INF304

Bases du développement logiciel. modularisation, tests

Documents de Cours/TD

2024–2025

Margaret Hamilton, et le code du logiciel de navigation
qu'elle et son équipe ont produit pour le programme *Apollo*

Table des matières

Présentation de l'UE

1. Sources d'informations

Site web : enseignement.gricad-pages.univ-grenoble-alpes.fr/inf304.

Site moodle (caséine : caseine.org). L'inscription à l'UE sur caséine est obligatoire, et sert notamment à la constitution des groupes et aux rendus.

2. Organisation des TD/TP

Un cours-TD + un TP par semaine.

- 5 premières semaines : TPs « indépendants »
- 4 semaines suivantes : mini-projet
- dernière semaine : soutenances de projet

Les TP et le projet se font en binôme (pas de monôme, un trinôme maximum, sauf circonstances exceptionnelles). Pas de changement de binôme en cours de projet.

3. Évaluation

L'évaluation se base à la fois sur du contrôle continu et sur deux examens ponctuels et porte sur :

- Une évaluation pratique (coef 0,4) :
  - À l'issue de chaque TP, un compte-rendu noté est à déposer sur le site moodle. Attention à la procédure de dépôt sur moodle, il faut s'inscrire dans deux groupes : le groupe «académique» (INM1, INM2, MIN1, etc.), **et** le groupe correspondant au binôme. Un seul membre du binôme dépose les fichiers, mais les deux doivent être inscrits. L'ensemble des TPs compte pour 20% de l'évaluation pratique.
  - Les 80% restants de l'évaluation pratique portent sur la soutenance de projet.
- Une évaluation écrite :
  - Partiel en novembre portant sur les 5 premières semaines (coef 0,2) : écrit de 1h15.
  - Examen final (coef 0,4) : écrit de 2h.

4. Savoirs et savoir-faire à avoir acquis à la fin de l'UE

- Lire, comprendre et écrire un court programme modulaire en C
- Savoir construire et compiler un programme modulaire : découper un programme en fonctions et en modules
- Automatiser la compilation d'un programme : lire, comprendre, utiliser, écrire un Makefile
- Tests fonctionnels et tests de robustesse :
  - Savoir les différencier
  - Comprendre la notion de couverture de test
  - Savoir décrire un jeu de tests fonctionnels pour un programme ou un code donné
  - Savoir décrire un jeu de tests de robustesse pour un programme ou un code donné
- Gestion d'erreurs :
  - Savoir identifier et spécifier ce qu'est une bonne utilisation d'un programme ou d'un paquetage
  - Savoir identifier des utilisations erronées et leurs conséquences sur le comportement d'un programme donné
  - Savoir mettre en œuvre une méthode de gestion des erreurs

    - Savoir tester le comportement d'un programme, notamment par le biais d'un oracle
- Types abstraits :
    - Savoir ce que sont les types abstraits et leurs intérêts
    - Savoir les définir, les implémenter et les manipuler

Environnement de programmation

1. Environnement Unix

Les Travaux Pratiques se font sur le serveur `turing`.

1.1 Connexion depuis Visual Studio

Depuis les machines DLST : lancez Visual Studio, puis cliquez sur le bouton « fenêtre distante » (bouton vert en bas à gauche).

Dans le menu qui s'ouvre en haut de la fenêtre :

— tapez le nom de la machine : `im2ag-turing.univ-grenoble-alpes.fr` puis validez par la touche Entrée.
— Entrez vos logins et mots de passe.

2. Connexion depuis l'extérieur

Depuis l'extérieur de l'UGA, l'adresse de la machine est `im2ag-turing.univ-grenoble-alpes.fr`.

Vous pouvez rentrer cette adresse depuis votre machine personnelle, depuis un client (par exemple depuis Visual Studio en suivant la même procédure que depuis une machine du DLST).

2.1 Caractéristiques de l'environnement

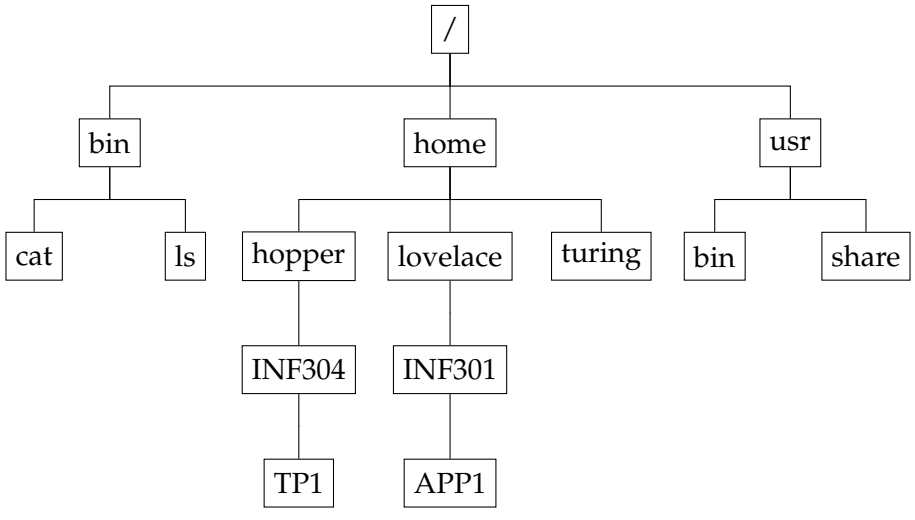
Le serveur `turing` est une machine tournant sous un système d'exploitation de type **Unix**.

Il s'agit d'un environnement **multi-utilisateurs** : vous êtes plusieurs à être connecté·es, et à travailler sur cette même machine (parfois plusieurs centaines d'utilisateurs·trices simultanément !).

Les ressources telles que la mémoire, ou les ressources de calcul (CPU/processeurs) sont partagées entre les utilisateurs connectés. Ces ressources sont **centralisées** : les programmes et fichiers sont ainsi accessibles, quelle que soit la machine cliente depuis laquelle on se connecte.

3. Système de fichiers

Les fichiers sont stockés sous forme **hiérarchique**, représentée par un arbre dont les feuilles désignent les fichiers (programmes, code source, fichiers textes...), les autres nœuds des répertoires[1].



1. NB : dans les systèmes Unix, les « répertoires » sont eux-mêmes considérés comme des « fichiers » particuliers

Dans un système Unix, la racine du système de fichiers est unique, il s'agit du répertoire « / ».

Depuis un terminal, on désigne un fichier par un **chemin** : il s'agit d'une séquence de noms de répertoires, séparés par des «/ », et terminée ou non par un fichier. Un chemin peut être :

— soit **absolu** : il commence alors par « / » (par exemple : `/home/hopper`, `/usr/bin/ls`, etc.)

— soit **relatif** : il commence par un nom de répertoire ou de fichier, et permet de désigner un fichier à partir du **répertoire courant**. Par exemple, dans le système de fichiers représenté ci-dessus, si le répertoire courant est `hopper`, on peut accéder aux fichiers ou répertoires : `INF304/TP1`,`../lovelace`, `../../usr/bin`, etc.

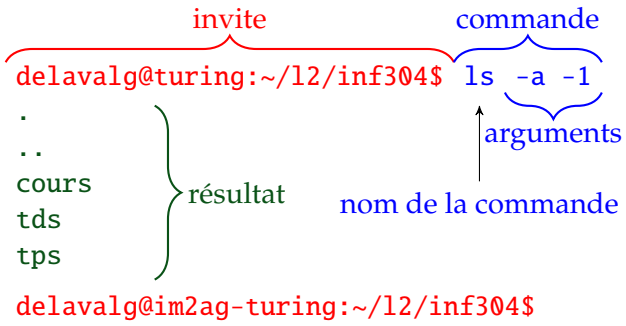
NB : dans un chemin, le répertoire « `.` » désigne le répertoire **courant**, « `..` » le répertoire **parent**.

Chaque utilisateur·trice a un espace personnel (souvent `/home/<`*`login`*`>`, au DLST `/home/`*`initiale`*`/`*`login`*), cet espace personnel a pour alias « ~ ».

4. Interface système

Pour interagir avec le système : compiler, exécuter des programmes, afficher le contenu d'un répertoire, connaître les paramètres d'un fichier, etc., on utilise un **terminal**, qui fournit une *interface en ligne de commande* de type **shell**.

Sur le serveur turing, comme sur la plupart des machines Unix, cette interface se présente sous la forme :



L'**invite** est ce qui est affiché dans le terminal, pour « inviter » l'utilisateur·trice à taper une commande. Dans cette invite, est en général rappelé un certain nombre d'informations utiles : le login, le nom de la machine (utile lorsqu'on travaille à distance sur plusieurs machines différentes), le répertoire courant.

La **commande** est ce qui est entré au clavier par l'utilisateur·trice. Une commande commence par un nom de commande — le *programme* à exécuter, suivi éventuellement d'**arguments**. Dans l'exemple ci-dessus, la commande exécutée est `ls` (pour afficher les fichiers d'un répertoire), « `-a -1` » sont les arguments (`-a` pour afficher les fichiers cachés, `-1` pour afficher 1 fichier par ligne). La commande est terminée par la touche « Entrée ».

Le **résultat** de la commande est affiché directement en-dessous, dans le terminal. Une fois le programme terminé, le terminal affiche à nouveau l'invite, et attend la commande suivante.

5. Redirections d'entrées/sorties

La plupart des programmes ont des *entrées* et des *sorties*.

Les **entrées** peuvent être données en argument (sur la ligne de commande), lues depuis un fichier, entrées depuis une interface, récupérées depuis une interface système ou réseau, etc.

