Les fonctions

MP2I - Informatique

Anthony Lick
Lycée Janson de Sailly

Les fonctions

Comme dans la plupart des langages, on peut en C découper un programme en plusieurs fonctions.

Une seule de ces fonctions existe obligatoirement ; c'est la fonction principale appelée **`main`**.

Cette fonction principale peut, éventuellement, appeler une ou plusieurs fonctions secondaires.

De même, chaque fonction secondaire peut appeler d'autres fonctions secondaires ou s'appeler elle-même.

Dans ce dernier cas, on parle alors de fonction **récursive**.

Définition d'une fonction

Définition d'une fonction

```
1  type nom_fonction(type1 arg1, ..., typeN argN)
2  {
3    [ déclaration de variables locales ]
4    instructions
5  }
```

Définition d'une fonction

La **définition** d'une fonction est la donnée :

- du texte de son algorithme, appelé **corps** de la fonction,
- ainsi que de sa **signature** (nombre d'arguments, types des arguments, type de retour).

Définition d'une fonction

```
1  type nom_fonction(type1 arg1, ..., typeN argN)
2  {
3    [ déclaration de variables locales ]
4    instructions
5  }
```

Entête

La première ligne ci-dessus est l'**entête** de la fonction, qui précise sa **signature**.

type est le **type de retour** de la fonction.

Si une fonction ne renvoie pas de valeur, son type de retour est spécifié par le type **`void`**.

Définition d'une fonction

```
1  type nom_fonction(type1 arg1, ..., typeN argN)
2  {
3    [ déclaration de variables locales ]
4    instructions
5  }
```

Paramètres

Les arguments de la fonction sont appelés **paramètres formels**, par opposition aux **paramètres effectifs** qui sont les paramètres avec lesquels la fonction est réellement appelée.
Le type de chaque paramètre formel doit être spécifié.
Leurs identificateurs n'ont d'importance qu'à l'intérieur de la fonction.

Définition d'une fonction

```
1  type nom_fonction(type1 arg1, ..., typeN argN)
2  {
3    [ déclaration de variables locales ]
4    instructions
5  }
```

Variables locales

Le corps de la fonction débute éventuellement par la déclaration de variables, qui sont **locales** à cette fonction.

Définition d'une fonction

```
1  type nom_fonction(type1 arg1, ..., typeN argN)
2  {
3    [ déclaration de variables locales ]
4    instructions
5  }
```

Valeur de retour

La valeur de retour est spécifiée par l'**instruction de retour à la fonction appelante**, **`return`**, dont la syntaxe est :

```
return(expression);
```

La première instruction **`return`** rencontrée interrompt l'exécution de la fonction, et le programme retourne alors à l'endroit du code où la fonction a été appelée.

Définition d'une fonction

Exemple

```
1  int produit (int a, int b)
2  {
3    return(a*b);
4  }
```

Exemple

La fonction ci-dessus calcule le produit de deux entiers.

Définition d'une fonction

Version impérative

```c
int puissance(int x, int n)
{
  int res = 1;
  int i;
  for (i = 0; i < n; i++)
  {
    res *= x;
  }
  return(res);
}
```

Version récursive

```c
int puissance(int x, int n)
{
  if (n == 0)
  {
    return(1);
  }
  return(x * puissance(x, n-1));
}
```

Exemple

Les deux fonctions ci-dessus calculent x^n.

Celle de gauche de manière **impérative** ; celle de droite de manière **récursive**.

Définition d'une fonction

Exemple

```c
void imprime_tab(int *tab, int nb_elements)
{
  int i;
  for (i = 0; i < nb_elements; i++)
  {
    printf("%d \t", tab[i]);
  }
  printf("\n");
}
```

Exemple

La fonction ci-dessus prend en arguments un pointeur vers un tableau d'entiers et la longueur du tableau, et affiche les éléments du tableau.

Appel d'une fonction

Appel d'une fonction

Appel de fonction

L'**appel** d'une fonction se fait par l'expression :

```
nom_fonction(para1, para2, ..., paraN)
```

L'ordre et le type des **paramètres effectifs** doivent concorder avec ceux donnés dans l'entête de la fonction.

Les **paramètres effectifs** peuvent être des expressions.

Déclaration d'une fonction

Déclaration d'une fonction

Déclaration

Le C n'autorise pas les fonctions **imbriquées**, c'est-à-dire de définir une fonction dans le corps d'une autre fonction.

