`.
- On procédera, par exemple, de la manière suivante :

```
AC_CONFIG_FILES(Makefile, README)
AC_OUTPUT dnl dernière ligne de 'configure.ac'
```

- Le fichier modèle `README.in` a, par exemple, l'allure suivante :

```
Manuel du logiciel @PACKAGE_NAME@ - version @PACKAGE_VERSION@
=====

1. Installation

tar xzf @PACKAGE_TARNAME@
.....
```

- **Note :** la macro `AC_OUTPUT`, ici sans arguments, doit apparaître en toute fin du fichier `configure.ac`.

Création de fichiers

- On a souvent besoin de créer, à partir de modèles, d'autres fichiers que le fichier `Makefile`.
- On procédera, par exemple, de la manière suivante :

```
AC_CONFIG_FILES(Makefile, README)
AC_OUTPUT dnl dernière ligne de 'configure.ac'
```

- Le fichier modèle `README.in` a, par exemple, l'allure suivante :

```
Manuel du logiciel @PACKAGE_NAME@ - version @PACKAGE_VERSION@
=====
```

1. Installation

```
tar xzf @PACKAGE_TARNAME@
.....
```

- **Note :** la macro `AC_OUTPUT`, ici sans arguments, doit apparaître en toute fin du fichier `configure.ac`.

Création de fichiers

- On a souvent besoin de créer, à partir de modèles, d'autres fichiers que le fichier `Makefile`.
- On procédera, par exemple, de la manière suivante :

```
AC_CONFIG_FILES(Makefile, README)
AC_OUTPUT dnl dernière ligne de 'configure.ac'
```

- Le fichier modèle `README.in` a, par exemple, l'allure suivante :

```
Manuel du logiciel @PACKAGE_NAME@ - version @PACKAGE_VERSION@
=====

1. Installation

tar xzf @PACKAGE_TARNAME@
.....
```

- **Note :** la macro `AC_OUTPUT`, ici sans arguments, doit apparaître en toute fin du fichier `configure.ac`.

Bibliothèques – La macro `AC_CHECK_LIB`

(1/2)

- Si l'on veut s'assurer de la disponibilité de la fonction `fonc` dans la bibliothèque `bib`, on utilisera la macro suivante :

```
AC_CHECK_LIB(  
    bib, func,  
    [action-si-fonction-trouvée],      dnl arg. optionnel  
    [action-si-fonction-non-trouvée],  dnl arg. optionnel  
    [autres-bibliothèques]             dnl arg. optionnel  
)
```

- Si l'argument « `action-si-fonction-trouvée` » n'est pas fourni, l'action par défaut est d'ajouter à la variable `LIBS`, générée par `autoconf`, la chaîne `-lbib` et également d'insérer dans le fichier `config.h` les lignes :

```
/* Define to 1 if you have the `bib' library (-lbib). */  
#define HAVE_LIBBIB 1
```

- Dernier argument de `AC_CHECK_LIB` : bibliothèques additionnelles nécessaires à l'édition de liens réalisé par la macro pendant le test.

Bibliothèques – La macro `AC_CHECK_LIB`

(1/2)

- Si l'on veut s'assurer de la disponibilité de la fonction `func` dans la bibliothèque `bib`, on utilisera la macro suivante :

```
AC_CHECK_LIB(  
    bib, func,  
    [action-si-fonction-trouvée],      dnl arg. optionnel  
    [action-si-fonction-non-trouvée],  dnl arg. optionnel  
    [autres-bibliothèques]             dnl arg. optionnel  
)
```

- Si l'argument « `action-si-fonction-trouvée` » n'est pas fourni, l'action par défaut est d'ajouter à la variable `LIBS`, générée par `autoconf`, la chaîne `-lbib` et également d'insérer dans le fichier `config.h` les lignes :

```
/* Define to 1 if you have the `bib' library (-lbib). */  
#define HAVE_LIBBIB 1
```

- Dernier argument de `AC_CHECK_LIB` : bibliothèques additionnelles nécessaires à l'édition de liens réalisé par la macro pendant le test.