Les **sorties** peuvent être écrites dans un fichier, affichées à l'écran, communiquées via des interfaces systèmes ou réseau, etc.

Les fonctions d'entrées/sorties standard lisent par défaut sur l'**entrée standard** (données entrées au clavier par l'utilisateur·trice), et écrivent par défaut sur la **sortie standard** (affichage à l'écran, sur le terminal comme la commande `ls` de la section précédente).

Ces entrées et sorties standard peuvent être *redirigées*, pour lire les entrées depuis un fichier, ou écrire les sorties

dans un fichier :

— *`commande`* `>` *`fichier`* : la *sortie* de la commande est écrite dans le fichier (au lieu d'être affichée sur le terminal). S'il n'existe pas le fichier est créé ; s'il existait déjà son contenu est écrasé.

  Par exemple :
  `ls -a > mes_fichiers`
  crée un fichier nommé `mes_fichiers`, contenant le résultat de la commande `ls -a` (soit la liste des fichiers du répertoire courant).

— *`commande`* `>>` *`fichier`* : la sortie de la commande est écrite dans le fichier, à la suite du contenu déjà existant qui est conservé (concaténation).

— *`commande`* `<` *`fichier`* : l'*entrée* de la commande est lue depuis le fichier (plutôt que depuis le clavier).

Bien sûr, ces redirections peuvent être combinées.

6. Commandes de base

— **`cd`** ***`chemin`*** (**c**hange **d**irectory) : change le répertoire courant

— **`pwd`** (**p**rint **w**orking **d**irectory) : affiche le répertoire courant

— **`ls`** : affiche les fichiers du répertoire courant, ou du répertoire donné en argument

  — option **`-l`** : affiche toutes les informations pour chaque fichier (droits, propriétaire, groupe, taille, date de modification)

  — option **`-a`** : affiche les fichiers cachés (dont le nom commence par «`.`»)

— **`cat`** ***`fichier`*** : affiche le contenu du fichier

— **`mkdir`** ***`répertoire`*** : création d'un répertoire

— **`rm`** ***`fichier`*** : supprime le fichier (option **`-r`** : suppression récursive ; **`-f`** : sans demande de confirmation)

— **`mv`** ***`fichier1 fichier2`*** : renommage de ***`fichier1`*** en ***`fichier2`***, ou déplacement de ***`fichier1`*** vers ***`fichier2`*** si ***`fichier2`*** est un répertoire

— **`man`** ***`commande`*** : affiche le manuel utilisateur de la commande

Programmation en C

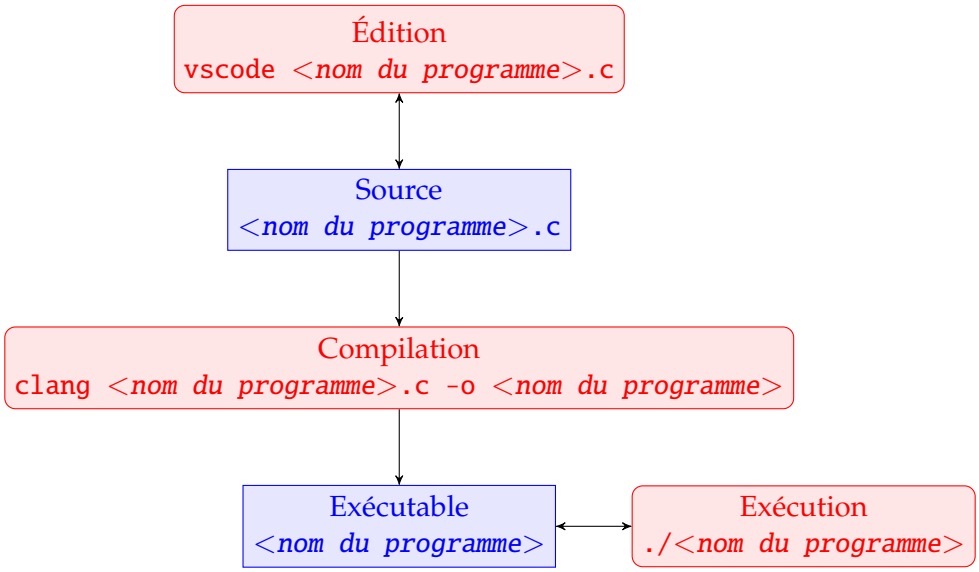
NB : ce chapitre n'est pas un cours complet de C, ni un manuel de référence. Il rappelle juste quelques éléments de base utiles pour les TPs.

1. Introduction au C

Le C est un langage de programmation :

— *impératif* (un programme est une séquence d'*instructions* qui vont avoir un `effet` : modification de l'état interne de la mémoire, lecture, affichage en sortie, etc.) ;
— *compilé* (et non *interpreté*).

Le *cycle de développement* typique sera de cette forme :



Le C étant un langage compilé, le *code source* (fichiers `*.c` ou `*.h`, que l'on va éditer) ne peut être « exécuté » directement : il faut passer par une phase de **compilation**.

La *compilation* permet de traduire du code source en fichier **exécutable** par la machine.

Par exemple, si le code source est dans un fichier nommé `programme.c` :

— on va éditer, modifier le code dans le fichier `programme.c`
— pour compiler ce programme, on va taper dans terminal la commande :

```
clang programme.c -o programme
```

`clang` est le *compilateur C* utilisé dans cette UE[2]

— s'il n'y a pas d'erreur dans le code (auquel cas le compilateur va les afficher), cette commande crée un fichier exécutable `programme`, que l'on peut exécuter (avec éventuellement des arguments et/ou redirections d'entrées/sorties) :

```
./programme argument_1 argument_2
```

2. Structure d'un programme C

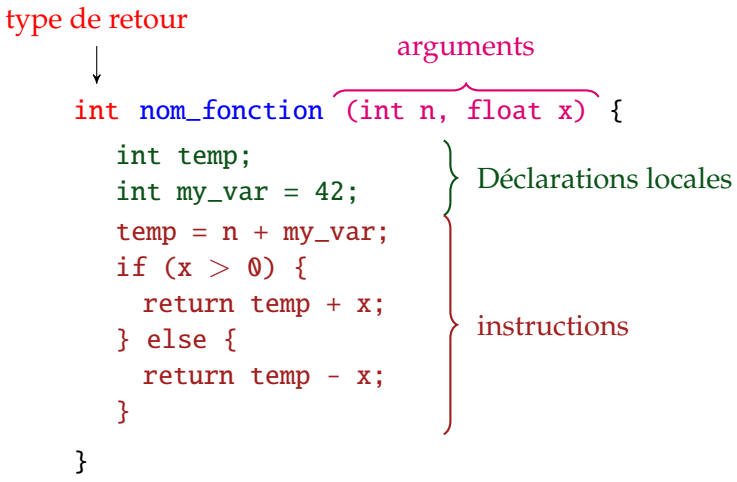
Un programme C est constitué de déclarations de *types*, de *variables* et de *fonctions*. Parmi ces fonctions, une fonction particulière appelée `main` sera la fonction exécutée lors de l'exécution du programme (cette fonction `main` appelle usuellement d'autres fonctions).

2. il existe d'autres compilateurs : `gcc` est le plus connu

2.1 Structure d'une fonction C

Une fonction C est constituée :

— d'un **nom de fonction**, permettant d'identifier la fonction ;
— d'un **type de retour** : le type de la valeur renvoyée par la fonction (ou le type spécial `void` si la fonction ne renvoie pas de valeur — on appelle de telles fonctions des *procédures*) ;
— de **paramètres** : valeurs passées en argument lors de l'appel de la fonction ;
— d'un **corps**, constitué de *déclarations locales* (n'existant qu'à l'intérieur de la fonction), et d'*instructions* qui vont s'exécuter lorsque la fonction va être appelée.



Dans une fonction, l'instruction **`return`** termine la fonction, et permet de définir sa valeur de retour.

2.2 Fonction principale `main`

Un programme C, pour pouvoir être compilé en un exécutable, doit comporter une fonction `main`, de profil :

```
1 int main(int argc, char ** argv) {
2   ...
3 }
```

Il s'agit de la fonction exécutée au lancement du programme. Les paramètres de cette fonction sont :

— `argc` : le nombre d'arguments de la ligne de commande (nom du programme inclus)
— `argv` : tableau de chaînes de caractères contenant les arguments
  — `argv[i]` : `i`$^{\text{e}}$ argument de la ligne de commande
  — `argv[0]` : nom du programme exécuté

La valeur de retour de cette fonction sert de code d'erreur, communiqué au terminal (par convention, 0 pour un programme qui s'exécute sans erreur).

3. Variables et types

3.1 Déclaration de variables

En C, toutes les variables utilisées doivent être *déclarées*. Une variable est déclarée avec son *type* :

```
int longueur;
```

Ci-dessus, la variable `longueur` est déclarée comme étant de type **`int`** (entier). À la suite de cette déclaration, cette variable peut être utilisée (affectée, utilisée dans un calcul, etc.).

Le typage est dit *statique* : le type d'une variable est défini par le programmeur, il ne peut pas changer au cours de l'exécution. Le *type* d'une variable a un *sens* (« sémantique ») ; il définit

— l'ensemble des valeurs possibles pour cette variable : entiers, entiers naturels, flottants, ainsi que l'intervalle de ces valeurs (en pratique, cet intervalle est borné) ;

— les *opérations* possibles sur ces valeurs : opérations arithmétiques, booléennes, opérations d'accès (pour les tableaux ou pointeurs), etc.

