[Math-L312] TP 2 : Structures de contrôle en C

Adrien Semin
adrien.semin@math.u-psud.fr

1 Rappels sur les types du langage C

1.1 Les types de base

Les types de base correspondent au stockage de valeurs numériques. Dans le cas de valeurs non entières on parlera de nombres *flottants* car ceux-si sont représentés de façon interne en utilisant une écriture scientifique (mantisse et exposant) autorisant une position variable de la virgule : on l'appelle représentation à *virgule flottante*.

Types entiers		
Signés	Capacité ou taille	
Oui	Petite	`char`
	Moyenne	`short`
	Grande	`long`
Non	Petite	`unsigned char`
	Moyenne	`unsigned short`
	Grande	`unsigned long`

Types flottants	
Précision	
Simple	`float`
Grande	`double`
Encore plus grande	`long double`

Types nommés	
Types énumérés	`enum`

Le type `int` usuel correspond au type entier signé natif le plus efficace pour les processeurs d'une machine donnée, qu'on désigne en général par le terme *mot*. Il correspond habituellement à `short`.

Les seules spécifications par rapport aux taille relatives des types d'une même famille entre eux, c'est que tous les nombres représentables par un type (entier ou flottant) doivent être représentables par le type de taille suivante (entier ou flottant). Par exemple, tout `short` peut être représenté par un `long`, mais en fonction de la machine où l'on travaille, ces 2 types peuvent être les mêmes.

Le type `enum` s'utilise de façon assez particulière. Par exemple :

```
typedef enum
  {lundi, mardi, mercredi,
   jeudi, vendredi, samedi,
   dimanche} jour_semaine;
```

introduit un nouveau type, qui en fait correspond à un type entier. Les constantes `lundi`,...,`dimanche` sont alors définies automatiquement (comme si on l'avait fait manuellement à l'aide de la directive `#define` [1]). Ici on aura les équivalences suivantes :

`lundi` $\Longleftrightarrow 0$ `mardi` $\Longleftrightarrow 1$ `dimanche+1` $\Longleftrightarrow 7$

1.2 Constantes littérales et tailles des types

Les constantes littérales numériques entières ou réelles suivent les conventions habituelles avec quelques particularités propres au langage C.

Les constantes entières

Elles sont sans signe : l'expression `-123` est comprise comme l'application de l'opérateur unaire `-` à la constante `123` ; mais puisque le calcul est fait pendant la compilation, cette subtilité n'a aucune conséquence pour le programmeur. Notez aussi qu'en C original, comme il n'existe pas d'opérateur `+` unaire, la notation `+123` est interdite.

Les constantes littérales entières peuvent aussi s'écrire en octal et en hexadécimal :

- une constante écrite en *octal* (base 8) commence par 0 (zéro) ;
- une constante écrite en *hexadécimal* (base 16) commence par `0x` ou `0X` (zéro, x).

Voici par exemple trois manières d'écrire le même nombre :

```
27      033      0x1B
```

Détail à retenir : on ne doit pas écrire de zéro non significatif à gauche d'un nombre : 0123 ne représente pas la même valeur que 123. Le type d'une constante entière est le plus

1. Cette directive s'utilise de la façon suivante : `#define x 2` fera que le compilateur remplacera le mot `"x"` par `"2"` partout où il le rencontrera. Il faut bien noter que ce remplacement a lieu avant la compilation.

petit type dans lequel sa valeur peut être représentée. Ou, plus exactement :

- si elle est décimale : si possible `int`, sinon `long`, sinon `unsigned long` ;
- si elle est octale ou hexadécimale : si possible `int`, sinon `unsigned int`, sinon `unsigned long`.

Certains suffixes permettent de changer cette classification :

- `U`, `u` : indique que la constante est d'un type `unsigned` ;
- `L`, `l` : indique que la constante est d'un type `long`.