Bibliothèques – La macro `AC_CHECK_LIB`

(1/2)

- Si l'on veut s'assurer de la disponibilité de la fonction `func` dans la bibliothèque `bib`, on utilisera la macro suivante :

```
AC_CHECK_LIB(  
    bib, func,  
    [action-si-fonction-trouvée],      dnl arg. optionnel  
    [action-si-fonction-non-trouvée],  dnl arg. optionnel  
    [autres-bibliothèques]             dnl arg. optionnel  
)
```

- Si l'argument « `action-si-fonction-trouvée` » n'est pas fourni, l'action par défaut est d'ajouter à la variable `LIBS`, générée par `autoconf`, la chaîne `-lbib` et également d'insérer dans le fichier `config.h` les lignes :

```
/* Define to 1 if you have the `bib' library (-lbib). */  
#define HAVE_LIBBIB 1
```

- Dernier argument de `AC_CHECK_LIB` : bibliothèques additionnelles nécessaires à l'édition de liens réalisé par la macro pendant le test.

Bibliothèques – Un exemple

(2/2)

```
AC_CHECK_LIB(  
    rt, clock_gettime,,  
    AC_MSG_ERROR(['clock_gettime()'␣manquante␣dans␣'-lrt'.])  
)  
AC_CHECK_LIB(  
    m, exp,,  
    AC_MSG_ERROR(['exp()'␣manquante␣dans␣'-lm'.])  
)
```

À la variable `LIBS`, `autoconf` rajoutera la chaîne `‘-lm -lrt’`, tandis que `configure` insérera, dans le fichier `config.h`, les lignes :

```
/* Define to 1 if you have the `m' library (-lm). */  
#define HAVE_LIBM 1  
  
/* Define to 1 if you have the `rt' library (-lrt). */  
#define HAVE_LIBRT 1
```

Bibliothèques – Un exemple

(2/2)

```
AC_CHECK_LIB(  
    rt, clock_gettime,,  
    AC_MSG_ERROR(['clock_gettime()'␣manquante␣dans␣'-lrt'.])  
)  
AC_CHECK_LIB(  
    m, exp,,  
    AC_MSG_ERROR(['exp()'␣manquante␣dans␣'-lm'.])  
)
```

À la variable `LIBS`, `autoconf` rajoutera la chaîne `‘-lm -lrt’`, tandis que `configure` insérera, dans le fichier `config.h`, les lignes :

```
/* Define to 1 if you have the `m' library (-lm). */  
#define HAVE_LIBM 1  
  
/* Define to 1 if you have the `rt' library (-lrt). */  
#define HAVE_LIBRT 1
```

Fichiers en-tête – La macro `AC_CHECK_HEADERS`

(1/2)

- Pour vérifier la présence et l'opérabilité d'un ou plusieurs en-têtes, c'est la macro qui suit :

```
AC_CHECK_HEADERS(  
  en-tête [en-tête...],  
  [action-si-en-tête(s)-trouvé(s)],      dnl arg. optionnel  
  [action-si-en-tête(s)-non-trouvé(s)],  dnl arg. optionnel  
  [autres-entêtes]                       dnl arg. optionnel  
)
```

- Ce qui est testé par cette macro dépend de la présence ou non du dernier argument : la macro vérifie, pour chaque en-tête, soit que celui-ci est prétraité (préprocesseur) sans erreurs, soit que le code suivant est compilé sans erreurs :

```
autres-entêtes  
#include <en-tête>
```

- Pour chaque en-tête trouvé, une définition `HAVE_EN_TETE` est ajoutée dans le fichier `config.h`.

