# CCI - Principes des langages de programmation
# TD 2 : Types de données

François Yvon[*], Thomas Tang[†]

25 septembre 2007

## Première partie
# Pour bien assimiler ...

## 1 Les variables et leurs types

Nous allons voir maintenant le concept de variable qui permet d'accéder à un espace de la mémoire où on stocke des informations à l'aide d'un nom ou identificateur.

### 1.1 Déclaration des variables

Du point de vue du programme, une variable a trois attributs :
- un **identificateur**
- un **type**
- une **valeur**

Du point de vue du compilateur, la variable a en plus une **adresse**. Pour pouvoir utiliser une variable, il faut au préalable la déclarer, par exemple :

```
char c='s';
```

Dans le programme suivant on déclare une variable x de type int (le type des entiers).

```
/* td2.c */
#include <stdio.h>
int main() {
    int x=3;
    printf("Hello World !\n") ;
    printf("Un entier : %i\n",x);
}
```

*Compilez et exécutez le programme.*

Chaque affectation change le contenu de la variable également dans la suite du programme :

```
/* td2.c */
#include <stdio.h>
int main() {
    int x=3;
    printf("Hello World !\n") ;
    printf("Un entier : %i\n",x) ;
```

[*] yvon@limsi.fr
[†] tang@cgm.cnrs-gif.fr

```
    x = 4 ;
    printf("Un entier : %i\n",x) ;
}
```

*Compilez et exécutez le programme.*

## 1.2 Portée des variables

Un programme C classique est composé de différents blocs et de différentes fonctions. On dit qu'une variable est dans la portée d'un bloc si elle est définie pour ce bloc. On distingue pour les variables 2 sortes de portées (sorte de domaine de définition) :

- **globale** : la variable est déclarée à l'extérieur de tous blocs et est accessible par l'ensemble du programme.
- **locale** : la variable est définie à l'intérieur d'un bloc et n'est accessible qu'à l'intérieur de ce bloc.

```
/* td2portee.c */
#include <stdio.h>
/* une variable de portée globale */
int var_global = 1 ;

int main() {
    /* une variable de portée locale à la fonction main */
    int x ;
    x = var_global + 1 ;
    printf("Un entier : %i\n",x) ;
    if (x == 3) {
        /* une variable de portée locale au bloc */
        int y ;
        y=x - var_global ;
    }
    printf("Un entier : %i\n",x) ;
}
```

*Compilez et exécutez le programme.*
*Que se passe-t-il si nous ajoutons à la fin de la fonction main la ligne suivante :*

```
printf("Un autre entier : %i\n",y) ;
```

*Pourquoi ?*

## 1.3 Entrées/Sorties

Nous avons vu jusqu'à présent comment afficher des entiers et des flottants (cf. TD1), mais nous n'avons pas encore vu comment un programme peut recevoir une entrée. La fonction `getchar()` permet de récupérer un caractère sur l'entrée standard du programme. Elle renvoie un entier représentant un caractère (son code ASCII).

```
/* td2_entree_sortie.c */
#include <stdio.h>
int main() {
    if (getchar() == 'Y') {
        printf("Yes\n") ;
    }
    else {
        printf("No\n") ;
    }
}
```

*Compilez et exécutez le programme.*

`scanf()` est une fonction un peu plus évoluée qui permet de récupérer plusieurs valeurs et de les convertir dans le bon type. Nous verrons ultérieurement le fonctionnement complet de `scanf()`, pour l'instant on en restera à une utilisation simple pour récupérer un entier ou un flottant sur l'entrée standard :

```
/* td2_entree_sortie.c */
#include <stdio.h>
int main() {
    int x ;
    float f ;
    scanf("%i",&x) ;
    scanf("%f",&f) ;
}
```

*Compilez et exécutez le programme.*

`scanf()` renvoie le nombre d'affectation effectuée (ici toujours 1 ou 0 s'il y a eu une erreur.) Ce comportement permet de vérifier que la valeur fournie est bien du type attendu.

```
/* td2_entree_sortie.c */
#include <stdio.h>
int main() {
    int x ;
    printf("Rentrez un nombre entier :\n") ;
    if ( scanf("%i",&x) ) {
        printf("Vous avez rentre : %i\n",x) ;
    }
    else {
        printf("Ce n'est pas un nombre entier !\n") ;
    }
}
```

*Compilez et exécutez le programme.*

# 2 Nombres entiers et réels

## 2.1 Le type `int`

Comme on l'a vu précédemment, le type pour représenter les entiers est `int`, les entiers sont codés sur 32 bits (sur une machine 32 bits) et sont signés (les valeurs vont donc de -2147483648 à 2147483647). On peut spécifier si l'on veut un entier non signé lors de la déclaration :

```
unsigned int x ;
```

Il existe d'autres tailles d'entier :
- `char` (8 bits, repésente également les caractères.
- `short int` (16 bits)
- `long int` (32 bits).