On peut *définir* (i.e., donner une valeur) une variable au moment de sa déclaration :

```
int longueur = 10;
```

On peut aussi déclarer plusieurs variables de même type :

```
int longueur, largeur, hauteur;
```

NB : dans une fonction, la déclaration des paramètres (entre parenthèses) tient lieu de déclaration de variables : ces paramètres peuvent être utilisés dans le corps de la fonction.

3.2 Types de base en C

— **`int`** : entier signé (sur 32 bits, entiers dans l'intervalle $[-2^{31}, 2^{31}-1]$).

— **`float`**, **`double`** : nombres flottants (représentation machine des nombres réels).

— **`char`** : caractères codés sur 8 bits.

3.3 Tableaux

Un tableau est groupe d'éléments du même type. Dans un tableau, les éléments sont *ordonnés*, et *indexés* : on peut accéder directement à n'importe quel élément à partir de son *indice*.

NB : En C, **la taille d'un tableau est fixe** : lorsqu'on déclare un tableau, il faut obligatoirement donner sa taille, qui ne changera pas à l'exécution.

Déclaration d'un tableau : un tableau de taille `N`, d'éléments de type `Telem` se déclare :

```
Telem Tab[N];
```

Les indices de ce tableau varient de 0 à $\texttt{N}-1$. Les indices sont de type **`int`**.

Par exemple, la déclaration d'un tableau `T` de 10 entiers :

```
int T[10];
```

Accès à une élément d'un tableau : `Tab[i]`

Exemple :

```
Tab[i] = x; // La valeur d'indice i du tableau Tab devient égale àx
x = Tab[i]; // On affecte àx la valeur d'indice i du tableau Tab
```

Attention : si `i` n'appartient pas à l'intervalle $[0, \texttt{N}-1]$, le comportement n'est pas défini ! Le résultat est soit un arrêt du programme avec le message «*segmentation fault*», soit un accès à une zone mémoire en-dehors du tableau (erreurs difficiles à analyser et à corriger...).

3.4 Types énumérés

Un *type énuméré* permet de définir un ensemble de valeurs par extension (i.e., en donnant la liste des valeurs de l'ensemble).

Par exemple, la déclaration d'un type énuméré `Couleur`, comportant les valeurs `Rouge`, `Jaune`, `Vert` :

```
typedef enum {Rouge, Jaune, Vert} Couleur;

Couleur ma_couleur = Rouge;
```

3.5 Structures

Les *types structures* permettent de rassembler plusieurs valeurs de types (éventuellement) différents.

Déclaration d'un type structure `couple` contenant les champs `x` (de type `T1`) et `y` (de type T2) :

```
typedef struct {
   T1 x;
   T2 y;
} couple;
```

Déclaration d'une variable de type `couple` :

```
couple c;
```

Accès aux valeurs des champs : `c.x`, `c.y`.

4. Structures de contrôle

4.1 Structures conditionnelles

Si/alors/sinon (`if/else`)

Le **`if`** permet d'exécuter certaines instructions de manière *conditionnelle*, en fonction de la valeur d'une *condition* booléenne :

```
if (condition) {
  <instructions>
} else if (condition2) {
  <instructions>
  ...
} else {
  <instructions>
}
```

Les instructions ci-dessous affichent « x est plus petit que y » si la condition « `x < y` » est vraie ; sinon, « x est plus grand que y » est affiché.

```
if (x < y) {
   printf("x est plus petit que y");
} else {
   printf("x est plus grand que y");
}
```

Switch/case

Le switch/case permet d'exécuter des instructions en fonction d'une valeur entière ou énumérée :

```
switch (x) {
  case 0:
    printf("x est nul");
    break;
  case 1:
    printf("x vaut 1");
    break;
  default:
    printf("x vaut autre chose que 0 ou 1");
}
```

Attention : en C, chaque bloc d'instruction pour un cas doit être terminé par l'instruction **break**, qui permet de sortir du **switch** (si l'instruction n'est pas présente, l'exécution se poursuit avec le cas suivant).

4.2 Structures de boucles

Boucle «tant que» (while)

La structure **while** permet de répéter plusieurs fois un bloc d'instructions :

```
while (condition) {
  <instructions>
}
```

Les instructions sont exécutées tant que la `condition` (booléenne) est vraie.

Par exemple, on peut définir toutes les cases d'un tableau (ici comme résultat de l'appel d'une fonction `f`) :

```
int tab[N];
int i = 0;
while (i < N) {
  tab[i] = f(i);
  i = i + 1;
}
```

Boucle « pour » (for)

Une boucle **for** permet d'exécuter un bloc d'instructions un certain nombre de fois, en principe connu au début de l'exécution de la boucle (typiquement, « pour i de 1 à n »).

La syntaxe d'une boucle **for** est :

```
for (<initialisation>, <condition>, <mise àjour>) {
  <instructions>
}
```

Une telle boucle est alors équivalente à :

```
<initialisation>;
while (<condition>) {
  <instructions>
  <mise àjour>;
}
```

Exemple :

```
int tab[N];
int i;
for (i = 0; i < N; i++) {
  tab[i] = f(i);
}
```

5. Pointeurs et allocation dynamique

Les **pointeurs** permettent de manipuler la mémoire de manière plus explicite que les simples déclarations de variables : un *pointeur* est une valeur qui *pointe* sur une autre valeur. On peut considérer qu'un pointeur sur une valeur contient l'« adresse » en mémoire de cette valeur.

La notion de pointeur permet d'effectuer de l'**allocation mémoire dynamique** : c'est-à-dire, de réserver des zones mémoires (valeurs simples, structures, tableaux) qui seront *libérées* explicitement. Ces zones mémoires peuvent persister notamment en-dehors des fonctions où elles sont *allouées* (réservées), contrairement aux déclarations de variables : ces déclarations sont dites « statiques », les zones mémoires associées à ces déclarations n'existent qu'à l'intérieur du bloc (fonction, structure de contrôle) où la déclaration apparaît.

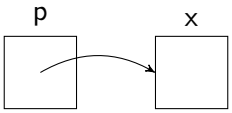
5.1 Vocabulaire

— si `T` est un type, le type `T *` est appelé «*type pointeur de T*». Une valeur de type `T *` est un *pointeur*, pointant sur une valeur de type `T`.

— si `x` est de type `T`, `&x` est l'*adresse* de `x` et est une valeur de type `T *`. Cette valeur peut être affectée à un pointeur de `T`.

— si `p` est de type `T *`, alors la *valeur* pointée par `p` est `*p`, de type `T`. `*p` est appelé le *déréférencement* de `p`.

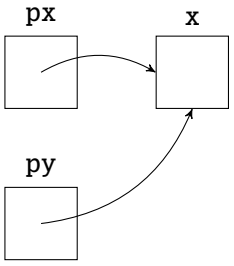
On représente de la manière suivante un pointeur `p` sur une valeur `x` :



5.2 Exemple

Une variable de type `int *` peut pointer sur une variable de type `int`, à l'aide de l'opérateur `&` :

```
int x;
int * px;
int * py;
x = 1;
px = &x;
py = &x;
```



5.3 Pointeurs et structures récursives

Pour la déclaration de types mutuellement récursifs (contenant des pointeurs vers ces types, pour les listes chaînées par exemple), on peut donner un nom à la structure récursive :

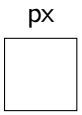
```
// Type cellule pour liste chaînée d'entiers
typedef struct s_cellule {
   int element; // élément courant
   struct s_cellule * suivant; // pointeur vers la cellule suivante
} Cellule;

// Une liste est un pointeur vers la cellule de tête
typedef Cellule * Liste;
```

5.4 Allocation mémoire

Lorsqu'un pointeur est déclaré (et **non défini**) :

```
int * px;
```

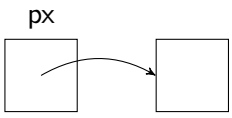


Ce pointeur ne « pointe » alors sur « rien » : il n'existe pas de zone mémoire allouée et associée à ce pointeur.

Si on souhaite *créer* une zone mémoire, on peut faire une emphallocation à l'aide de la fonction `malloc`. Cette fonction prend en paramètre la *taille de la zone mémoire à allouer* (on peut utiliser la fonction **`sizeof`**`(t)` pour obtenir la taille occupée par une valeur de type `t`).

La valeur de retour de `malloc` est l'adresse de la zone mémoire dynamique allouée : cette valeur peut être affectée à un pointeur.

```
px = (int *)malloc(sizeof(int));
```



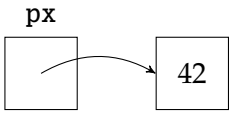
Une zone mémoire contenant un entier est *créée*, mais à son tour *non définie* (comme pour une déclaration).

Contrairement aux déclarations locales, cette zone mémoire existe encore à la sortie du bloc courant. Elle existe jusqu'à sa *libération*.

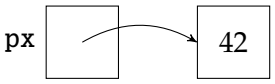
5.5 Accès à la valeur pointée

L'opérateur `*p` permet d'accéder à la *valeur* pointée par `p` (soit pour récupérer cette valeur, soit pour la modifier) :

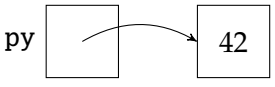
```
*px = 42;
```



NB : il ne faut pas confondre :

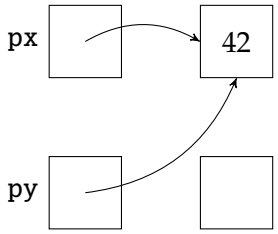



```
*py = *px;
```



et

```
py = px;
```



5.6 Libération de la mémoire allouée

Libérer un bloc mémoire permet de le «rendre» au système, et faire en sorte que la zone mémoire puisse à nouveau être utilisée (à l'occasion par exemple d'une nouvelle demande d'allocation).

