Université Savoie Mont Blanc,
UFR SceM
INFO504, L3 INFO

September 29, 2020
J.-O. Lachaud

Fiche 1 — les fonctions en C

(Déclaration, définition, passage de paramètres, récursivité)

1 Fonctions et procédure en C

1.1 Fonctions en programmation impérative

Les fonctions structurent le code des programmes impératifs. Elles permettent de découper un programme complexe en un ensemble de blocs d'instructions plus simples et plus courts, avec un objectif plus limitée de transformation de données en entrées vers une ou plusieurs données de sorties.

Par exemple, on veut faire un programme qui lit un fichier formé de nombres, trie ces nombres par ordre croissant, puis écrit un nouveau fichier formé de ces nombres dans l'ordre. Plutôt que de tout faire dans une fonction `main`, on découpera le code en des blocs de traitement bien identifiés et réutilisables:

- une fonction capable de lire un fichier et de placer les données lues dans un tableau;
- une fonction capable de trier un tableau de nombres par ordre croissant;
- une fonction capable d'écrire un tableau de nombres dans un fichier;
- et le programme principal (fonction `main`) qui récupère en paramètres sur la ligne de commande les noms des fichiers à lire et à écrire, et qui appelle en séquence les trois fonctions.

Cette bonne pratique est *fondamentale* et ubiquitaire dans le métier d'informaticien.

A noter qu'en C on ne différencie pas *procédure* (bloc d'instructions qui reçoit des paramètres donnés mais qui ne retourne pas de valeur de retour) et *fonction* (bloc d'instructions qui traite des paramètres donnés sans les modifier et retourne une valeur en retour).

1.2 Déclaration, appel et définition des fonctions

Pour utiliser une fonction, il faut qu'elle ait été déclarée au préalable. Le code associé à la fonction n'est pas nécessairement encore connu.

déclaration La syntaxe générale de la déclaration d'une fonction prend la forme:

<*type-val-retour*> <*nom-fonction*> `(` <*type1*> <*nom1*> `,` <*type2*> <*nom2*> `, ...);`

où <*nom-fonction*> désigne le nom de la fonction (utilisé pour appeler la fonction ailleurs dans le code), et <*nomx*> désigne les noms des *paramètres (formels)* donnés à la fonction en entrée.

appel Une fois déclarée, la fonction peut être appelée dans la suite du programme source, simplement en écrivant le nom de la fonction, et entre parenthèses, les *arguments d'appel* qui correspondent un pour un aux paramètres :

<*nom-fonction*> `(` <*arg1*> `,` <*arg2*> `, ...)`

Notez qu'il s'agit d'une expression, dont le type est <*type-val-retour*>.

définition Enfin, pour définir une fonction, il suffit d'écrire la même ligne que sa déclaration (sans le point-virgule) mais de mettre entre accolades les instructions correspondent à la fonction.

<*type-val-retour*> <*nom-fonction*> `(` <*type1*> <*nom1*> `,` <*type2*> <*nom2*> `, ...)`
`{ ... }`

valeur retournée Une fonction qui retourne une valeur doit avoir (au moins) une instruction `return` <*expr*>`;`, où <*expr*> est une expression de type <*type-val-retour*>.

procédure Une fonction qui ne retourne pas de valeur a son <*type-val-retour*> qui est `void`.

On s'en sert alors comme une instruction.

Un exemple vaut mieux qu'un long discours:

```
// déclaration de f
void f( int i ); // indique l'existence d'une fonction f,
                 // prenant un entier en paramètre.

int main( void )
{
  int a = 3;
  f( 7 ); // 1er appel de f,  7 de type int
  f( a ); // 2ème appel de f, a de type int
  return 0; // valeur retournée à la fonction appelante, ici le shell
}

// Définition de f
void f( int i )
{
  printf( "%d au carré = %d\n", i, i*i );
}
```

s'exécute ainsi:

```
7 au carré = 49
3 au carré = 9
```

2 Passage de paramètres par valeur

Dans une fonction, les paramètres formels reçoivent les valeurs des arguments d'appel au moment de l'appel. Les paramètres formels sont des *variables locales,* dont l'existence est bornée par le bloc de définition de la fonction.

Le C ne connaît que le **passage par valeur** (types de base, struct, union, pointeurs). Ainsi l'exemple classique de l'échange de variable ne marche pas.

```c
void echange( int i, int j )
{             // i = 4, j = 8
  int t = i; // i = 4, j = 8, t = 4
  i = j;     // i = 8, j = 8, t = 4
  j = t;     // i = 8, j = 4, t = 4
} // i, j et t sont détruits automatiquement

int main( void )
{
  int x = 4;
  int y = 8;
  echange( x, y );
  // x et y sont inchangés.
}
```

On pourrait croire alors qu'une fonction ne pourra jamais modifier des arguments donnés à l'appel. En réalité, la possibilité en C de pouvoir accéder à l'adresse en mémoire d'une variable permettra de le faire !