Fichiers en-tête – La macro `AC_CHECK_HEADERS`

(1/2)

- Pour vérifier la présence et l'opérabilité d'un ou plusieurs en-têtes, c'est la macro qui suit :

```
AC_CHECK_HEADERS(  
  en-tête [en-tête...],  
  [action-si-en-tête(s)-trouvé(s)],      dnl arg. optionnel  
  [action-si-en-tête(s)-non-trouvé(s)],  dnl arg. optionnel  
  [autres-entêtes]                       dnl arg. optionnel  
)
```

- Ce qui est testé par cette macro dépend de la présence ou non du dernier argument : la macro vérifie, pour chaque en-tête, soit que celui-ci est prétraité (préprocesseur) sans erreurs, soit que le code suivant est compilé sans erreurs :

```
autres-entêtes  
#include <en-tête>
```

- Pour chaque en-tête trouvé, une définition `HAVE_EN_TETE` est ajoutée dans le fichier `config.h`.

Fichiers en-tête – La macro `AC_CHECK_HEADERS`

(1/2)

- Pour vérifier la présence et l'opérabilité d'un ou plusieurs en-têtes, c'est la macro qui suit :

```
AC_CHECK_HEADERS(  
  en-tête [en-tête...],  
  [action-si-en-tête(s)-trouvé(s)],      dnl arg. optionnel  
  [action-si-en-tête(s)-non-trouvé(s)],  dnl arg. optionnel  
  [autres-entêtes]                       dnl arg. optionnel  
)
```

- Ce qui est testé par cette macro dépend de la présence ou non du dernier argument : la macro vérifie, pour chaque en-tête, soit que celui-ci est prétraité (préprocesseur) sans erreurs, soit que le code suivant est compilé sans erreurs :

```
autres-entêtes  
#include <en-tête>
```

- Pour chaque en-tête trouvé, une définition `HAVE_EN_TETE` est ajoutée dans le fichier `config.h`.

Fichiers en-tête – La macro `AC_CHECK_HEADERS`

(2/2)

```
AC_CHECK_HEADERS(  
    libconfig.h linux/uio.h,,  
    AC_MSG_FAILURE(['libconfig.h...' est manquant])  
)
```

```
$ ./configure  
.....  
checking libconfig.h usability... yes  
checking libconfig.h presence... yes  
checking for libconfig.h... yes  
checking linux/uio.h usability... yes  
checking linux/uio.h presence... yes  
checking for linux/uio.h... yes  
.....
```

```
/* Define to 1 if you have the <libconfig.h> header file. */  
#define HAVE_LIBCONFIG_H 1
```

```
/* Define to 1 if you have the <linux/uio.h> header file. */  
#define HAVE_LINUX_UIO_H 1
```

Fichiers en-tête – La macro `AC_CHECK_HEADERS`

(2/2)

```
AC_CHECK_HEADERS(  
    libconfig.h linux/uio.h,,  
    AC_MSG_FAILURE(['libconfig.h...' est manquant])  
)
```

```
$ ./configure  
.....  
checking libconfig.h usability... yes  
checking libconfig.h presence... yes  
checking for libconfig.h... yes  
checking linux/uio.h usability... yes  
checking linux/uio.h presence... yes  
checking for linux/uio.h... yes  
.....
```

```
/* Define to 1 if you have the <libconfig.h> header file. */  
#define HAVE_LIBCONFIG_H 1
```

```
/* Define to 1 if you have the <linux/uio.h> header file. */  
#define HAVE_LINUX_UIO_H 1
```

Fichiers en-tête – La macro `AC_CHECK_HEADERS`

(2/2)

```
AC_CHECK_HEADERS(  
    libconfig.h linux/uio.h,,  
    AC_MSG_FAILURE(['libconfig.h...' est manquant])  
)
```

```
$ ./configure  
.....  
checking libconfig.h usability... yes  
checking libconfig.h presence... yes  
checking for libconfig.h... yes  
checking linux/uio.h usability... yes  
checking linux/uio.h presence... yes  
checking for linux/uio.h... yes  
.....
```

```
/* Define to 1 if you have the <libconfig.h> header file. */  
#define HAVE_LIBCONFIG_H 1  
  
/* Define to 1 if you have the <linux/uio.h> header file. */  
#define HAVE_LINUX_UIO_H 1
```