Attention : *Tout bloc mémoire alloué* (avec `malloc` ou équivalent) *doit être libéré !*

Attention aux *fuites mémoires* : mémoire allouée et jamais libérée...

Si `p` est un pointeur, *`free(p)`* libère la zone mémoire pointée par `p`. La valeur de `p` doit être une valeur retournée par une primitive d'allocation mémoire : il n'est pas possible de libérer une partie seulement d'un bloc alloué (la moitié d'un tableau par exemple).

Attention : après libération, les valeurs contenues dans une zone mémoire ne doivent plus être accédées (sinon, arrêt du programme avec l'erreur «*segmentation fault*»).

5.7 Pointeurs et paramètres résultats

Les paramètres des fonctions C sont passés *par valeur* : la *valeur* est transmise à la fonction (et non la *référence*).

Pour qu'une fonction puisse *modifier* un paramètre, il faut fournir l'*adresse* de la valeur à modifier, c'est-à-dire un *pointeur* sur cette valeur :

```
void echanger(int * x, int * y) {
  int aux;
  aux = *x;
  *x = *y;
  *y = aux;
}

int a = 42;
int b = 3;
echanger(&a, &b);
```

TD1 — Programmes et modules C

Exercice 1. Lire le programme C fourni en annexe.

1. Dessiner un schéma de la structure de ce programme (modules et programme principal), en indiquant les dépendances entre modules.
2. Quel est le résultat de l'exécution du programme ?
3. Quelles sont les contraintes sur les valeurs en entrée ?

Exercice 2. Modifier le programme pour qu'il affiche la valeur maximum du tableau en entrée. Les contraintes sur les valeurs d'entrées changent-elles avec cette nouvelle spécification ?

A. Programme principal

Programme principal `td1` composé d'un seul fichier `td1.c` :

```c
#include "es_tableau.h"
#include "operations_tableau.h"
#include "type_tableau.h"

int main() {
  tableau_entiers tableau;

  /* Lire le tableau dans le fichier "data1.txt" */
  lire_tableau("data1.txt", &tableau);
  /* Renverser le tableau */
  renverser(&tableau);
  /* Ecrire le tableau renversé dans le fichier "data1_renverse.txt" */
  ecrire_tableau("data1_renverse.txt", &tableau);
}
```

B. Paquetage `type_tableau`

Paquetage `type_tableau` composé du seul fichier « spécification » ou « interface » `type_tableau.h` :

```c
#ifndef _TYPE_TABLEAU_H_
#define _TYPE_TABLEAU_H_

#define TAILLE_MAX 10000

/* Définition du type vecteur_entiers : tableau d'entiers de taille TAILLE_MAX */
typedef int vecteur_entiers[TAILLE_MAX];

/* Structure contenant un tableau (de taille TAILLE_MAX) et un entier
   taille : le nombre d'entiers du tableau */
typedef struct {
  int taille;
  vecteur_entiers tab;
} tableau_entiers;

#endif /* _TYPE_TABLEAU_H_ */
```

C. Paquetage es_tableau

Paquetage es_tableau composé :

— d'un fichier « spécification » ou « interface » es_tableau.h :

```c
#ifndef _ES_TABLEAU_H_
#define _ES_TABLEAU_H_

#include "type_tableau.h"

void lire_tableau(char *nomFichier, tableau_entiers *t);

void ecrire_tableau(char *nomFichier, tableau_entiers *t);

#endif /* _ES_TABLEAU_H_ */
```

— d'un fichier « corps » ou « implémentation » es_tableau.c :

```c
#include "es_tableau.h"
#include <stdio.h>

void lire_tableau(char *nomFichier, tableau_entiers *t) {
  /* Descripteur du fichier d'entrée */
  FILE *fichier;
  int i;

  /* Ouverture du fichier en lecture */
  fichier = fopen(nomFichier, "r");

  /* Lecture de la taille du tableau */
  fscanf(fichier, "%d", &(t->taille));

  /* Lecture des valeurs du tableau */
  for (i = 0; i < t->taille; i++) {
    fscanf(fichier, "%d", &(t->tab[i]));
  }
  fclose(fichier);
}

void ecrire_tableau(char *nomFichier, tableau_entiers *t) {
  /* Descripteur du fichier de sortie */
  FILE *fichier;
  int i;

  /* Créer et ouvrir le fichier en écriture */
  fichier = fopen(nomFichier, "w");

  /* Écrire la taille du tableau dans le fichier */
  fprintf(fichier, "%d\n", t->taille);

  /* Écrire les valeurs du tableau */
  for (i = 0; i < t->taille; i++) {
    fprintf(fichier, "%d\n", t->tab[i]);
  }
  /* Fermeture du fichier */
  fclose(fichier);
}
```

D. Paquetage operations_tableau

Paquetage operations_tableau composé :

— d'un fichier « spécification » ou « interface » operations_tableau.h :

```c
#ifndef _OPERATIONS_TABLEAU_H_
#define _OPERATIONS_TABLEAU_H_

#include "type_tableau.h"

void renverser(tableau_entiers *t);

#endif /* _OPERATIONS_TABLEAU_H_ */
```

— d'un fichier «corps» ou « implémentation » operations_tableau.c :

```c
#include "operations_tableau.h"

/* Permute les entiers p et q
   Données-résultats : p, q : entiers
   Post-condition : les valeurs de p et q ont été permutées
 */
void permuter(int *p, int *q) {
  int aux;

  aux = *p;
  *p = *q;
  *q = aux;
}

/* renverse l'ordre des elements du tableau T
   l'algorithme utilisé est le suivant
   soit n le nombre de valeurs dans le tableau T
   on permute l'élément d'indice 1 de T avec l'élément d'indice n
   on permute l'élément d'indice 2 de T avec l'élément d'indice n-1
   ...
   on permute l'élément d'indice i1 de T avec l'élément d'indice i2,
     avec i1<i2 et i1+i2=n+1
   ...
   jusqu'à la moitié du tableau c'est àdire lorsque i1>=i2
 */
void renverser(tableau_entiers *t) {
  /* Indices de parcours du tableau */
  int i1, i2;

  i1 = 0;
  i2 = t->taille - 1;

  while (i1 < i2) {
    /* Effectuer la permutation des éléments d'indices i1 et i2 de t */
    permuter(&(t->tab[i1]), &(t->tab[i2]));

    /* Incrémenter i1 et décrémenter i2 */
    i1 = i1 + 1;
    i2 = i2 - 1;
  }
}
```

Tests et débogage

1. Pourquoi tester un programme ?[3]

— La taille des «vraies» applications rend l'activité de test obligatoire.

Par exemple : un éditeur de texte, un compilateur, un navigateur Web, un système d'exploitation, etc., sont des programmes ou logiciels composés de plusieurs milliers/millions de lignes de code, écrites par différentes équipes ou par ré-utilisation de code existant.

Ces programmes complexes contiennent nécessairement des erreurs de programmation.

— Interaction avec le matériel, la plate-forme d'exécution

Par exemple : les logiciels embarqués (smartphone, pilote automatique, etc.)

- — une application est en général la combinaison de logiciels et matériels dédiés ;
- — il n'existe pas (encore) de méthode de développement infaillible ;
- — il est nécessaire de tester le produit final dans son ensemble.

— Développement en équipe (voire entre plusieurs équipes !) : personne n'a l'ensemble du code «dans la tête»

— Il n'existe pas de méthode «sûre» de développement logiciel ($\neq$ construction d'un bâtiment, formules de résistance des matériaux, ...) et les erreurs logicielles peuvent coûter très cher — parfois en vies humaines !

Le test reste une des méthodes de validation les plus efficaces.

L'activité de test est **une composante importante** du processus de développement logiciel : environ 30 à 50% du temps de développement. Cette activité *fait partie intégralement du développement*.

Différentes formes de tests sont possibles :

- — tester *a posteriori* (après la phase de codage) : le plus classique, mais pas le plus efficace en coût et temps !
- — tester **au fur et à mesure** : c'est la meilleure solution, car elle permet de détecter les erreurs plus tôt. Par contre, cette méthode nécessite d'être en mesure de tester du code incomplet, de «simuler» les morceaux manquants non encore codés.
- — diriger le développement par le test (TDE, pour "test driven developemnt"). Cette méthode consiste en :
  - — écrire les tests **avant** le code
  - — répéter le cycle «coder-tester-coder-tester-...» tout au long du développement.

En général, pour des gros projets logiciels, l'équipe de testeurs est distincte de l'équipe de développeurs.

Il existe de nombreux outils pour **automatiser** certaines phases du test. Ces outils permettent :

- — un gain de temps
- — moins d'erreurs possibles dans la mise en oeuvre des tests

Remarque importante : le test sert à **rechercher des erreurs éventuelles**, plutôt que de **vérifier le bon fonctionnement**. Cette remarque a deux corollaires :

- — on ne trouvera des erreurs que là où on les a cherchées...
- — l'absence d'erreurs **ne permettra pas de «prouver» que l'application est correcte**...

Remarque n°2 : il n'est pas possible de faire du test s'il n'y a pas de «spécification» disponible (c'est-à-dire, une *description* du comportement attendu du programme). Autrement dit, **avant de commencer à tester il faut savoir ce que l'on teste !**

2. Différentes formes de test pour différents objectifs

On peut tester un programme dans différents buts (liste non exhaustive) :

3. alors que tout étudiant·e en informatique est sensé passer des heures à apprendre à écrire des programmes corrects... après tout si les programmes sont corrects, il n'y a pas besoin de les tester, non ?