Exemples : `1L` `0x7FFFU`
On peut combiner ces deux suffixes : `16UL`

Les constantes flottantes

Une constante littérale est l'expression d'un nombre flottant si elle présente, dans l'ordre :

- une suite de chiffres décimaux (la partie entière) ;
- un point, qui joue le rôle de virgule décimale ;
- une suite de chiffres décimaux (la partie fractionnaire) ;
- une des deux lettres `E` ou `e` ;
- éventuellement un signe `+` ou `-` ;
- une suite de chiffres décimaux.

Les trois derniers éléments forment l'exposant. Exemple : `123.456E-78`. On peut omettre :

- la partie entière ou la partie fractionnaire, mais pas les deux ;
- le point ou l'exposant, mais pas les deux.

Exemples : `.5e7` `5.e6` `5000000.` `5e6`

Une constante flottante est supposée de type `double`, à moins de comporter un suffixe explicite :

- les suffixes `F` ou `f` indiquent qu'elle est du type `float` ;
- les suffixes `L` ou `l` indiquent qu'elle est du type `long double`.

Exemples : `1.0L` `5.0e4f`

Caractères et chaînes de caractères

Une constante de type caractère se note en écrivant le caractère entre apostrophes. Une constante de type chaîne de caractères se note en écrivant ses caractères entre guillemets.

Exemples, trois caractères : `'A' '2' '"'`

Quatre chaînes de caractères : `"A" "Bonjour à tous !"  "" "'"`

On peut faire figurer n'importe quel caractère, même non imprimable, dans une constante caractère ou chaîne de caractères en utilisant les combinaisons suivantes, appelées séquences d'échappement :

`\n`	Nouvelle ligne (LF)
`\t`	Tabulation (HT)
`\b`	Espace-arrière (BS)
`\r`	Retour-chariot (CR)
`\f`	Saut de page (FF)
`\a`	Signal sonore (BELL)
`\\`	\
`\'`	'
`\"`	"
$\backslash d_3d_2d_1$	Le caractère qui a pour code le nombre *octal* $d_3d_2d_1$; s'il commence par un ou deux zéros et si cela ne crée pas une ambiguïté, on peut aussi le noter $\backslash d_2d_1$ ou $\backslash d_1$.

Par exemple, la chaîne suivante définit la suite des 9 caractères[2] : A, *escape* (de code ASCII 27), B, ", C, saut de page, D, \ et E :

```
"A\033B\"C\fD\\E"
```

Une constante de type caractère appartient au type char, c'est-à-dire entier représenté sur un octet. La valeur d'une constante caractère est le nombre qui représente le caractère de manière interne ; il s'agit presque toujours du code ASCII.

Une constante de type chaîne de caractères représente une suite finie de caractères, de longueur quelconque. Le codage interne d'une chaîne de caractères est le suivant :

- les caractères constituant la chaîne sont rangés en mémoire de manière contigüe, dans l'ordre où ils figurent dans la chaîne ;
- un caractère nul (`\0`) est ajouté après le dernier caractère pour indiquer la fin de la chaîne ;
- la constante chaîne représente alors, à l'endroit où elle est écrite, l'adresse de la cellule où est rangé le premier caractère.

Voici par exemple la représentation en mémoire (octet par octet) de la chaîne `"bonjour"` :

		b	o	n	j	o	u	r	\0		

Par conséquent, une constante chaîne de caractères a pour type celui d'un tableau de caractères (c'est à dire `char []`), et pour valeur l'adresse d'une cellule de la mémoire. Le caractère nul `\0` est celui dont le code ASCII est zéro[3].

2. En fait, le compilateur rajoute à la fin un caractère supplémentaire qui en marque la fin : le caractère \0.
3. Attention, il s'agit de `'\0'`, et surtout pas de `'0'` !

On peut aussi le noter `0` en notation décimale.

