Programmation en C (avec connaissances en Pascal)

Alessandro Volz

Février 2004

Si vous avez des commentaires ou des corrections écrivez à `mailto:alessandro.volz@etu.unil.ch`

Ce document se concentre sur vos éventuels problèmes concernant les TP du cours de Systèmes Informatiques (cette année donné par le Prof. Jacques Menu). Les cours et les TP n'ayant jamais bien été donnés, voici donc un petit document d'introduction au C.

Vous vous posez probablement la question: quelle est la différence entre C et C++? En C il n'y a pas de classes. Il y a des autres petites différences, mais les plus grandes spécifités des deux langages ont été supprimées lors des dernières standardisations ANSI. Vous allez *most probably* programmer en C99. Si vous voulez en savoir plus, cherchez sur le web!

Une autre bonne question que vous pouvez également vous poser est: dans quel langage dois-je faire mes TPs, en C ou en C++? Je vous conseille le C parce que même si vous utilisez le C++, dans aucune des séries vous allez faire usage de son avantage, qui réside dans la définition des classes. Par contre vous verrez cette propriété des langages de programmation pendant le deuxième semestre, là où vous apprenderez Java, où leur utilisation est obligatoire.

1 Coup d'oeil sur un fichier C par rapport à un fichier Pascal.

```
program bidule;

const limite = 100;

typedef
     tableau = array [1..limite] of integer;
     blah = record
          b: integer;
          c: string;
          d: array [1..10,1..10] of integer;
          e: tableau;
          end;

var
     a: real;
     b: blah;

begin
     procedure p(a, b: integer);
     begin
          ...
     end;

     ...
end.
```

```
const int limite = 100;

typedef tableau int[limite];
typedef struct {
     int a, b;
     char c[256];
     int d[9][9];
     tableau e;
} blah;

float a;
blah b;

void p(int a, int b) {
     ...
}

int main(int argc, char *args[]) {
     ...
     return 0;
}
```

Cela devrait déja beaucoup vous aider. Remarquez qu'en C on n'a pas le droit d'emboîter des fonctions les une dans des autres! Le programme principal en C est la fonction main(), laquelle a les mêmes propriétés que les autres fonctions.

La fonction main() prend des paramètres d'entrée et retourne un entier. Les entrées vous diront les options de lancement du programme (ce qui a été écrit au terminal après le nom du programme): vous allez probablement vouloir les analyser. On verra cela plus tard.

Commentaires

- Sur une ligne: `// tout ce qui suit les // jusqu'à la fin de ligne`
- Sur plusieures lignes: `/* tout ce qui suit le /* jusqu'au prochain */ --- dans certains compilateurs on a le droit d'imbriquer ces commentaires! */`

2 Déclarations de variables, types de variables, structures.

Types simples

Entiers: char (8 bits), short (16 bits), int (32 bits), long (64 bits).
Réels: float (64 bits), double (128 bits).
Rémarquez l'absence de boolean et string!

En fait, en C un string est un tableau de char. Un char est un entier et son intéreprétation comme caractère est faite au niveau des fonctions qui traitent spécifiquement des chaines de caractères. `if ('A' == 65)` est vrai! Si cela vous intéreresse, consultez une table ASCII (voir la syntaxe des conditions plus tard).

Là où elle n'est pas explicite, on peut introduire une conversion de type par *typecasting*:
`float i = (int)(3.14*10);` mettra 6 dans i.

On verra plus tard des fonctions de traitement de strings.

Tableaux

Exemples de déclarations de tableaux:

`int tab[100];` est un tableau de 100 éléments.
`int tab[] = {0,10,5,2,1,7};` est un tableau de 6 éléments initialisés.
`int tab[x];` avec x une variable entière, est un tableau de x éléments. Rappelez-vous qu'en C vous avez le droit de déclarer des variables (presque) n'importe où!
La numérotation des tableaux en C commence à zéro: un tableau de taille x aura ses éléments de 0 à x-1.