Tests fonctionnels (ou tests de conformité) : permet de tester que le programme est «conforme à sa spécification» (sujet des exercices à venir...).

Par exemple : un programme de tri doit trier les éléments (mais pas en inventer ni en supprimer...).

Tests de robustesse : permet de tester que le programme résiste à un environnement d'exécution «dégradé» (pannes matérielles, entrées «hors-spécification»)

Par exemple : un pilote automatique, une interface graphique...

On utilise pour cela un «modèle de fautes», qui décrit les problèmes possibles.

Tests unitaires : tester les différents sous-programmes, paquetages indépendamment. Par exemple :

- une fonction de tri d'un tableau
- un paquetage de lecture d'un fichier image

Tests de non-régression : tester qu'un changement de version, une mise à jour, une correction ne «détruit pas» les fonctionnalités existantes.

Par exemple : changement de version de l'OS, correction d'un bug...

Tests de performance : tester que les performances du programme sont satisfaisante

- temps de réponse
- résistance à la charge
- occupation mémoire (RAM, disque, etc.), bande passante utilisée...
- etc.

Par exemple : un serveur Web, une application smartphone, ...

Tests de sécurité : tester que le programme résiste à un environnement d'exécution «volontairement hostile».

- Éviter que l'application soit rendu inopérante («déni de service») ;
- Éviter que des informations soit modifiées (intégrité), accédées (confidentialité) par des utilisateurs non autorisés à le faire.

Exemples : un serveur Web, un système d'exploitation...

Remarque : dans les 4 derniers cas on ne s'intéresse pas du tout aux fonctionnalités de l'application...

Tests d'intégration : tester une application complète.

3. Le test fonctionnel

Principe du **test fonctionnel** :

- **produire** une suite de tests (ou un jeu de test), selon un *objectif de test* ;
- **exécuter** chacun de ces tests : décider quel est le *verdict* en utilisant un *oracle* ;
- en fonction du résultat, décider de recommencer ou non d'autres tests...

Un **objectif de test** peut être par exemple :

- une fonctionnalité particulière que l'on veut tester

  (ex. : ajouter un utilisateur déjà existant dans une application) ;

— un scénario complet d'exécution

(ex. : lire une image de taille 100x100, appeler une fonction inverse vidéo, la ré-écrire dans le même fichier).

Le **verdict** est généralement de la forme :

— **Pass** (le test a réussi)
— **Fail** (une erreur a été trouvée)
— **Inconclusive** (le test n'a pas pu se dérouler correctement ou n'a pas atteint son objectif)

L'**oracle** peut soit être le testeur lui-même (qui «voit» l'exécution du test), soit un programme qui établit le verdict automatiquement (éventuellement à partir d'un log).

Exemple : vérifier qu'un tableau de 100 ou 1000 éléments est correctement trié peut être fastidieux à faire, «visuellement». L'utilisation d'un oracle permet d'automatiser cette vérification.

Remarque : le rôle de l'oracle est de vérifier que la spécification a bien été respectée.

Toutes ces étapes peuvent être plus ou moins automatisées, mais le plus difficile reste la production de la suite de tests (**choisir** des tests à exécuter).

On s'intéresse dans la suite à deux grandes techniques : le test **«boîte noire»** et le test **«boîte blanche»**.

3.1 Le test fonctionnel en «boîte noire»

Le principe d'un test en *«boîte noire»* est qu'on ne se sert pas du programme source pour générer les tests, mais uniquement de sa spécification. Dans l'idéal, le testeur ou l'équipe de test *ne regarde même pas le code source*, voire dans certains contextes *n'y a pas accès* (contextes industriels critiques) !

Il est alors évidemment nécessaire d'avoir des spécifications précises.

— Intérêt majeur : le testeur n'a pas d'*a priori*, il peut imaginer des scénarios de test originaux.
— Inconvénients : le testeur doit être créatif ; on ne sait pas quelle portion du code a été testée...

Quelques approches possibles :

— **Test aux bornes** : on va tester les valeurs limites des entrées

  Par exemple : trier un tableau vide, inverser une image ne contenant que des pixels noirs, etc.
— **Partitionner l'ensemble d'entrée en fonction du résultat attendu** ; énumérer les différentes classes de résultats possibles et choisir les entrées susceptibles de générer ces résultats

  Par exemple : tester la recherche du max de 3 entiers (le maximum peut être le 1er, le 2^{e} ou le 3^{e} entier saisi).
— **Partitionner l'ensemble des entrées en fonction de sa structure**

  Par exemple : tester le tri de 3 entiers (séquence d'entrée croissante, décroissante, stationnaire, croissante puis décroissante, décroissante puis croissante).
— **Test aléatoire** : on choisit aléatoirement des entrées (**correctes**) (la fonction de génération aléatoire peut éventuellement privilégier certaines entrées).

Remarque : il faut associer à ces approches la notion de «couverture du domaine des entrées», le but étant de couvrir le maximum de cas possibles. Dans le cas général, il n'est pas possible d'être exhaustif : on se base sur une **partition** du domaine d'entrées (c'est une «hypothèse d'uniformité»).

3.2 Le test fonctionnel en «boîte blanche»

Le principe du test en «boîte blanche» est que le testeur utilise le code source du programme à tester pour produire les tests.

Deux approches complémentaires possibles :

— On peut (doit) appliquer la même technique que pour les tests en «boîte noire» (raisonnement à partir de la spécification)

— On peut **EN PLUS** utiliser le code source. On peut alors avoir une approche basée sur la notion de «couverture de code» :
  — couverture des fonctions/sous-programme (chaque fonction doit être exécutée au moins une fois par au moins un test du jeu de tests) ;
  — couverture des instructions (chaque instruction doit être exécutée au moins une fois dans un des tests) ;
  — plus complexe (voire parfois impossible) à mettre en œuvre : couverture des **chemins d'exécution**.

4. Débogage

Le débogage, comme le test, est une activité à part entière du développement logiciel. C'est une activité **inévitable**, puisqu'on part du principe qu'un programme un peu complexe **contient toujours**[4] **des bugs !**

Comme le test toujours, c'est une activité qui ne se fait pas «au hasard». Il existe des *méthodes* et des *outils* de débogage.

Enfin, on ne «débuggue» pas à partir de rien : on part d'un test qui a «échoué».

Remarque importante : ON N'EFFACE JAMAIS UN TEST, QUEL QUE SOIT SON RÉSULTAT !

Une fois un tel test identifié, deux approches sont ensuite possibles.

— **À partir du test** : essayer de «caractériser» le bug en essayant de le reproduire avec différents tests, ou différentes variantes du test qui a échoué (hypothèse : «quelle propriété du test a fait apparaître ce bug ?»)
— **À partir du programme** : identifier la position/la source de l'erreur.
  — Partir d'*hypothèses* sur l'exécution du programme : pour une entrée *donnée* (le test qui a échoué...), on connaît *a priori* le déroulement *attendu* du programme

    Par exemple : valeur d'une variable à une position donnée, nombre d'itérations, chemin d'exécution emprunté, ...
  — On cherche ensuite à quel endroit l'exécution du programme *diverge* de ces hypothèses.

    On peut se servir d'un certain nombre d'outils : assertions, points d'observations,...

    Ou encore mieux : utilisation d'un *débogueur* !

4. TOUJOURS

TD2 — Tests et débogage

Soit le programme C suivant :

```c
#include <stdio.h>

int main() {
  int n;
  int p, d, aux, ec;
  int i;

  printf("Nombre d'entrées : ");
  scanf("%d", &n);

  scanf("%d", &p);
  scanf("%d", &d);
  if (d > p) {
    aux = p;
    p = d;
    d = aux;
  }
  for (i = 3; i <= n; i++) {
    scanf("%d", &ec);
    if (ec > p) {
      d = p;
      p = ec;
    } else if (ec > d) {
      d = ec;
    }
  }
  printf("Valeur de p : %d\n", p);
  printf("Valeur de d : %d\n", d);
}
```

1. Que fait ce programme ?
2. Quel est le domaine de valeurs valides des entrées ?
3. Écrire un jeu de test complet pour ce programme, en justifiant sa construction.
4. Que se passe-t-il si des données en-dehors du domaine de validité sont fournies ?

Compilation séparée, Makefile

1. Modularité et compilation séparée

La plupart des programmes sont composés de plusieurs «modules» ou «paquetages». Ce découpage permet :

— une meilleure structuration du programme ;
— la gestion d'abstractions (types abstraits) ;
— le passage à l'échelle (coût important en temps d'exécution et en mémoire des outils de développement pour les gros programmes).

Cette structuration fait appel à la notion de **modularité** : un *module* peut être développé indépendamment du ou des programmes où il sera utilisé[5].

Cette modularité permet la **compilation séparée** des modules : chaque module peut être compilée de manière indépendante.

Deux étapes sont à distinguer pour la «compilation» de programmes composés de modules :

1. la **compilation** elle-même consiste en la traduction du code source (langage de programmation) en langage machine (assembleur, bytecode, instructions machines binaires) ; c'est cette étape qui peut être modulaire ;
2. l'**édition de lien** : lie plusieurs «*fichiers objets*» (en langage machine) pour produire un *exécutable*. Les «liens» qui sont «édités» sont intuitivement les adresses des fonctions appelées d'un module à l'autre (donc d'un «fichier objet» obtenu par compilation séparée à l'autre).