La taille des variables typées

Il ne faut pas oublier que d'une architecture machine à l'autre, la taille occupée par les différents types de base peut varier. La macro `sizeof` permet de connaître la taille mémoire occupée par un type ou une variable typée. On donne ici un programme en C qui permet d'afficher la taille occupée en octets par les types courants. Ce programme est téléchargeable à l'adresse `/home/doc/semin/TP/c/sizes.c`.

```
#include <stdio.h>

int main(){
  printf("sizeof(char) : %i\n",sizeof(char));
  printf("sizeof(short) : %i\n",sizeof(short));
  printf("sizeof(long) : %i\n",sizeof(long));
  printf("sizeof(unsigned char) : %i\n",sizeof(unsigned char));
  printf("sizeof(unsigned short) : %i\n",sizeof(unsigned short));
  printf("sizeof(unsigned long) : %i\n",sizeof(unsigned long));
  printf("sizeof(float) : %i\n",sizeof(float));
  printf("sizeof(double) : %i\n",sizeof(double));
  printf("sizeof(long double) : %i\n",sizeof(long double));
  return 0;
}
```

1.3 Les types dérivés

À partir des types de base précédents, on peut définir cinq nouveaux types qui en dérivent :

Les tableaux `[ ]`

Par exemple, on peut déclarer :

```
int T[255];
```

La variable `T` sera alors un tableau de `int` à 255 éléments. L'accès à ses éléments se fait avec des crochets (`[ ]`) : `T[0]` désigne le premier élément, `T[254]` le dernier. On peut aussi définir des types tableaux :

```
typedef float float_array
    [10000];
```

Cette ligne définit le type `float_array` : des tableaux de `float` à 10000 éléments. Il n'existe pas de tableaux multi-indicés en C, mais on peut toutefois faire des tableaux de tableaux : `int T[5][5]`.

Les déclarations de tableaux peuvent aussi être initialisées :

```
int T[5]={1,2,3,4,5};
```

Dans le cas de tableaux de caractères, on peut les initialiser avec une chaîne de caractères :

```
char S[10]="abcdefghi";
```

Le caractère `'\0'` sera ici inclus en bout de chaîne.

Les fonctions `( )`

On peut définir des variables et des types de fonctions. Par exemple, dans ce programme téléchargeable à l'adresse `/home/doc/semin/TP/c/fun.c` :

```
#include <stdio.h>

typedef int func(int x,int y);        /* type de fonction appelé func */

int f1(int x,int y){return x+y;} /* 2 "vraies" fonctions compatibles */
int f2(int x,int y){return x*y;}

int main(){
  printf("Valeur: %i\n",f1(1,2));             /* Le résultat est 1+2=3 */
  int a = f2(3,6); /* a contient 18 */
  printf("Valeur: %i\n",a);
  return 0;
}
```

Le type `void` Certaines fonctions peuvent très bien ne rien retourner. Dans ce cas-là, on les déclarera en `void`, comme dans le programme ci-dessous, téléchargeable à l'adresse `/home/doc/semin/TP/c/fun2.c`

```
#include<stdio.h>

void affichage(char[]);

int main(void)
{
  char chaine[200] = "Hello u-psud!\n";
  affichage(chaine);
  return 0;
}

void affichage(char t[])
{
  printf(t);
}
```

Les pointeurs *

Les variables qu'on définit en C sont stockées dans la mémoire physique de la machine qui les exécute. Chaque case de cette mémoire (l'unité de base étant l'octet) est numérotée. Les pointeurs permettent de référencer l'endroit physique de la mémoire où est stocké une variable. Par exemple, si on déclare :

```
int n;
```

alors `&n` est une valeur qui représente l'adresse de `n`, son type est `int*`, on dira que c'est un pointeur vers `n`. L'opérateur inverse de `&` est `*`, par exemple `*(&n)` est strictement équivalent à `n`. Les pointeurs sont utiles pour passer des paramètres par référence à des fonctions plutôt que par valeur :

```
void inc1(int x){x=x+1;}
void inc2(int* x){*x=(*x)+1;}
```

L'appel de fonction `inc1(y)` ne modifie pas la variable `y`, mais `inc2(&y)` si. On notera au passage que le type pointeur le moins fort est `void*`, tous les types pointeur en sont des «descendants». `sizeof(void*)` permet de connaître, pour une machine donnée, la taille des adresses pour accéder à la mémoire. Elle est en général de 4 octets (32 bits).