La définition d'une fonction secondaire doit donc être placée soit avant, soit après la fonction **`main`**.

Toutefois, il est indispensable que le compilateur "connaisse" la fonction au moment où celle-ci est appelée.

Si une fonction est **définie** après son premier appel (en particulier, si sa définition est placée après la fonction **`main`**), elle doit impérativement être **déclarée** au préalable.

Déclaration d'une fonction

Prototype d'une fonction

Une fonction secondaire est déclarée par son **prototype**, qui spécifie sa **signature** (mais sans le corps de la fonction). Cela ce fait avec l'instruction suivante :

```
type nom_fonction(type1, ..., typeN);
```

Déclaration d'une fonction

Remarque

Le **prototype** correspond donc à l'**entête** d'une fonction, mais sans les noms des paramètres (uniquement les types).

En fait, on peut laisser les noms des paramètres, ce qui est pratique : il suffit de copier la ligne d'entête d'une fonction pour ensuite coller son prototype au bon endroit.

Dans ce cas, n'oubliez pas de rajouter le **;** à la fin de la ligne !

Déclaration d'une fonction

Exemple

```c
#include <stdio.h>

int puissance(int, int);

int main()
{
  int a = 2, b = 5;
  printf("%d\n", puissance(a,b));
}

int puissance(int x, int n)
{
  if (n == 0)
    return(1);
  return(x * puissance(x, n-1));
}
```

Déclaration d'une fonction

Remarque

Même si la déclaration est parfois facultative (par exemple quand les fonctions sont définies avant la fonction **`main`** et dans le bon ordre), elle seule permet au compilateur de vérifier que le nombre et le type des paramètres utilisés dans la **définition** concordent bien avec le **protype**.

De plus, la présence d'une déclaration permet au compilateur de mettre en place d'éventuelles conversions des paramètres effectifs, lorsque la fonction est appelée avec des paramètres dont les types ne correspondent pas aux types indiqués dans le prototype.

Déclaration d'une fonction

Fichiers header

Lorsqu'on organise notre programme en plusieurs fichiers (**programmation modulaire**), on sépare notre code entre des fichiers **`.c`** qui contiennent notre code (et en particulier les définitions de nos fonctions), et les fichiers **`.h`** (fichier **header**) qui contiennent uniquement les prototypes de nos fonctions.

En pratique, un fichier **`mon_fichier.h`** doit être accompagné d'un fichier **`mon_fichier.c`** portant le même nom (seule l'extension du fichier change).

Déclaration d'une fonction

Directive de préprocesseur

```
1  #include <math.h>
```

Exemple

On trouve dans le fichier `math.h` le prototype de la fonction **pow** (élévation à la puissance) :

```
double pow(double, double);
```

Ainsi, la directive de préprocesseur ci-dessus permet d'inclure la déclaration de la fonction **pow** (et des autres fonctions de la librairie **math.h**). De plus, si la fonction **pow** est utilisée avec des paramètres de type **int**, ces paramètres seront automatiquement convertis en **double** à la compilation.

Déclaration d'une fonction

main.c

```c
#include <stdio.h> // entre < > pour une librairie installée
#include "arith.h" // entre " " pour un fichier se trouvant dans le dossier courant

int main()
{
  int a = 2, b = 5;
  printf("%d\n", puissance(a,b));
}
```

arith.c

```c
int puissance(int x, int n)
{
  if (n == 0)
    return(1);
  return(x * puissance(x, n-1));
}
```

arith.h

```c
int puissance(int x, int n);
```

Console

```
janson@mp2i-janson:~$ gcc main.c arith.c -o test
janson@mp2i-janson:~$ ./test
32
```

Durée de vie des variables

Durée de vie des variables

Variables

Les variables manipulées dans un programme C ne sont pas toutes traitées de la même manière.

En particulier, elles n'ont pas toutes la même **durée de vie**.

Allocation statique ou dynamique

Allocation

Un programme compilé gère la mémoire d'un ordinateur de deux manières très différentes :

- **statiquement** : c'est le cas des variables locales ou globales définies dans le programme. Au moment de la compilation, le compilateur dispose de l'information suffisante pour prévoir de la place en mémoire pour stocker ces données.
- **dynamiquement** : c'est le cas des objets dont la taille n'est connue qu'à l'exécution et peut varier selon l'état du programme. C'est alors au moment de l'exécution que le programme va faire une demande d'allocation pour obtenir une place mémoire.