Schéma de compilation avec un seul module

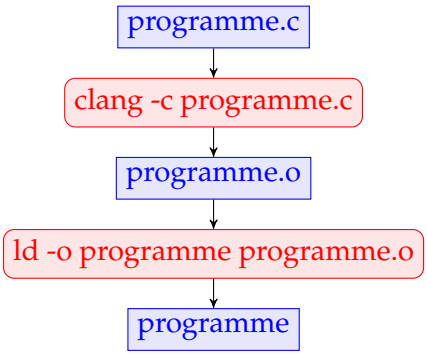
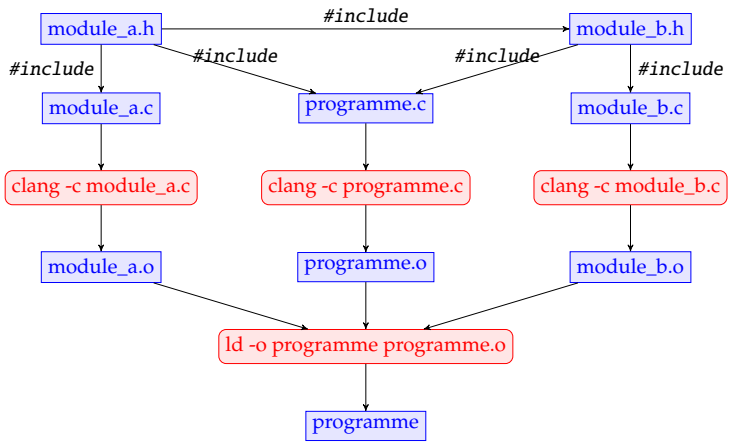


Schéma de compilation complet avec plusieurs modules



NB : `clang` peut aussi être utilisé directement comme éditeur de lien.

2. Makefiles

La gestion de ces différentes étapes peuvent être faites à l'aide d'un fichier `Makefile`. Un fichier `Makefile` est un fichier texte particulier permettant d'automatiser certaines actions (en particulier la création d'un programme exécutable), ainsi que de gérer les **dépendances** entre les différents modules.

Dans un fichier `Makefile`, on va trouver :

— la définition de variables personnalisées,
— la définition de **règles**, chaque règle étant composée d'un **nom** pouvant être le nom d'un fichier à produire, suivie de **dépendances** éventuelles (fichiers déjà existant) et d'une **suite de commandes**, chaque commande devant commencer par une tabulation.

```
cible: dépendance_1 ... dépendance_n
	commande <arg1> ... <argn>
```

5. NB : les *fonctions* aussi sont modulaires, elles sont *compilées indépendamment de leur contexte*

Dans l'exemple ci-dessus, la règle indique que le fichier `cible`, pour être produit, nécessite les fichiers `dépendance_1` à `dépendance_n`. Si ces fichiers n'existent pas ou ne sont pas à jour, les règles de production de ces fichiers vont être examinées avant de produire le fichier `cible`. Ensuite, une fois que ces dépendances ont été traitées, la commande `commande <arg1> ... <argn>` est exécutée pour produire le fichier `cible`.

Pour exécuter une règle du fichier Makefile, on entre la commande

```
make nom_de_la_regle
```

Par défaut, si aucun nom de règle est donnée :

```
make
```

alors la première règle du fichier est exécutée.

Une règle dont le nom correspond à un fichier n'est exécutée que :

- si le fichier n'existe pas ;
- **ou** que l'une des dépendances est un fichier plus récent que le fichier cible existant ;
- **ou** qu'une règle dont une dépendance est un fichier plus récent que le fichier cible est exécutée.

Ceci permet par exemple de mettre à jour un fichier exécutable après avoir modifié un des fichiers sources : le `Makefile` (s'il est bien écrit) permet de ne recréer que les fichiers nécessaires.

Les variables personnalisées peuvent, une fois définies, être utilisées en les incluant entre `$(` et `)` :

```
CC=clang

programme.o: programme.c module.h
    $(CC) -c programme.c
```

Certaines variables dites *internes* peuvent être utilisées dans un `Makefile` :

`$<`	correspond au nom de la première dépendance
`$@`	correspond au nom de la cible (le nom de la règle)
`$^`	correspond à la liste des dépendances

TD3 — Compilation séparée, Makefile

On donne ci-dessous le fichier `Makefile` du TP2 :

```
CC=clang -Wall -g

all: test_tri

test_tri.o: test_tri.c tri.h es_tableau.h type_tableau.h
	$(CC) -c test_tri.c

es_tableau.o: es_tableau.c es_tableau.h type_tableau.h
	$(CC) -c es_tableau.c

type_tableau.o: type_tableau.c type_tableau.h
	$(CC) -c type_tableau.c

tri.o: tri.c tri.h type_tableau.h
	$(CC) -c tri.c

test_tri: test_tri.o es_tableau.o type_tableau.o tri.o
	$(CC) test_tri.o es_tableau.o type_tableau.o tri.o -o test_tri

clean:
	rm -f test_tri *.o
```

Exercice 1. Rappeler la structure du programme.

Exercice 2. Donner la séquence de commandes exécutées :

1. par `make tri.o`
2. par `make`
3. par `make` après modification du fichier `tri.c`
4. par `make` après modification du fichier `tri.h`

Exercice 3. Modifier le `Makefile`, en utilisant des motifs (`%`) et variables internes (`$<`, **`$@`**, `$^`), pour avoir une seule règle de compilation d'un fichier C.

Exercice 4. On souhaite ajouter un module `oracle`, contenant les fonctions permettant de tester la fonction de tri. Décrire les fichiers ajoutés (sans détailler l'implémentation des fonctions), et les modifications à apporter aux fichiers existants.

TD4 — Types abstraits

1. Vers l'abstraction

Considérons une première version du paquetage **type_sequence** définissant un type `SequenceEntiers`. Dans cette version l'utilisateur a un accès direct aux champs de la structure `SequenceEntiers` via l'opérateur **.**.

Exercice 1. Écrire un programme qui réalise les actions suivantes :

1. initialiser une séquence d'entiers `S` de taille 100 avec la valeur 0
2. lire à la console 20 entiers et les ranger dans la séquence à partir de l'indice 0
3. échanger le premier et le dernier entier de la séquence
4. changer la taille de la séquence

```
1
2  #define TAILLE_MAX 10000
3
4  /* structure permettant de manipuler des séquences contenant n entiers, */
5  /* n entre 1 et 10000 */
6  /* les valeurs des éléments de la séquence sont stockés dans le */
7  /* champ tab, des indices 0 àn-1 */
8  typedef struct {
9    int n;
10   int tab[TAILLE_MAX];
11 } SequenceEntiers;
```

Exercice 2. Considérons une deuxième version du paquetage **type_sequence** qui fournit à l'utilisateur des fonctions et procédures lui permettant d'utiliser le type `SequenceEntiers` sans accéder aux champs de la structure. Cette deuxième version est donnée en annexe.

Écrire un programme qui réalise les mêmes actions que précédemment.

NB : nous avons vu qu'il n'est pas nécessaire de connaître la réalisation des procédures et fonctions de l'objet pour utiliser celui-ci.

2. Une pile d'entiers

Exercice 3. Quelles sont les opérations nécessaires à la manipulation d'une pile d'entiers ?

Décrire un paquetage **type_pile** définissant un type `PileEntiers` et les opérations associés.

Écrire un programme de test du paquetage **type_pile** constitué des opérations :

1. Créer une pile p
2. Mettre 4 en sommet de pile
3. Mettre 9 en sommet de pile
4. Dépiler le dernier élément, l'afficher
5. Mettre les éléments $10, 20, 30, \ldots, 80$ en sommet de pile
6. Dépiler et afficher tous les éléments de la pile

A. Annexe : fichiers type_sequence.h et type_sequence.c

```c
#define TAILLE_MAX 10000

/* structure permettant de manipuler des séquences contenant n entiers, */
/* n entre 1 et 10000 */
/* les valeurs des éléments de la séquence sont stockés dans le */
/* champ tab, des indices 0 àn-1 */
typedef struct {
  int n;
  int tab[TAILLE_MAX];
} SequenceEntiers;

/* Opération de construction */

/* Créer une séquence de taille n avec des valeurs nulles */
void creer_sequence(int n, SequenceEntiers *s);

/* Opérations d'accè s */

/* Taille de la séquence */
int nb_elements(SequenceEntiers *s);

/* Retourner le i-è me élément de la séquence s, 1 <= i <= n */
int get_element(SequenceEntiers *s, int i);

/* Opérations de modification */

/* Modifier la taille de la séquence */
/* n >= 0 est la nouvelle taille de la séquence */
/* NB : si la séquence est agrandie, la valeur des éléments ajoutés
   n'est pas définie */
void modifier_taille(SequenceEntiers *s, int n);

/* Changer la valeur du i-è me élément de s avec la valeur v, 1 <= i <= n */
/* La fonction n'est pas définie si i>nb_elements(s) */
void put_element(SequenceEntiers *s, int i, int v);
```

```c
#include "type_sequence.h"

void creer_sequence(int n, SequenceEntiers *s) {
  int i;
  s->n = n;
  for (i = 0; i < n; i++) {
    s->tab[i] = 0;
  }
}

int nb_elements(SequenceEntiers *s) { return s->n; }

int get_element(SequenceEntiers *s, int i) { return s->tab[i - 1]; }

void modifier_taille(SequenceEntiers *s, int n) { s->n = n; }

void put_element(SequenceEntiers *s, int i, int v) { s->tab[i - 1] = v; }
```

TD5 — Gestion d'erreurs, robustesse

On considère le paquetage `type_pile` du TD/TP n° 4, dont voici ci-dessous une implémentation.

```c
#include "type_pile.h"
#include <stdio.h>

/* Créer une pile vide */
void creer_pile(PileEntiers *p) { p->n = 0; }

/* Retourne vrai ssi p est vide */
int est_vide(PileEntiers *p) { return (p->n == 0); }

/* Renvoie l'entier en haut de la pile */
int sommet(PileEntiers *p) { return p->tab[p->n - 1]; }

/* Renvoie le nombre d'éléments dans la pile */
int taille(PileEntiers *p) { return p->n; }

/* Vider la pile p */
void vider(PileEntiers *p) { p->n = 0; }

/* Empiler un entier x */
void empiler(PileEntiers *p, int x) {
  p->tab[p->n] = x;
  p->n = p->n + 1;
}

/* Supprimer et renvoyer l'entier en haut de la pile */
int depiler(PileEntiers *p) {
  p->n = p->n - 1;
  return p->tab[p->n];
}
```

Exercice 1. Donner les préconditions requises pour l'utilisation de chaque fonction de ce paquetage.