Les options du script `configure`

(1/2)

- Par défaut, le script `configure`, généré par `autoconf` possède déjà un certain nombre d'options (`./configure -h` pour les afficher).
- Il est possible également d'ajouter ses propres options grâce à la macro `AC_ARG_WITH` (on utilisera aussi la macro `AS_HELP_STRING`).
- Reprenons notre exemple `projet`. Supposons maintenant que l'on veuille proposer à l'utilisateur la possibilité de compiler `projet` en mode *debug*.
- On va pour ce faire ajouter l'option `--with-debug-mode` au script `configure`.
- Cet ajout d'option se fera dans le fichier de configuration `configure.ac` à l'aide de la macro `AC_ARG_WITH`.

Les options du script `configure`

(1/2)

- Par défaut, le script `configure`, généré par `autoconf` possède déjà un certain nombre d'options (`./configure -h` pour les afficher).
- Il est possible également d'ajouter ses propres options grâce à la macro `AC_ARG_WITH` (on utilisera aussi la macro `AS_HELP_STRING`).
- Reprenons notre exemple `projet`. Supposons maintenant que l'on veuille proposer à l'utilisateur la possibilité de compiler `projet` en mode *debug*.
- On va pour ce faire ajouter l'option `--with-debug-mode` au script `configure`.
- Cet ajout d'option se fera dans le fichier de configuration `configure.ac` à l'aide de la macro `AC_ARG_WITH`.

Les options du script `configure`

(1/2)

- Par défaut, le script `configure`, généré par `autoconf` possède déjà un certain nombre d'options (`./configure -h` pour les afficher).
- Il est possible également d'ajouter ses propres options grâce à la macro `AC_ARG_WITH` (on utilisera aussi la macro `AS_HELP_STRING`).
- Reprenons notre exemple `projet`. Supposons maintenant que l'on veuille proposer à l'utilisateur la possibilité de compiler `projet` en mode *debug*.
- On va pour ce faire ajouter l'option `--with-debug-mode` au script `configure`.
- Cet ajout d'option se fera dans le fichier de configuration `configure.ac` à l'aide de la macro `AC_ARG_WITH`.

Les options du script `configure`

(1/2)

- Par défaut, le script `configure`, généré par `autoconf` possède déjà un certain nombre d'options (`./configure -h` pour les afficher).
- Il est possible également d'ajouter ses propres options grâce à la macro `AC_ARG_WITH` (on utilisera aussi la macro `AS_HELP_STRING`).
- Reprenons notre exemple `projet`. Supposons maintenant que l'on veuille proposer à l'utilisateur la possibilité de compiler `projet` en mode *debug*.
- On va pour ce faire ajouter l'option `--with-debug-mode` au script `configure`.
- Cet ajout d'option se fera dans le fichier de configuration `configure.ac` à l'aide de la macro `AC_ARG_WITH`.

Les options du script configure

(2/2)

```
AC_MSG_CHECKING([si '--with-debug-mode' est spécifié])
AC_ARG_WITH(debug-mode,
  AS_HELP_STRING(
    [--with-debug-mode[=ARG]],
    ['projet est en mode debug [ARG=yes]]
  ),
  [case $withval in
    no)
      AC_MSG_RESULT(no);;
    *)
      AC_MSG_RESULT(yes)
      AC_DEFINE(DEBUG_MODE, 1,
        [Définir à 1 si 'projet' est en mode debug.])
      ;;
  esac
],
  AC_MSG_RESULT(no)
)
```

Écrire ses propres macros

(1/2)

- On peut aussi écrire ses propres macros et ainsi se constituer sa bibliothèque de macros `autoconf`.
- Cette bibliothèque sera constituée d'un fichier par macro définie, ces fichiers étant installés dans un catalogue généralement nommé `m4`. On signalera ce catalogue dans `configure.ac` en ajoutant l'instruction suivante (juste après `AC_INIT`) : `AC_CONFIG_MACRO_DIR([m4])`.
- La macro permettant de définir de nouvelles macros est :

```
AC_DEFUN([nom-macro], [corps-macro])
```

Les deux arguments de `AC_DEFUN` doivent impérativement être marqués (*quoted*) par '[' et ']'.

