Premiers programmes en C

1 Du code source à l'exécution

On utilisera un éditeur de texte simple (de type *gedit*) pour écrire le code source dans un fichier avec l'extension `.c`. Le code sera compilé avec *gcc*, lancé en ligne de commande dans un terminal (ou console) qui interprète des commandes écrites en *bash*.

1.1 Utilisation de *bash*

On donne ci-dessous quelques commandes utiles dans un terminal :

`cd ..`	remonter au dossier précédent
`cd dossier`	accéder à un dossier contenu dans le dossier courant
`ls`	obtenir la liste des fichiers du dossier courant
`echo texte`	afficher du texte
`echo $VAR`	afficher la valeur d'une variable (exemples : `SHELL`, `PATH`, etc.)

1.2 Structure d'un programme

La structure de base d'un programme en C est la suivante :

```c
#include <librairie.h>
int main(){
    instructions
    //commentaire sur une ligne
    /*commentaire sur plusieurs lignes*/
    return 0;
}
```

La directive `#include <lib.h>` permet d'inclure les déclarations de fonctions contenues dans le fichier `lib.h`. Les définitions de telles fonctions sont alors contenues dans un fichier `lib.c`.

Ensuite, un programme en C contient toujours une fonction `main`, qui est le point d'entrée du programme. Pour nous, son type de retour (ou type de sortie) est `int`, et par convention elle retourne `0`, valeur que l'on ne cherchera jamais à récupérer. Dans un premier temps, cette fonction ne prend pas d'arguments (*cf.* chapitre 3 – "pointeurs, tableaux et structures").

> **Remarque :** La fonction `main` doit être placée après la déclaration de toutes les fonctions auxquelles elle fait appel.

1.3 Compilation

La compilation (simple) d'un fichier source se fait comme suit : `gcc code.c -o exe`.

L'opération de compilation :

- gère l'inclusion des déclarations des libraires (les `#include`)
- vérifie que tous les identificateurs utilisés (noms de fonctions, de variables ou de types) sont

bien déclarés préalablement
- détecte les erreurs de syntaxe
- vérifie la cohérence des types dans les appels de fonction, dans les affectations de variable, et autour des opérateurs
- vérifie l'existence de la définition d'une fonction `main` de type de retour `int`
- produit un fichier exécutable (ici, `exe`) qui correspond à l'exécution de cette fonction `main`

1.4 Exécution

L'exécution d'un fichier exécutable se fait via la commande : `./exe`.

2 Quelques types de base

2.1 Types

Le tableau ci-dessous donne un récapitulatif des types les plus fréquemment recontrés en C.

Type T	Valeurs possibles	Remarques	Opérateurs internes ($T\,\alpha\,T \to T$)
`int`	$[-2^{31}..2^{31}[$ (sur 32 bits)	Il existe d'autres types pour les entiers (*cf.* chapitre 2 – "codage des nombres")	`+`, `*`, `-` (binaire), `-` (unaire), `/` (division entière), `%` (modulo)
`float`	*cf.* chapitre 2 – "codage des nombres"	Si l'une des deux opérandes est de type `float` et l'autre de type `int`, c'est l'opération pour les `float` qui s'applique	`+`, `*`, `-` (binaire), `-` (unaire), `/` (division réelle)
`bool`	`true` et `false`	Il faut inclure la librairie correspondante en tapant `#include <stdbool.h>`	`!` (négation, unaire), `&&` (conjonction "et"), `||` (disjonction "ou")
`char`	minuscules, majuscules, chiffres, symboles et autres caractères, `\n` (nouvelle ligne), `\t` (tabulation), `\b` (retour en arrière), `\0` (caractère nul)	On utilise des guillemets simples pour les initialiser (par exemple `'a'`)	
`char*`	chaînes de caractères	On utilise des guillemets doubles (par exemple `"salut"`)	

2.2 Opérateurs de comparaison

Les opérateurs de comparaison ($T\,\alpha\,T \to$ `bool`) sont des opérateurs binaires, *i.e.* agissant sur deux valeurs du même type, et leur sortie est booléenne. En-voici les principaux :

`==`	teste l'égalité
`!=`	teste la différence
`>=`	teste la supériorité large
`<=`	teste l'infériorité large
`>`	teste la supériorité stricte
`<`	teste l'infériorité stricte

Remarque : On peut écrire de manière équivalente `a!=b` ou `!(a==b)`.

3 Affichage et saisie

Afin de réaliser l'affichage et la saisie, on utilise la librairie standard : `#include <stdio.lib>`. La partie `std` signifie "standard", et `io` signifie "in-out".

3.1 La fonction `printf`

La fonction `printf` affiche une chaîne de caractères dont des portions peuvent être variables au sens où elles sont fixées seulement à l'exécution. On indique ces portions par des spécificateurs de format qui dépendent du type de l'expression à afficher :

`%d`	pour les `int`
`%f`	pour les `float`
`%c`	pour les `char`
`%s`	pour les `char*`

La chaîne de caractères contenant les spécificateurs de format s'appelle une chaîne de contrôle.