*Pour chaque type d'entier donnez sa valeur maximale et minimale s'il est signé.*
*Pour chaque type d'entier donnez sa valeur maximale et minimale s'il est non signé.*
*Soit la fonction sizeof(type) qui renvoie dans un int la taille d'un type en octets, écrire un programme affichant la taille en octet de chaque type d'entier.*

### 2.2 Le type `float`

Les réels sont représentés par le type `float` sur 32 bits (le calcul des valeurs maximales et minimal n'est pas aussi simple que pour les entiers.) Comme pour le type entier il existe d'autres tailles :

- `double` (64 bits)
- `long double` (80 bits).

*Ecrivez un programme qui stocke dans une variable f de type float la valeur de 2/3 et qui affiche cette valeur avec une précision de 3, puis 6 et enfin 12 chiffres (cf. le précédant TD).*
*Modifier le programme pour que f soit de type double. Que pensez-vous des résultats ? Conclure.*
*Reprendre le programme qui affiche la taille des entiers et ajouter la taille des différents types flottants.*

Les flottants sont des nombres réels dit à virgule flottante c'est à dire que la représentation du nombre n'accorde pas toujours la même taille pour la partie entière et la partie réelle du nombre, ce qui permet de représenter des nombres réels très petits ou très grands. Cette flexibilité a un coût, les calculs sur les flottants sont plus complexes que ceux sur les entiers, mais surtout il existe des approximations et le comportement n'est pas toujours celui attendu :

```
#include <stdio.h>
int main() {
    printf("un réel surprenant : %.17f\n",1.1+2.7) ;
}
```

*Qu'affiche ce programme d'après vous ? Compilez le et exécutez le, êtes-vous surpris du résultat ?*

## 3 Conversion entre types : Transtypage

Habituellement on ne différencie pas toujours les entiers des réels et faire des opérations mélangeant les deux n'est pas toujours ne nous pose pas de problème. Les choses sont différentes avec un langage de programmation, les entiers et les flottants ne sont pas représentés de la même façon en mémoire et les conversions peuvent provoquer des surprises.

### 3.1 Transtypage implicite

```
#include <stdio.h>
int main() {
    int i1 = 3, i2 = 2, i3;
    float f1 = 3, f2 = 2, f3 ; i3 = i1/i2 ; f3 = f1/f2 ;
    printf("i3 = %i\nf3 = %f\n",i3,f3) ;
    i3 = f1/f2 ; f3 = i1/i2 ;
    printf("i3 = %i\nf3 = %f\n",i3,f3) ;
}
```

*Quelles valeurs seront affichées d'après vous ?*
*Compilez et exécuter ce programme, les valeurs affichées corresponde-t-elle à votre attente ?*

### 3.2 Transtypage explicite

On peut forcer la conversion explicite d'un type vers un autre avec la notation `(type)expression`. Par exemple, vous pouvez explicitement transformer un flottant en entier par `(int)1.5`. *Utilisez le transtypage explicite pour afficher comme entier la valeur flottante 1.5*
*Modifier votre programme précédant avec le transtypage explicite de manière à ce que le deuxième affichage de f3 corresponde au premier.*

# 4 Les structures

Les entiers et les flottants ne sont pas les seuls types de données disponibles, dans la suite nous verrons d'ailleurs d'autres types de base. Mais ces types de base ne sont pas toujours suffisants, on peut avoir besoin de types plus évolués que l'on nomme en général types structurés, ces types peuvent être définis par l'utilisateur en fonction de ses besoins.

## 4.1 Définitions

Nous allons maintenant voir les structures (il existe d'autre types structurés comme les tableaux et les unions.) Une structure est le regroupement au sein d'une même valeur de plusieurs autres valeurs auxquelles on affecte un nom. Voici un exemple :

```
{
    champ1 = 1 ;
    champ2 = "une chaine" ;
    champ3 = 1.5 ;
}
```

Pour pouvoir déclarer une structure il faut tout d'abord définir la forme structure à l'aide du mot clef `struct`, par exemple :

```
struct exemple {
    int champ1;
    char * champ2;
    float champ3;
};
```

ATTENTION ! ne pas oublier le **;** après l'accolade fermant la structure.