- L'usage est que le nom des macros commence par `AX_` (pour `eXtended`).
- On va donner l'exemple d'une macro qui teste si l'on dispose ou pas du compilateur GNU CC.

Écrire ses propres macros

(1/2)

- On peut aussi écrire ses propres macros et ainsi se constituer sa bibliothèque de macros `autoconf`.
- Cette bibliothèque sera constituée d'un fichier par macro définie, ces fichiers étant installés dans un catalogue généralement nommé `m4`. On signalera ce catalogue dans `configure.ac` en ajoutant l'instruction suivante (juste après `AC_INIT`) : `AC_CONFIG_MACRO_DIR([m4])`.
- La macro permettant de définir de nouvelles macros est :

```
AC_DEFUN([nom-macro], [corps-macro])
```

Les deux arguments de `AC_DEFUN` doivent impérativement être marqués (*quoted*) par '[' et ']'.

- L'usage est que le nom des macros commence par `AX_` (pour `eXtended`).
- On va donner l'exemple d'une macro qui teste si l'on dispose ou pas du compilateur GNU CC.

Écrire ses propres macros

(1/2)

- On peut aussi écrire ses propres macros et ainsi se constituer sa bibliothèque de macros `autoconf`.
- Cette bibliothèque sera constituée d'un fichier par macro définie, ces fichiers étant installés dans un catalogue généralement nommé `m4`. On signalera ce catalogue dans `configure.ac` en ajoutant l'instruction suivante (juste après `AC_INIT`) : `AC_CONFIG_MACRO_DIR([m4])`.
- La macro permettant de définir de nouvelles macros est :

```
AC_DEFUN([nom-macro], [corps-macro])
```

Les deux arguments de `AC_DEFUN` doivent impérativement être marqués (*quoted*) par '[' et ']'

- L'usage est que le nom des macros commence par `AX_` (pour `eXtended`).
- On va donner l'exemple d'une macro qui teste si l'on dispose ou pas du compilateur GNU CC.

Écrire ses propres macros

(1/2)

- On peut aussi écrire ses propres macros et ainsi se constituer sa bibliothèque de macros `autoconf`.
- Cette bibliothèque sera constituée d'un fichier par macro définie, ces fichiers étant installés dans un catalogue généralement nommé `m4`. On signalera ce catalogue dans `configure.ac` en ajoutant l'instruction suivante (juste après `AC_INIT`) : `AC_CONFIG_MACRO_DIR([m4])`.
- La macro permettant de définir de nouvelles macros est :

```
AC_DEFUN([nom-macro], [corps-macro])
```

Les deux arguments de `AC_DEFUN` doivent impérativement être marqués (*quoted*) par '[' et ']'.

- L'usage est que le nom des macros commence par `AX_` (pour **eXtended**).
- On va donner l'exemple d'une macro qui teste si l'on dispose ou pas du compilateur GNU CC.

Écrire ses propres macros

(1/2)

- On peut aussi écrire ses propres macros et ainsi se constituer sa bibliothèque de macros `autoconf`.
- Cette bibliothèque sera constituée d'un fichier par macro définie, ces fichiers étant installés dans un catalogue généralement nommé `m4`. On signalera ce catalogue dans `configure.ac` en ajoutant l'instruction suivante (juste après `AC_INIT`) : `AC_CONFIG_MACRO_DIR([m4])`.
- La macro permettant de définir de nouvelles macros est :

```
AC_DEFUN([nom-macro], [corps-macro])
```

Les deux arguments de `AC_DEFUN` doivent impérativement être marqués (*quoted*) par '[' et ']'.