Exercice 2. Ces préconditions peuvent ne pas être satisfaites, dans le cas d'une mauvaise utilisation de ce paquetage. Quel sera alors le comportement des fonctions ?

Exercice 3. Modifiez le paquetage pour permettre la gestion des erreurs.

Mini-projet «Curiosity Revolutions»

TD6 — Structure de base, lecture du terrain

NB : ce projet utilise le même environnement que l'APP2 «Curiosity Reloaded» de l'UE INF301 ; cependant les problèmes posés et les exercices demandés sont orthogonaux. Il n'est pas nécessaire d'avoir suivi l'UE INF301 pour réaliser ce projet.

1. Contexte

> *« Liquid water on Mars ! » Maintenant, on en est sûrs, il y a de l'eau liquide sur Mars. Petit problème : le plan d'exploration du robot Curiosity est complètement perturbé, il ne faudrait pas qu'il s'embourbe dans de la boue martienne ou tombe au fond d'une flaque par mégarde...*

De nouveaux programmes ont été développés pour ce robot, pour tenir compte de cette nouvelle situation. Vous allez travailler sur un simulateur permettant de tester ces programmes : il vous faudra notamment vérifier que ce simulateur a bien le comportement attendu.

2. Organisation générale

Le programme sur lequel on travaille est un simulateur du robot Curiosity. Ce robot évolue sur un *terrain*, en exécutant un *programme* écrit dans un langage spécifique.

Exercice 1. Réflechissez à la structure globale de ce simulateur : quels peuvent être les modules du programme ? Quels types et/ou fonctions sont définies dans chacun de ces modules ?

3. Le robot

Le robot va être amené à se déplacer dans le terrain à partir d'une case initiale pour si possible atteindre une case but, ou bien en sortir (la sortie est l'extérieur du terrain).

Le robot est défini par :

- sa position (`x`,`y`) en coordonnées entières,
- son orientation `o` pouvant prendre 4 valeurs possibles :

NORD : ↑	EST : →	SUD : ↓	OUEST : ←

Le type robot peut donc être défini ainsi :

```
typedef enum { Nord, Est, Sud, Ouest } Orientation;

typedef struct {
  int x, y;
  Orientation o;
} Robot;
```

Le robot est capable des actions suivantes :

- avancer,
- tourner d'un quart de tour à droite sur lui-même,
- tourner d'un quart de tour à gauche sur lui-même,
- effectuer une mesure.

Pour ce projet, l'interprétation des programmes des robots (fournis dans un langage spécifique — cf INF301) sera fournie.

4. Le terrain

Un *terrain* est un rectangle composé de cases carrées (`L` en largeur et `H` en hauteur), chaque case pouvant être libre (caractère `'.'`), occupée par de l'eau (caractère '~'), ou occupée par un rocher (caractère `'#'`). La case initiale du robot est une case libre, marquée par le caractère `'C'`.

Préconditions pour le terrain :

— les dimensions `L` (largeur) et `H` (hauteur) doivent être inférieures à une dimension maximale `DIM_MAX`,

— il existe un chemin (formé de cases libres) entre la case centrale et l'extérieur du terrain (la sortie).

Un fichier *terrain* est un fichier composé :

1. d'un entier `L`, la largeur du terrain
2. d'un entier `H`, la hauteur du terrain
3. de `H` lignes de `L` caractères dans l'ensemble {`'#'`,`'.'`,`'~'`,`'C'`}.

Exemple de fichier «terrain» :

```
11
9
#..#......#
...........
....C......
..........#
#..#.......
#........#.
......#....
#..##......
..#..#.....
```

Déclaration du type terrain en C :

```c
typedef enum { LIBRE, EAU, ROCHER } Case;

#define DIM_MAX 256

// indexation utilisée :
// 1er indice : abscisse = colonne (colonne de gauche : abscisse = 0)
// 2è me indice : ordonnée = ligne (ligne du haut : ordonnée = 0)

typedef struct {
  int largeur, hauteur;
  Case tab[DIM_MAX][DIM_MAX];
} Terrain;
```

Exercice 2. Écrire une fonction `lire_terrain`, permettant de lire un terrain dans un fichier (on suppose dans un premier temps que le format du fichier à lire est correct et les préconditions remplies).

Exercice 3. Si le fichier n'existe pas ou est incorrect, quelles erreurs peuvent se produire à l'exécution de la fonction `lire_terrain` ?

Compléter la fonction `lire_terrain` afin de renvoyer une erreur si le format du fichier n'est pas correct.

TD7 — Tests de programmes robots

1. Programmes robots

Le robot Curiosity évolue maintenant sur un terrain de manière autonome, à partir d'un *programme*. Ce programme est une séquence de caractères (`'A'`, `'G'`, `'D'`, `'M'`, etc.), lue dans un fichier. Un paquetage d'*interprétation* d'un programme est fourni. Le but de cette deuxième semaine est de tester de manière systématique cet interprète.

La syntaxe complète du langage de programmation du robot Curiosity est rappelée en annexe.

2. Paquetage `environnement`

L'*environnement* est la composition d'un *terrain* et d'un *robot*. C'est sur cet environnement que vont être effectuées les actions de base du robot : *avancer*, *tourner à gauche*, *tourner à droite* et *effectuer une mesure*. Ces actions seront réalisées par l'interprète du programme.

Voici ci-dessous un extrait de ce paquetage :

```
1  /* Initialise l'environnement envt :
2     - lit le terrain dans le fichier fichier_terrain
3     - initialise le robot : coordonnées initiales lues dans le fichier
4     terrain, orientation initiale vers l'est
5  */
6  erreur_terrain initialise_environnement(Environnement *envt,
7                                          char *fichier_terrain);
8  /* Afficher le terrain avec la position et l'orientation du robot */
9  void afficher_envt(Environnement *envt);
```

3. Paquetage `programme`

Le paquetage `Programme` fournit le type de données et la fonction de lecture d'un programme dans un fichier :

```
1  /* Programme : séquence de commandes */
2  typedef struct {
3    Commande tab[PROG_TAILLE_MAX];
4    int lg;
5  } Programme;
6
7  /* Erreurs de lecture d'un programme */
8  typedef enum {
9    OK_PROGRAMME,
10   ERREUR_FICHIER_PROGRAMME,
11   ERREUR_BLOC_NON_FERME,
12   ERREUR_FERMETURE_BLOC_EXCEDENTAIRE,
13   ERREUR_COMMANDE_INCORRECTE
14 } type_erreur_programme;
15
16 typedef struct {
17   type_erreur_programme type_err;
18   char *ligne; /* Ligne du programme contenant l'erreur */
19   int num_ligne, num_colonne; /* Position de l'erreur dans le fichier */
20 } erreur_programme;
21
22 /* Lecture d'un programme prog dans le fichier nom_fichier */
23 erreur_programme lire_programme(Programme *prog, char *nom_fichier);
```

4. Paquetage interprete

Le paquetage interprete fournit les fonctions d'interprétation d'un programme : initialisation d'un état, exécution d'un pas du programme.

```
1  /* Résultat de l'interprétation */
2  typedef enum {
3    OK_ROBOT, /* Le robot est sur une case libre et le programme n'est pas terminé
4              */
5    SORTIE_ROBOT, /* Le robot est sorti du terrain */
6    ARRET_ROBOT, /* Le programme est terminé */
7    PLOUF_ROBOT, /* Le robot est tombé dans l'eau */
8    CRASH_ROBOT, /* Le robot est rentré dans un rocher */
9    ERREUR_PILE_VIDE, /* Erreur : pile vide */
10   ERREUR_ADRESSAGE, /* Erreur d'adressage : indice de commande incorrect */
11   ERREUR_DIVISION_PAR_ZERO, /* Erreur : tentative de division par 0 */
12 } resultat_inter;
13
14 /* Etat de l'interprè te */
15 typedef struct {
16   /* Program counter : adresse de la prochaine commande àexécuter */
17   int pc;
18   /* Pile de données */
19   PileEntiers stack;
20   /* Pile d'adresses de retour */
21   PileEntiers sp;
22 } etat_inter;
23
24 /* Initialisation de l'état */
25 void init_etat(etat_inter *etat);
26
27 /* Pas d'exécution de l'interprè te : exécute une commande, modifie
28    l'environnement et l'état, renvoie l'état du robot */
29 resultat_inter exec_pas(Programme *prog, Environnement *envt, etat_inter *etat);
```

Exercice 1. Liens de dépendances Donner les liens de dépendances entre ces différents paquetages. Écrire leurs règles Makefile de compilation.