Le nombre d'arguments de `printf` est $1+n$ où n est le nombre de spécificateurs de format contenus dans son premier argument (qui est une chaîne de contrôle). On l'appelle en tapant :

`printf("chaîne de contrôle",val`$_1$`,val`$_2$`,...,val`$_n$`)`

où la chaîne de contrôle contient n fois le symbole %.

3.2 La fonction `scanf`

La fonction `scanf` premet de récupérer une valeur tapée par l'utilisateur. Elle prend en argument une chaîne de caractères contenant le spécificateur de format correspondant au type de la valeur que l'on souhaite enregistrer, ainsi que l'adresse d'une variable existante (*i.e.* déjà déclarée).

Lorsqu'elle est appelée, elle stocke la valeur saisie dans cette variable, et ce en écrasant son éventuelle valeur précédente (elle réalise donc un passage par référence : *cf.* chapitre 3 – "pointeurs, tableaux et structures") :

```
type var;
scanf("%t",&var);
```

où `type` est un type et `%t` est le spécificateur de format associé.

4 Fonctions

4.1 Syntaxe

4.1.1 Définition

La définition d'une fonction se fait de la manière suivante :

```
typesortie nomfonction (t1 var1,t2 var2,...,tn varn){
    instructions
    return exp;
}
```

où $\texttt{var}_1$,...,$\texttt{var}_n$ sont les arguments, $\texttt{t}_1$,...,$\texttt{t}_n$ leurs types respectifs, et où `exp` est de type `typesortie`.

Dans la partie `instructions`, on peut :
- créer des variables locales, au début par convention
- modifier les variables localces, réaliser des opérations, des tests, des affichages, des saisies... dans ce qu'on appelle le corps de la fonction.

Remarque : Si le type de sortie est `void`, la fonction ne doit rien retourner. On n'écrit donc pas la dernière ligne du modèle précédent.

4.1.2 Appel

L'appel à une fonction ayant n arguments de types $\texttt{t}_1$,...,$\texttt{t}_n$ se fait comme suit :

$$\texttt{nomfonction (exp}_1\texttt{,exp}_2\texttt{,...,exp}_n\texttt{)}$$

À l'exécution de cet appel, chaque expression est évaluée et la valeur obtenue est stockée dans une variable locale à la fonction (qui est supprimée à la fin de l'appel) nommée $\texttt{var}_1$,$\texttt{var}_2$,...,$\texttt{var}_n$:

$$\begin{aligned}\texttt{t}_1\ \texttt{var}_1\ &= \texttt{exp}_1\\ \texttt{t}_2\ \texttt{var}_2\ &= \texttt{exp}_2\\ &\vdots\\ \texttt{t}_n\ \texttt{var}_n\ &= \texttt{exp}_n\end{aligned}$$

On parle de passage par valeurs. Ainsi, si $\texttt{exp}_i$ est une variable, toute instruction à l'intérieur de la fonction qui modifierait la valeur de $\texttt{var}_i$ laissera inchangée une telle variable.

Remarque : Si la fonction est de type `void`, alors cet appel correspond à une instruction.

Remarque : Dans un code source, on évitera toujours autant que possible d'avoir des variables globales, privilégiant plutôt les variables locales, c'est-à-dire déclarée es à l'intérieur des fonctions (y compris la fonction `main`).

4.2 Jeu de test

Afin de "vérifier" si les fonctions que l'on écrit renvoient bien les résultats voulus, il est possible de réaliser des jeux de tests grâce à la fonction `assert` de la librairie assert : `#include <assert.h>`.

Elle prend pour unique argument une expression booléenne. Par exemple pour une fonction :

$$\texttt{assert (nomfonction(exp}_1\texttt{,exp}_2\texttt{,...,exp}_n\texttt{) == exp\_sortie)}$$

Si la fonction est de type de sortie booléen, on peut (et on préférera) écrire :

- `assert (nomfonction(`e_1`,`e_2`,...,`e_n`))` pour tester si elle retourne bien `true`
- `assert (!nomfonction(`e_1`,`e_2`,...,`e_n`))` pour vérifier qu'elle retourne bien `false`.

Remarque : Ces jeux de tests nécessiteront parfois des descriptions sous formes de commentaires, par exemple s'il n'est pas évident de savoir ce que l'on teste précisément ou bien à quel résultat on s'attend.

Remarque : En général, on évite d'utiliser les opérateurs `&&` et `||` dans les `assert`.

Remarque : Notons par ailleurs que l'on ne peut pas faire de jeu de test pour une fonction de type de sortie `void`. Il faudra dans ces cas se contenter de vérifier les éventuels affichages ou modifications de la mémoire qu'elle produit, y compris en utilisant des `assert` sur d'autres objets que la fonction elle-même, lorsque cela est possible.

