Variables

En C, une donnée est stockée dans une variable caractérisée par :

- son **type** et son **identificateur** (définis lors de la **déclaration**) ;
- sa **valeur**, définie la première fois lors de l'**initialisation** puis éventuellement modifiée par la suite.

Rappels de syntaxe :

```
type id ;
type id = valeur;

id = expression ;
```

Types élémentaires :

```
int
double
char
```

Exemples :

```
int val = 2 ;
double const pi = 3.141592653;
i=j+3;
```

Les variables non modifiables se déclarent avec le mot réservé `const` :

```
double const g = 9.81;
```

Opérateurs

Operateurs arithmétiques

`*`	multiplication	
`/`	division	
`%`	modulo	
`+`	addition	
`-`	soustraction	
`-`	opposé	*(1 opérande)*
`++`	incrément	*(1 opérande)*
`--`	décrément	*(1 opérande)*

Operateurs de comparaison

`==`	teste l'égalité logique
`!=`	non égalité
`<`	inférieur
`>`	supérieur
`<=`	inférieur ou égal
`>=`	supérieur ou égal

Operateurs logiques

`&&`	"et" logique	
`||`	ou	
`!`	négation	*(1 opérande)*

Priorités (par ordre décroissant, tous les opérateurs d'un même groupe sont de priorité égale) :

`()` `[]` `->` `.`, `!` `++` `--`, `*` `/` `%`, `+` `-`, `<` `<=` `>` `>=`, `==` `!=`, `&&`, `||`, `=` `+=` `-=` etc., `,`

Les structures de contrôle

les branchements conditionnels : *si ... alors ...*

```
if (condition)
    instructions
.............................
if (condition 1)
    instructions 1
...
else if (condition N)
    instructions N
else
    instructions N+1
```

```
switch (expression) {
    case valeur:
        instructions;
        break;
    ...
    default:
        instructions;
}
```

les boucles conditionnelles : *tant que ...*

```
while (condition)
Instructions
```

```
do Instructions while
(condition);
```

les itérations : *pour ... allant de ... à ...*

```
for (initialisation ; condition ; increment)
    instructions
```

les sauts : `break;` et `continue;`

Note : `instructions` représente 1 instruction élémentaire ou un bloc.
`instructions;` représente une suite d'instructions élémentaires.

Les fonctions

Prototype (à mettre **avant** toute utilisation de la fonction) :

```
type nom ( type1 arg1, ..., typeN argN );
```

`type` est `void` si la fonction ne retourne aucune valeur.

Définition :

```
type nom ( type1 arg1, ..., typeN argN )
{
   corps
   return value;
}
```

Passage par **valeur** :

```
type f(type2 arg);
f(x)
```

☞ `x` **ne** peut **pas** être modifié par `f`

Passage par **référence** (valeur de pointeur en fait) :

```
type f(type2* arg);
f(&x)
```

☞ `x` peut **être modifié** par `f`

Les tableaux

déclaration : `type identificateur[taille];`
déclaration/initialisation :

$$type\ identificateur[taille] = \{val_1, \ldots, val_{taille}\};$$

Accès aux éléments : `tab[i]` `i` entre **0** et **taille-1**

Le passage `type1 f(type2 tab[]);` d'un tableau `tab` à une fonction `f` se fait automatiquement **par référence**
pour éviter les effet de bords : `type1 f(type2 const tab[]);`

tableau multidimentionnel :
$type\ identificateur[taille1][taille2];$
`tab[i][j];`

Les tableaux ne peuvent pas être des types de retour pour les fonctions. `:-(`

Les structures

Déclaration du type correspondant :

```
struct Nom_du_type {
        type1 champ1 ;
        type2 champ2 ;
        ...
};
```

Déclaration d'une variable :

```
struct Nom_du_type identificateur;
```

Déclaration/Initialisation d'une variable :

```
struct Nom_du_type identificateur = { val1, val2, ... };
```

Accès à un champs donné de la structure :

```
identificateur.champ
```

Affectation globale de structures :

```
identificateur1 = identificateur2
```

Les pointeurs

Déclaration : `type* identificateur;`

Adresse d'une variable : `&variable`
Accès au contenu pointé par un pointeur : `*pointeur`

Pointeur sur une constante : `type const* ptr;`
Pointeur constant : `type* const ptr = adresse;`

Allocation mémoire :

```
#include <stdlib.h>

pointeur = malloc(sizeof(type));
pointeur = calloc(nombre, sizeof(type));
pointeur = realloc(pointeur, sizeof(type));
```

Libération de la zone mémoire allouée : `free(pointeur);`

Pointeur sur une fonction : `type_retour (* ptrfct)(arguments...)`

Les chaînes de caractères

Valeur littérale : `"valeur"`

Déclarations :

```
char* nom;
char nom[taille];
char nom[] = "valeur";
```

Écriture : `printf("...%s...", chaine);` ou `puts(chaine);`

Lecture : `scanf("%s", chaine);` ou `gets(chaine);`

Quelques fonctions de `<string.h>` :

`strlen`	`strcat`
`strcpy`	`strncat`
`strncpy`	`strchr`
`strcmp`	`strrchr`
`strncmp`	`strstr`

Jean-Cédric Chappelier
EPFL

Les entrées/sorties

Clavier / Terminal : `stdin` / `stdout` et `stderr`

Fichier de définitions : `#include <stdio.h>`

Utilisation :

écriture : `int printf("FORMAT", expr1, expr2, ...);`
lecture : `int scanf("FORMAT", ptr1, ptr2, ...);`

Saut à la ligne : `'\n'`

Lecture d'une ligne entière :

```
char* fgets(char* s, int size, FILE* stream);
```

Les entrées/sorties (fichiers)

Type : `FILE*`

ouverture : `FILE* fopen(const char* nom, const char* mode)`

Mode :
`"r"` en lecture, `"w"` en écriture (écrasement), `"a"` en écriture (à la fin), suivit de '`+`' pour ouverture en lecture et écriture, et/ou de '`b`' pour fichiers en binaires

Écriture :
`fprintf(FILE*, ...)` pour fichiers textes
`size_t fwrite(const void* adr_debut, size_t taille_el, size_t nb_el, FILE*);` pour les fichiers binaires

Lecture :
`fscanf(FILE*, ...)` pour fichiers textes
`size_t fread(void* adr_debut, size_t taille_el, size_t nb_el, FILE*);` pour les fichiers binaires

Test de fin de fichier : `feof(FILE*)`

Fermeture du fichier : `fclose(FILE*)`

Jean-Cédric Chappelier
EPFL

Compilation séparée

Compilation modulaire
$\Rightarrow$ séparation des prototypes (dans les fichier `.h`) des définitions (dans les fichiers `.c`)

$\Rightarrow$ compilation séparée

1. Inclusion des prototypes nécessaires dans le code :
   `#include "header.h"`
2. Compilation vers un fichier "objet" (`.o`) : `gcc -c prog.c`
3. Lien entre plusieurs objets :
   `gcc prog1.o prog2.o prog3.o -o monprog`

`Makefile` :
moyen utile pour décrire les dépendances entre modules d'un projet (et compiler automatiquement le projet)

Syntaxe :

```
cible: dependance <TAB>commande
```


Jean-Cédric Chappelier
EPFL