Les structures `struct`

Donnons un exemple pour commencer :

```
struct eleve{
  char[30] nom;
  char[30] prenom;
  int age;
}
```

Cette séquence permet de rendre visible un nouveau type : `eleve`. On peut définir de tels nouveaux types, ainsi qu'en déclarer et initialiser des variables :

```
typedef struct {
  int jour;
  int mois;
  int annee;
} date;

date aujourdhui={1,10,2005};
```

Pour accéder à l'un des champs de la structure on utilise l'opérateur unaire `"."`, les champs se comportent alors comme des variables classiques :

```
aujourdhui.jour=2;
```

Dans le cas d'un pointeur sur une structure, on peut utiliser l'opérateur `"->"` (qui signifie champ pointé) pour accéder aux champs du pointeur :

```
date* hier;
(*hier).mois=5 /* accès
    classique à un champ de la
     structure */
hier->mois=5   /* accès
    direct au champ pointé */
```

On peut aussi imbriquer des structures :

```
struct personne{
  long int num;
  struct{
    char rue[32];
    char *ville;
  } adresse;
} fiche;
```

Les unions `union`

Les unions se déclarent de la même façon que les structures, sauf qu'on utilise le mot-clef `union` au lieu de `struct`. Leur fonctionnement est toutefois très différent : les différents champs d'une union sont tous stockés à la même adresse. Ceci permet d'implémenter des types au comportement polymorphe.

La taille d'une union est donc déterminée par la taille de son champ le plus grand. Dans cet exemple, la taille totale occupée par l'union est de 4 octets (taille d'une adresse `char*`) :

```
union personne{
  char* nom;
  char* surnom;
} voisin;

voisin.nom="Jean-Paul";
voisin.surnom="JP";
printf(voisin.nom);
```

Ce morceau de code affichera "JP". En effet, l'affectation du champ `surnom` de la variable `voisin` va «recouvrir» la valeur précédemment affectée au champ `nom`.

On notera toutefois que les types `union` sont assez peu utilisés en pratique, et de plus assez dangereux si l'on n'est pas vigilant. En effet, il est tout à fait possible de modifier un autre champ par inadvertance si jamais le champ qu'on modifie «dépasse» dessus.

2 Travail à faire

1. `Approx.c` ***À Rendre!*** Ecrire un programme `Approx.c` qui initialise une variable de type `float` à 0.0001. Puis, à l'aide d'une boucle, on additionne successivement sa valeur dans une autre variable de type `float` initialisée à zero, jusqu'à ce que la valeur de cette variable soit au moins égale à 1. Finalement, afficher le nombre d'itérations nécessaires.
   - Quel phénomène explique le résultat obtenu ?
   - Que se passe-t-il si l'on prend des variables de type `double` au lieu de `float` ?
2. `Power.c` ***À Rendre!*** Ecrire deux fonctions `powerdirect` et `powerrecur` qui prennent en argument un nombre réel x et un nombre entier p, et qui retourne x^p. On programmera la fonction `powerrecur` de manière récursive (cette fonction s'appelle elle-même, en constantant que, pour $p > 0$, $x^p = xx^{p-1}$). Tester ces fonctions.
3. `Dates.c` ***À Rendre!*** À partir de la structure date, écrire deux fonctions
   - `jours_ecoules` qui prend comme argument un objet de type `date` et qui retourne le nombre de jours écoulés depuis le 1er janvier de la même année. On pourra utiliser un tableau de 12 cases pour stocker le nombre de jours dans un mois, comme suit :

     ```
     int jours_par_mois[12] = {31,28,31,30,31,30,31,31,30,31,30,31};
     ```

   - `comparaison` qui prend comme arguments deux objets de type `date` et qui retourne :
     - -1 si la première date est avant la seconde date,
     - 0 si la première date est la même que la seconde date,
     - 1 si la première date est après la seconde date.