5 Branchement conditionnel

Les branches conditionnelles permet de réaliser des instructions selon que des conditions choisies au préalable sont vérifiées ou non. Elle prennent la forme suivante :

```
if(cond_a){
    instructions_si_a
}
else if(cond_b){
    instructions_si_b
}
else{
    intructions_sinon
}
```

où $\texttt{cond}_a$ et $\texttt{cond}_b$ sont des expressions booléennes.

Remarque : Bien évidemment, on peut écrire des branches conditionnelles avec autant de conditions "alternatives" que l'on souhaite.

Remarque : Au contraire, il est possible de n'avoir aucune condition alternative, *i.e.* de ne pas mettre de branche `else`. Si de plus la réalisation de la condition ne conduit qu'à une seule instruction, on peut omettre les accolades et taper tout simplement : `if(cond) instruction`.

6 Boucles `while`

6.1 Syntaxe

Les boucles `while` ou "tant que" en C utilisent la syntaxe suivante :

```
while (cond){
    instructions
}
```

où `cond` est une expression de type `bool`.

6.2 Simulation de boucle

Définition (*simulation de boucle*) :

Une simulation de boucle est un tableau qui représente l'exécution d'une boucle avec des valeurs fixées pour les variables qui ne sont pas modifiées dans la boucle. Plus précisément :

- la première colonne indique le tour de boucle, et par convention 0 représente la situation initiale
- à cette colonne, on ajoute ensuite une colonne par variable modifiée dans la boucle, dans l'ordre de leur première modification dans la boucle
- chaque ligne donne les valeurs de ces variables à la fin du tour de boucle

6.3 Preuve de terminaison et de correction d'algorithmes

6.3.1 Définitions

Définition (*terminaison*) :

On dit qu'un algorithme ou une fonction termine si pour toute famille d'arguments valide, l'appel de la fonction consiste en un nombre fini d'instructions.

Définition (*correction*) :

On dit qu'un algorithme ou une fonction est correct(e) si elle termine et que pour toute famille d'arguments valide, la sortie est celle attendue.

Définition (*variant de boucle*) :

Un variant de boucle est une expression écrite avec les variables de boucle (et éventuellement d'autres variables localces de la fonction) et qui est à valeurs dans un ensemble ordonné bien fondé, *i.e.* dans lequel il n'existe pas de suite infinie et strictement décroissante, (*cf.* chapitre 8 – "ordre et induction").

Remarque : On utilise typiquement l'ensemble ordonné bien fondé $(\mathbb{N}, \leq)$ ou bien $(\mathbb{Z}^-, \geq)$ qui lui est équivalent, mais il existe évidemment existe d'autres ensembles bien fondés (qui par ailleurs ne contiennent pas nécessairement des nombres : *cf.* chapitre 8 – "ordre et induction").

Définition (*invariant de boucle*) :

Un invariant de boucle est une expression booléenne écrite avec les variables de la boucle (et éventuellement des variables locales à la fonction) et qui est évaluée à vrai à chaque tour de boucle pour n'importe quel appel à la fonction avec des arguments valides.

6.3.2 Schéma de preuve

En pratique, lorsque l'on veut montrer la terminaison d'un algorithme ou d'une fonction comportant une boucle, on procède comme suit :

- on fixe les valeurs des arguments
- on suppose par l'absurde que l'appel de la fonction sur ces arguments ne termine pas
- on introduit les suites des valeurs prises par les variables à chaque tour de boucle
- on décrit la variation (strictement croissante ou décroissante) de la suite à l'aide des instructions du corps de la boucle et de la condition initiale, et on justifie qu'elle est à valeurs entières
- grâce à la condition de boucle, on montre que cette suite est majorée ou minorée
- on conclut alors en disant que c'est absurde.

Ensuite, la terminaison ayant été démontrée (ou admise, auquel cas il faut fixer les arguments puisque ce n'est pas déjà fait), on prouve que la fonction est correcte grâce à un invariant, qui se montre par récurrence, et selon le schéma suivant :

- on considère le nombre de tours de boucles effectués par l'appel
- on introduit les suites finies des valeurs prises par les variables intervenant dans l'invariant
- on introduit la propriété à démontrer par récurrence ($\mathcal{P}_k$, $\mathcal{H}_k$, etc.)
- on effectue l'initialisation avec les instructions écrites avant la boucle
- on effectue l'hérédité grâce au corps de la boucle
- on conclut par récurrence
- on utilise la propriété ainsi démontrée au dernier rang et la condition de boucle pour expliciter la valeur de la variable qui nous intéresse au dernier rang
- on conclut sur la correction de l'algorithme.

Exemples : *cf.* annexe "exemples de preuves de terminaison et de correction".