- L'usage est que le nom des macros commence par `AX_` (pour **eXtended**).
- On va donner l'exemple d'une macro qui teste si l'on dispose ou pas du compilateur GNU CC.

Écrire ses propres macros – Un exemple

(2/2)

```
AX_PROG_GNU_CC([action-si-GNU-CC], [action-si-pas-GNU-CC])
```

```
AC_DEFUN([AX_PROG_GNU_CC], [
  AC_REQUIRE([AC_PROG_CC])
  if test x"$GCC" != xyes; then
    ifelse([$2],,
      AC_MSG_ERROR([Le compilateur GNU CC est manquant]),
      [$2]
    )
  else
    ifelse([$1],,
      AC_DEFINE(HAVE_GNU_CC, 1,
        [Définir à 1 si on a le compilateur GNU CC]
      ),
      [$1]
    )
  fi
])
```

Écrire ses propres macros – Un exemple

(2/2)

```
AX_PROG_GNU_CC([action-si-GNU-CC], [action-si-pas-GNU-CC])
```

```
AC_DEFUN([AX_PROG_GNU_CC], [  
  AC_REQUIRE([AC_PROG_CC])  
  if test x"$GCC" != xyes; then  
    ifelse([$2],,  
      AC_MSG_ERROR([Le compilateur GNU CC est manquant]),  
      [$2]  
    )  
  else  
    ifelse([$1],,  
      AC_DEFINE(HAVE_GNU_CC, 1,  
        [Définir à 1 si on a le compilateur GNU CC]  
      ),  
      [$1]  
    )  
  fi  
)
```

Conclusion

- Nous avons survolé le logiciel `autoconf`. Néanmoins, nous en avons abordé les principales fonctionnalités. Nous avons aussi passé en revue les macros les plus courantes (excepté `AC_DEFUN` qui est d'un usage moins fréquent).
- Même si de prime abord, l'écriture du fichier de configuration `configure.ac` peut sembler difficile, il ne faut pas hésiter à utiliser `autoconf` dès que l'on a à traiter un projet quelque peu conséquent dont on veut assurer la **portabilité**.
- Une dernière macro, la macro `AC_PREREQ` permettant de contrôler la version du logiciel `autoconf`. Elle apparaîtra au début du fichier de configuration. Par exemple : `AC_PREREQ(2.69)` imposera que la version d'`autoconf` soit ≥ 2.69 .

Conclusion

- Nous avons survolé le logiciel `autoconf`. Néanmoins, nous en avons abordé les principales fonctionnalités. Nous avons aussi passé en revue les macros les plus courantes (excepté `AC_DEFUN` qui est d'un usage moins fréquent).
- Même si de prime abord, l'écriture du fichier de configuration `configure.ac` peut sembler difficile, il ne faut pas hésiter à utiliser `autoconf` dès que l'on a à traiter un projet quelque peu conséquent dont on veut assurer la **portabilité**.
- Une dernière macro, la macro `AC_PREREQ` permettant de contrôler la version du logiciel `autoconf`. Elle apparaîtra au début du fichier de configuration. Par exemple : `AC_PREREQ(2.69)` imposera que la version d'`autoconf` soit ≥ 2.69 .

Conclusion

- Nous avons survolé le logiciel `autoconf`. Néanmoins, nous en avons abordé les principales fonctionnalités. Nous avons aussi passé en revue les macros les plus courantes (excepté `AC_DEFUN` qui est d'un usage moins fréquent).
- Même si de prime abord, l'écriture du fichier de configuration `configure.ac` peut sembler difficile, il ne faut pas hésiter à utiliser `autoconf` dès que l'on a à traiter un projet quelque peu conséquent dont on veut assurer la **portabilité**.
- Une dernière macro, la macro `AC_PREREQ` permettant de contrôler la version du logiciel `autoconf`. Elle apparaîtra au début du fichier de configuration. Par exemple : `AC_PREREQ(2.69)` imposera que la version d'`autoconf` soit ≥ 2.69 .