Evidemment, cela nécessitera soit d'utiliser des *pointeurs*, soit des tableaux (qui en tant que paramètres se comportent comme des pointeurs). On reviendra dessus plus tard au moment des pointeurs, sachez que ce type de passage de paramètre s'appelle passage par adresse, qui n'est rien d'autre qu'un passage par valeur d'adresses en mémoire.

Exemple : Fonction calculant x^k

```c
double power( double x, int k )
{
  double r = 1.0;
  for ( ; k > 0; k-- )
    r *= x;
  return r;
}
```

Question 1. Adaptez le code précédent pour qu'il fonctionne avec un entier k possiblement négatif.

Question 2. On peut calculer plus rapidement x^k grâce à l'exponentiation rapide. Le principe est de remarquer que si $k = 2i$ est pair alors $x^k = x^{2i} = (x^2)^i$, et que si $k = 2i + 1$ est impair alors $x^k = x^{2i+1} = x * (x^2)^i$.

Question 3. Ecrivez une fonction qui calcule le bit le plus significatif d'un entier `int`, autrement $[log_2(n)]$.

Question 4. Ecrivez une fonction qui calcule le bit le moins significatif d'un entier `int`.

Question 5. Ecrivez une fonction qui calcule le nombre de bits à 1 d'un entier `int`.

Question 6. Ecrivez une fonction qui compte le nombre de triplets pythagoriciens de plus grand élément inférieur à un n donné, c'est-à-dire les triplets d'entiers (a, b, c) tels que $a^2 + b^2 = c^2$. Ces triplets sont en lien avec des rotations de points entiers vers des points entiers.

3 Récursivité

La façon dont les fonctions s'appellent les unes les autres forment un graphe, appelé graphe d'appel des fonctions. Lorsque ce graphe contient une boucle partant d'une fonction, on dit que la fonction est *récursive*. La récursivité n'est pas vraiment beaucoup plus lente qu'une boucle équivalente, quoique toute fonction récursive peut être transformée en boucle avec une structure de données auxiliaire (une pile). Elle permet surtout des écritures très concises et, pour qui est habitué, très clair, souvent proches des définitions abstraites mathématiques. La récursivité est le socle de la programmation fonctionnelle.

Notez que toute fonction récursive doit contenir un test conditionnel dit test de terminaison. Très souvent un des paramètres est entier et est garanti de décroître à chaque appel récursif. Le test de terminaison teste alors ce paramètre par rapport à zéro et ne rappelle plus la fonction dans ce cas.

Exemple : Fonction calculant $n!$ pour n entier

```
long long int fact( long long int n )
{
  if ( n > 0 ) return n * fact( n-1 );
  else         return 1;
}
```

On peut l'appeler ainsi. Observez jusqu'à quel entier le résultat est correct.

```
int main( int argc, char* argv[] )
{
  int i = atoi( argv[ 1 ] );
  printf( "%d! = %lld\n", i, fact( i ) );
  return 0;
}
```

```
18! = 6402373705728000     # ok
30! = -8764578968847253504 # clairement faux
```

Question 7. Ecrivez la fonction `double power( double x, int k )` en récursif.

Question 8. Ecrivez le calcul du pgcd(a, b) (plus grand commun diviseur) par l'algorithme d'Euclide en itératif et en récursif. On rappelle qu'il existe une version soustractive basée sur la relation pgcd(a, b) = pgcd$(a - b, b)$ lorsque $a > b > 0$. Il existe aussi une version par divisions successives basée sur la relation pgcd(a, b) = pgcd$(a\%b, b)$ lorsque $a > b > 0$, en sachant que $a\%b$ désigne le reste de la division euclidienne de a par b. L'algorithme stoppe lorsqu'un paramètre vaut 0, et le pgcd est la valeur de l'autre paramètre.

Question 9. Il existe une version "binaire" du calcul du pgcd(a, b) de deux nombres. Elle est basée sur les relations suivantes (mettons $a > b$):

- si a et b sont tous deux pairs, alors pgcd$(a, b) = 2 *$ pgcd$(a/2, b/2)$
- si a est pair et b impair, alors pgcd(a, b) = pgcd$(a/2, b)$
- si a est impair et b pair, alors pgcd(a, b) = pgcd$(a, b/2)$
- si a et b sont impairs, alors pgcd(a, b) = pgcd$((a - b)/2, b)$
- si b vaut 0, pgcd$(a, 0) = a$.

Ecrivez la fonction correspondante de manières itérative et récursive. Quel intérêt présente cette façon de calculer le pgcd par rapport à la méthode d'Euclide ?