Exercice 2. Exécution simple À l'aide de ces paquetages, écrire un programme prenant en argument deux noms de fichiers (un terrain et un programme), et qui exécute ce programme sur un robot dans ce terrain.

Donner la règle Makefile pour la compilation et l'édition de liens de ce programme.

Exercice 3. Tests fonctionnels Décrire un protocole complet permettant de tester l'interprète.

Exercice 4. Tests de robustesse Donner des exemples de programmes déclenchant une erreur :

— à la lecture du programme ;
— à l'exécution du programme.

A. Langage de programmation du robot Curiosity

Rappel : la syntaxe [a_1 $a_2 \cdots a_n$ `C`] signifie que la commande `C` prend n arguments, récupérés sur la pile (le dernier argument a_n est en haut de la pile). Dans tous les cas, les arguments sont enlevés de la pile avant l'exécution de la commande.

`A` — Avancer le robot d'une case

`G` — Tourner le robot de 90° vers la gauche

`D` — Tourner le robot de 90° vers la droite

n **`M`** — Effectuer une «mesure» :

Valeurs d'entrées		Valeurs de sortie	
0	sur place	0	rien
1	devant	1	eau
2	devant droite	2	rocher
3	droite		
4	derrière droite		
5	derrière		
6	derrière gauche		
7	gauche		
8	devant gauche		

`{` *commandes* **`}`** — Crée un groupe de commandes *{commandes}* et le place sur la pile

cmd **`!`** — Exécute *cmd*

Exemple :

Pile	Programme
vide	`{ A D } { G A } ! !`
`{ A D }`	`{ G A } ! !`
`{ A D } { G A }`	`! !`
`{ A D }`	`G A !`
`{ A D }`	`!`
vide	`A D`
vide	*vide*

Dans cet exemple, Curiosity va d'abord tourner à gauche, puis avancer deux fois, puis tourner à droite.

n V F **`?`** — Instruction conditionnelle : exécute V si $n \neq 0$, et F si $n = 0$.

Exemple :

Pile	Programme
vide	`3 M { G A } { A } 7 M { D A } { A } ? ?`
`0`	`{ G A } { A } 7 M { D A } { A } ? ?`
`0 { G A } { A }`	`7 M { D A } { A } ? ?`
`0 { G A } { A } 2 { D A } { A }`	`? ?`
`0 { G A } { A }`	`D A ?`
`0 { G A } { A }`	`?`
vide	`A`
vide	*vide*

Ce code fait d'abord tourner à droite et avancer Curiosity, puis le fait avancer une nouvelle fois.

A B **`X`** — Échange A et B sur la pile

x y op — avec $op \in \{+, -, *, /\}$: calcule *x op y* et place le résultat sur la pile

$A_n \ldots A_2\ A_1\ n\ x$ **`R`** — Effectue une « rotation » de x pas vers la gauche des n éléments en haut de la pile : le haut de la pile devient $A_{n-x} \ldots A_1\ A_n\ A_{n-1} \ldots A_{n-x+1}$

e **`C`** — Clone *e* sur la pile

cmd n **`B`** — Exécute *cmd*, décrémente n sans enlever `B` du programme si $n > 0$. Sinon enlève *cmd* et n de la pile, et `B` du programme.

e **`I`** — Ignore l'élément en haut de la pile

TD8 — Génération de terrains

1. Motivation

On va maintenant s'intéresser à des programmes-robots plus «complets», indépendants des terrains sur lesquels ils sont utilisés, pour permettre une plus grande autonomie du robot.

Pour tester les programmes-robots, on peut :

— leur fabriquer à la main des terrains particuliers : s'assurer qu'ils sortent de terrains «faciles», mesurer leur performance sur des terrains «difficiles»...
— vouloir les tester sur un grand nombre de terrains pour évaluer leur comportement moyen.

Pour que cette notion de «comportement moyen» ait un sens, on va tester les automates sur des terrains de mêmes caractéristiques : dimensions et densité/nombre d'obstacles.

Pour fabriquer un grand nombre de terrains (de caractéristiques données), on souhaite les générer aléatoirement.

Au final, on pourra comparer les performances de différents programmes-robots sur différents types de terrains : grands avec peu d'obstacles, petits avec beaucoup d'obstacles...

Remarque : on garantit que les terrains ont tous une issue (le programme est fourni — cf. exercice 4).

2. Génération de terrains

Il s'agit donc de fabriquer un fichier de `N` terrains de dimensions `l` et `h`, avec une certaine densité ou avec un certain nombre d'obstacles.

2.1 Génération de séquences pseudo-aléatoires (Rappel du TP2)

On dispose en C de la fonction `int rand(void)` de la librairie `stdlib.h` qui nous renvoie des nombres *pseudo-aléatoires*. Il s'agit d'une séquence de nombres très grande qui semble donc aléatoire mais qui est entièrement définie par une valeur initiale : la *«graine»*. La fonction `void srand(int)` permet d'initialiser cette séquence en fournissant cette «graine». On peut réutiliser la même valeur à chaque exécution (pour reproduire l'expérience dans les mêmes conditions), ou utiliser par exemple la fonction `int time(NULL)` (du paquetage `time.h`, qui renvoie le nombre de secondes écoulées depuis le 1/1/1970), pour obtenir une séquence différente à chaque exécution du programme.

Exercice 1.

— Écrire un programme qui calcule 100 tirages de nombres entre deux naturels a et b donnés.
— Écrire un programme qui calcule 100 tirages d'une pièce pile/face.

Exercice 2. Écrire un programme qui produit un fichier contenant un entier `N` suivi de `N` tirages aléatoires d'un dé à 6 faces.

3. Production d'un ensemble de terrains

Notre problème est de créer un fichier contenant un entier `N` suivi de `N` terrains de dimensions `L` et `H` donnés ayant un certain nombre d'obstacles. Cela revient à reprendre l'exercice 2 en remplaçant «dé à 6 faces» par «terrain de dimensions `L` et `H` avec `X` obstacles».

Le terrain étant quelque chose d'un peu plus compliqué qu'un dé, il peut être pratique de le générer entièrement, c'est-à-dire de calculer une variable de type `terrain`, avant de l'écrire dans le fichier.

Exercice 3. Reprendre la solution de l'exercice 2 et l'adapter à notre problème.

TD9 — Observateurs et vérification dynamique

1. Motivation

On souhaite maintenant vérifier qu'à l'exécution d'un programme, certaines propriétés *dynamiques* soient respectées : par exemple, pour l'interprète, que la fonction «avancer» soit toujours précédée d'une «mesure» par le robot de la case devant lui.

Pour cela, on va utiliser un *observateur*, c'est-à-dire une *instrumentation* du programme de test dédiée à la vérification à l'exécution de cette propriété.

2. Définition de l'observateur

Exercice 1. Donner sous forme d'automate d'états fini la propriété «avant d'avancer, une mesure est toujours effectuée».

Exercice 2.

1. Écrire un exemple de programme-robot *correct*, *accepté* par l'observateur.
2. Écrire un exemple de programme-robot *incorrect*, *rejeté* par l'observateur.
3. Écrire un exemple de programme-robot *correct*, *rejeté* par l'observateur.
4. Écrire un exemple de programme-robot *incorrect*, *accepté* par l'observateur.

3. Implémentation de l'observateur

Exercice 3. Spécifier un paquetage `observateur`, fournissant les types de données et fonctions permettant d'instrumenter un programme. Écrire l'implémentation de ce paquetage pour l'automate défini à l'exercice 1.

Exercice 4. Modifier le paquetage `environnement` (dont la spécification est rappelée en annexe), en utilisant le paquetage `observateur`, pour permettre la vérification à l'exécution de la propriété associée à l'observateur.

A. Spécification du paquetage environnement

```
#ifndef _ENVIRONNEMENT_H_
#define _ENVIRONNEMENT_H_

#include "robot.h"
#include "terrain.h"

/* Environnement : terrain + robot */

typedef struct {
  Robot r;
  Terrain t;
} Environnement;

/* Initialise l'environnement envt :
   - lit le terrain dans le fichier fichier_terrain
   - initialise le robot : coordonnées initiales lues dans le fichier
   terrain, orientation initiale vers l'est
*/
erreur_terrain initialise_environnement(Environnement *envt,
                                  char *fichier_terrain);

/* Résultat d'un déplacement de robot */
typedef enum {
  OK_DEPL, /* Déplacement sur case libre */
  PLOUF, /* Déplacement dans l'eau */
  CRASH, /* Déplacement dans un rocher */
  SORTIE, /* Sortie du terrain */
} resultat_deplacement;

/* Avancer le robot sur le terrain : */
resultat_deplacement avancer_envt(Environnement *envt);

/* Tourner le robot àgauche */
void gauche_envt(Environnement *envt);

/* Tourner le robot àdroite */
void droite_envt(Environnement *envt);

/* Effectuer une mesure
   Paramè tre d : la direction de la mesure
     0 sur place
     1 devant
     2 devant droite
     3 droite
     4 derriè re droite
     5 derriè re
     6 derriè re gauche
     7 gauche
     8 devant gauche
   Renvoie le résultat de la mesure :
     0 rien (case libre ou en-dehors du terrain)
     1 eau
     2 rocher
 */
int mesure_envt(Environnement *envt, int d);

/* Afficher le terrain avec la position et l'orientation du robot */
void afficher_envt(Environnement *envt);

#endif
```