Structures (et premier regard sur les pointeurs)

Exemple de déclaration de structure:

```
typedef struct {
      int largeur, hauteur;
} NomDeStructure;
```

C'est aussi simple que ça, cependant cela se complique un peu quand on introduit les pointeurs. Ici, la structure décrivant un noeud d'un arbre:

```
typedef struct noeud {
      Data data;
      noeud *filsg, *filsd;
} Noeud, *NoeudPtr;
```

On a dû utiliser le type dans sa même déclaration. C'est pour cette raison qu'on a écrit `noeud` après `typedef struct`. Une autre façon de faire la même chose:

```
typedef struct bidule *BidulePtr;
typedef struct bidule {
      ...
      BidulePtr next;
} Bidule;
```

On peut déclarer des variables de ce type de plusieures façons:
`Bidule a, b = {..., NULL};` a et b des instances de Bidule, b initialisé (NULL est un pointeur nul).
`BidulePtr p = nil;` est un pointeur initialisé à 0. Cela signifie qu'il ne pointe pas sur un objet valable!
`Bidule *p = nil;` est la même chose que la ligne précedante.

`BidulePtr p = malloc(sizeof(Bidule));` initialise p à l'adresse d'un nouveau bloc de memoire alloué.
`Bidule p[] = malloc(sizeof(Bidule)*x);` avec x un entier, p réprésente un tableau d'objets Bidule. `Bidule t[x];` fait la même chose!

Cependant l'allocation dynamique avec `malloc` est utile si on doit agrandir le tableau plus tard: il suffira alors d'utiliser la commande `p = realloc(p, sizeof(Bidule)*m);` avec m la nouvelle taille.

Il ne faut jamais oublier de libérer les blocs de mémoire qu'on n'utilise plus: `free(p);`

On accède aux membres d'un pointeur sur une structure par l'opérateur `->`
Si on a seulement la structure (et donc pas le pointeur), on accède aux membres par l'opérateur `.`
En fait, `s->m` corréspond à `(*s).m`

Pointeurs

Vous venez de voir les fonctions de gestion de blocs mémoire: malloc, realloc et free. Faites attention: un pointeur n'est pas un bloc mémoire alloué dynamiquement. Un pointeur est *une variable numérique de 32 bits pointant une adresse en mémoire.*

En C il existe une seule technique de passage de variables: le passage par valeur. Malgré tout, en passant la valeur du pointeur à un entier on fait ce qui est appelé *passage par référence.* Ainsi on aura le droit, dans la fonction prenant comme paramètre un pointeur sur un entier, de changer le contenu de l'entier. Idem pour le passage d'une structure (dans ce cas on est obligé de passer un pointeur!).

```
void dearboriser(NoeudPtr n) {
      ...
      dearboriser(n->filsg);
      dearboriser(n->filsd);
      free(n);
}
```

En C, un tableau est representé par un pointeur sur son premier élément (et avec la taille de ses éléments au niveau de la compilation). Lorsqu'on passe un tableau à une fonction, si cette fonction change les valeurs dans le tableau alors le tableau a changé globalement. Par exemple:

```
void incrementer(int taille, int tab[]) {
      int i;
      for (i=0; i<taille; i++) {
            tab[i]++;
}
void test() {
      int t = 7;
      int tab[t];
      incrementer(t, tab);
      // le tableau tab a changé de contenu!
}
```

Il faudra considérer ceci lorsqu'on passe des chaînes de caractères comme paramètres de fonctions!

```
void assigner(int *target, int valeur) {
      *target = valeur;
}
```

Les opérateurs sur les pointeurs sont * et &. Au début cela aura l'air compliqué mais en fait ce qui se passe est très logique: & déreference une variable, * fait l'inverse.
x est une variable
&x est son adresse en mémoire
x est la valeur entière representée à l'adresse x

```
int i = 333;
int *ip = &i;
printf("%d", *ip);
int **ipp = &ip;
printf("%d", **ipp);
```

3 Assignations, opérateurs

```
a = b+c;
a = b-c;
a = b*c;
a = b/c;
a = b%c; // en pascal, c := a mod b;

a++; ++a;
a--; --a;
```

exemples clarifiant la différence entre ces derniers opérateurs unaires:

`int a = 1, b = 333;`	`int a = 1, b = 333;`
`b = a++;`	`b = ++a;`
a vaut 2, b vaut 1	a vaut 2, b vaut 2

Même chose pour `a--` et `--a`.

```
a = 12;
a /= 2;
a *= 10;
a %= 2;
a += 1; // pareil que a++ et ++a
a -= 7;
```

Après cette séquence de commandes a vaut -6.