La définition des structures se fait en général au même niveau que les variables globales pour que celle-ci soit accessible dans tout le programme. On pourra après définir des variables pouvant contenir une structure de cette forme toujours grâce au mot clef structure :

```
/* la variable s peut contenir une structure de forme exemple */
struct exemple s ;
```

*Ecrire la définition de la forme d'une structure permettant de représenter un point sur le plan avec les champs x et y de type float. Vous donnerez comme nom à cette forme de structure point.*

## 4.2 Utilisation

Pour accéder aux champs d'une structure on utilise le point **.** :

```
int x ;
struct exemple s ;
x = s.champ1 ;
```

On peut utiliser le champ d'une structure comme une autre variable :

```
struct exemple s ;
s.champ1 = 1 ;
```

*Ecrire un programme qui utilise la structure de point définie précédemment, déclarer une variable origine correspondant au point (0, 0). Votre programme devra afficher la valeur des champs de la variable origine.*

## Deuxième partie

# Pour approfondir ...

## Exercice 1 : Portée des variables

```
/* td2exo1.c */
#include <stdio.h>

int x1 ;

int main() {
    int x2 ;
    x1 = 5 + 3 ;
    x2 = x1 * x1 ;
    if (x1 == x2 - x1){
        int x3 = 5 ;
        printf("x1 = %i; x2 = %i; x3 = %i\n",x1,x2,x3) ;
    }
    else {
        int x1 = 1 ;
        printf("x1 = %i\n",x1) ;
    }
    printf("x1 = %i; x2 = %i; x3 = %i\n",x1,x2,x3) ;
}
```

*Pour chaque variable donner sa portée.*
*Sans compiler ni exécuter le programme, indiquez ce qu'affiche la ligne 14.*
*Est ce que chaque variable utilisée à la ligne 11 est dans la portée du bloc ?*
*Idem pour la ligne 16 ?*

## Exercice 2 : Opérations

*Ecrire un programme qui demande à l'utilisateur quelle opération élémentaire il souhaite effectuer (désignée par + pour l'addition, - pour la soustraction, * pour la multiplication et / pour la division) sur deux entiers saisis par l'utilisateur et qui renvoie le résultat de l'opération.*

## Exercice 3 : Conversions Degrés / Grades / Radians

*Vous devrez écrire un programme qui convertit un angle en degrés/grades/radians vers le même angle dans une des deux autres unités. Votre programme devra demander l'unité d'origine (désigné par la lettre D pour les degrés, la lettre R pour les radians et G pour les grades) puis la valeur à convertir et enfin afficher la conversion.*

## Exercice 4 : Structure et vecteurs

*Ecrire un programme en utilisant la notion de structure permettant à partir des coordonnées de l'espace (x,y,z) de 2 points rentrées par l'utilisateur d'afficher en sortie :*

- *les coordonnées des deux points rentrées*
- *la distance séparant les deux points*
- *les coordonnées du vecteur*

*Modifier le programme de façon à pouvoir rentrer quatre points A, B, C, D (toujours de l'espace). Calculer les produits scalaire et vectoriel des deux vecteurs ainsi formés : u=vecteur(AB), v=vecteur (CD)*

## Troisième partie

# Annexe : Opérations sur les entiers

| Opérations arithmétiques | | |
|---|---|---|
| Addition | `x + y` | |
| Soustraction | `x - y` | |
| Multiplication | `x * y` | |
| Division | `x / y` | |
| Modulo | `x % y` | Le reste de la division entière |
| **Comparaison** | | |
| Égal | `x == y` | |
| Différent | `x != Y` | |
| Supérieur | `x > y` | |
| Inférieur | `x < y` | |
| Supérieur ou égal | `x >= y` | |
| Inférieur ou égal | `x <= y` | |
| **Opérateur logique** | | |
| Et | `x && y` | |
| Ou | `x \|\| y` | |
| Non | `!x` | |
| **Opérateur bit à bit** | | |
| Et | `x & y` | `0011 & 0101 = 0001` |
| Ou (inclusif) | `x \| y` | `0011 \| 0101 = 0111` |
| Ou (exclusif) | `x^y` | `0011^0101 = 0110` |
| Complément | `x` | Négation bit à bit |
| Décalage à droite | `x` $\gg$ `y` | équivalent à $\frac{x}{2^y}$ |
| Décalage à gauche | `x` $\ll$ `y` | équivalent à $x \times 2^y$ |
| **Affectations particulières** | | |
| Addition | `x += y` | `x = x + y` |
| Incrément | `++x` | `x = x + 1` et renvoie la nouvelle valeur de x |
| Postincrément | `x++` | `x = x + 1` et renvoie l'ancienne valeur de x |
| Décrément | `--x` | `x = x - 1` et renvoie la nouvelle valeur de x |
| Postdécrément | `x--` | `x = x - 1` et renvoie l'ancienne valeur de x |

FIG. 1 – Opération sur les entiers