Opérateurs sur les bits

```
char a = 1, b = 4, c, d;
c = a&b;
d = a|b;
```

Après ces commandes, c vaut 0 et d vaut 5. Pourquoi? Voyons ces opérations de façon binaire:

```
       a = 00000001
       b = 00000100
c = a&b = 00000000
d = a|b = 00000101
```

& est un et logique appliqué à chaque n^{eme} bit de a et b. | est un ou logique appliqué à chaque n^{eme} bit de a et b.

Il y a aussi des opérateurs de décallage:

```
char a = 1, b;
b = a<<5;
b = b>>1;
```

Comme pour les autres opérateurs, il existe la version d'assignation:

```
char a = 1, b = 4;
a |= b<<2;
a <<= 3; // décaller a 3 fois vers la gauche
a &= b;
b >>= 1;
```

4 Conditions

```
if (condition) commande; else commande;
if (condition) commande;
```

commande est soit une seule commande suivie par un point-virgule, soit une suite de commandes groupées entre { et }. *condition* peut être des types suivants:

```
if (c == 'x')
if (i == 0) // même que if (!i)
if (i != 0) même que if (i)
if (a > b) même que if (b <= a)
if (a < b) même que if (b >= a)
```

Si on a défini une fonction `int bleu(int a, int b)`

```
if (bleu(1, x))
if (2>bleu(bleu(12, 0), bleu(x, x)))
```

Exemples:

```
if (x < 0) x = -x; // valeur absolue
if (x >= 0) y = sqrt(x); else ExitToShell();
if (!x) {
     x++; y = 2;
} else {
     x = 0; y /= 2;
}
```

Toujours faire attention à l'emboîtement de plusieurs conditions et à l'alignement du texte pour qu'on comprenne tout de suite l'algorithme associé!

Une instruction switch peut être utilisée pour remplacer un ensemble d'instructions if. Par exemple,

```
if (x == 1')
     a = f();
else if (x == 2)
     a = g();
else
     a = h();
```

s'écrit également:

```
switch (x) {
     case 1:
          a = f();
          break;
     case 2:
          a = g();
          break;
     default:
          a = h();
}
```

Les différents cas doivent être constants. Par exemple, `case var:` avec var une variable n'est pas permis.
L'*effet de chute* est ce qui se passe lorsqu'on n'écrit pas de `break`: un switch execute le cas selectionné et tout le code qui suit (dans les autres cas aussi) jusqu'au premier `break`.

Combinaisons

```
if (condition && condition) ...
if (condition || condition) ...
```

Faites attention à ne pas mélanger `&&` et `||` avec `&` et `|`.
Les différents opérateurs de comparaison n'évaluent le second argument d'une condition que si cela s'avère nécessaire.

4.1 Spécial

On peut mettre des petites conditions n'importe où: `condition?alors:sinon` peut par exemple être utilisé comme suit:

```
printf("Mike has %d monkey%s\n", m, m==1?"":"s");
```

5 Boucles

Il en existe 3 types: while, for et do.

```
while (condition) commande;
while (condition) {
      commandes; ...
      if (cond) break; // la boucle arrête
      if (cond) continue; // la boucle reprend à partir du contrôle de la condition
}

for (init; cond; incr) commande;
for(i=0; i<max; i++)
      for (j=0; j<max; j++) {
            commandes sur tab[i][j];
      }

do commande; while (condition);
do {
      commandes;
} while (condition);
```

6 Fonctions (et procédures)

En C une procédure est une fonction qui rétourne une valeur de type `void` - rien. Par contre, faites attention que `void *` est le type générique pour les pointeurs.

```
int moo(variables en entrée) { ... }
float goo(...)  { ... }
void proc(...)  { ... }
```

La valeur retournée est retournée grâce à la commande *return*. Celle-ci peut être effectuée à n'importe quel moment dans la fonction mais obligatoirement comme dernière commande.

```
return 0;
return sqrt(x);
return; // pour les void
```

Des fonctions predéfinies

Une liste de fonctions dont vous aurez probablement besoin. Pour en savoir plus, utilisez la commande `man 2 <commande>`.

Chaînes de caractères: `memcpy, memmove, strcpy, strlen, strcat, strchr, strstr, strcmp, strncmp, strspn, strcspn, ...` (voir *string.h*).
Référez-vous au site `http://www.cppreference.com/` dans la section *Standard C Library*.

Ce document a été écrit en LaTeX.
Merci à Adrien Chantre et Gregory Loichot.