Allocation statique ou dynamique

Gestion de la mémoire

En terme d'**allocation statique**, on peut distinguer plusieurs types de mémoire :

- les variables **globales** initialisées qui seront stockées dans le binaire et placées en mémoire dans une zone spécifique chargée avec le binaire ;
- les variables **globales** non initialisées dont seule la déclaration sera dans le binaire et qui seront allouées, placées en mémoire et initialisées à 0 au moment du chargement du binaire ;
- les variables **locales** et les **paramètres** qui sont placés dans une structure de **pile** (dont nous reparlerons plus tard) afin de les allouer uniquement au moment de l'exécution du bloc ou de la fonction.

Allocation statique ou dynamique

Gestion de la mémoire

L'allocation **dynamique** utilise une zone mémoire appelée **tas** (dont nous verrons également une implémentation plus tard).

Portée et durée de vie des variables

Portée

La **portée** d'un identificateur est définie par la zone du texte d'un programme dans laquelle il est possible d'y faire référence sans erreur.

Durée de vie

La **durée de vie** d'une variable correspond à la période de son exécution durant laquelle la variable est présente en mémoire.

Portée des variables

```
1   int a = 1;
2
3   int f(int x)
4   {
5     int y = x + a;
6     return y;
7   }
8
9   int g()
10  {
11    int z = 3;
12    return z + f(z);
13  }
```

Identificateur	Portée
a	$1 - 13$
x	$3 - 7$
y	$5 - 7$
f	$4 - 13$
g	$10 - 13$
z	$11 - 13$

Remarque

Afin de pouvoir écrire des fonctions **récursives**, l'identificateur d'une fonction est utilisable dans le corps de la fonction.

Variables globales

Variable globale

On appelle **variable globale** une variable déclarée en dehors de toute fonction.

L'identificateur associé à une variable globale a pour **portée** l'ensemble des lignes suivant sa déclaration.

La **durée de vie** d'une variable globale est l'intégralité du programme.

Variables globales

Exemple

```c
#include <stdio.h>

int n;

void fonction();

int main()
{
  int i;
  for (i = 0; i < 5; i++)
  {
    fonction();
  }
}

void fonction()
{
  n++;
  printf("appel numero %d\n", n);
}
```

Output

```
appel numero 1
appel numero 2
appel numero 3
appel numero 4
appel numero 5
```

Exemple

Dans le programme ci-contre, **n** est une variable **globale**.

Variables locales

Variable locale

On appelle **variable locale** une variable déclarée à l'intérieur d'une **fonction** (ou d'un **bloc d'instructions**).

Par défaut, les variables locales sont **temporaires**.

Quand une fonction est appelée, elle place ses variables locales dans la **pile**.

À la sortie de la fonction, les variables locales sont **dépilées** (et donc **perdues**).

Attention

Les **variables locales** n'ont en particulier aucun lien avec les **variables globales** du même nom.

Variables locales

Exemple

```c
#include <stdio.h>

int n = 10;

void fonction();

int main()
{
  int i;
  for (i = 0; i < 5; i++)
    fonction();
}

void fonction()
{
  int n = 0;
  n++;
  printf("appel numero %d\n",n);
  return;
}
```

Output

```
appel numero 1
appel numero 1
appel numero 1
appel numero 1
appel numero 1
```

Exemple

Dans le programme ci-contre, la variable **globale n** définie ligne 3 est **masquée** par la variable **locale** ligne 16.

Variables locales

static

Les variables locales à une fonction ont une durée de vie limitée à une seule exécution de cette fonction.

Leurs valeurs ne sont pas conservées d'un appel au suivant.

Il est toutefois possible de créer une variable locale de classe **statique** en faisant précéder sa déclaration du mot clé `static` :

```
static type nom_variable;
```

Une telle variable reste locale à la fonction dans laquelle est est déclarée, mais sa valeur est conservée d'un appel au suivant. Elle est également initialisée à zéro à la compilation.

Variables locales

Exemple

```c
#include <stdio.h>

int n = 10;

void fonction();

int main()
{
  int i;
  for (i = 0; i < 5; i++)
    fonction();
}

void fonction()
{
  static int n;
  n++;
  printf("appel numero %d\n",n);
  return;
}
```

Output

```
appel numero 1
appel numero 2
appel numero 3
appel numero 4
appel numero 5
```

Exemple

Dans le programme ci-contre, la variable **n** définie à la ligne 16 est de classe **statique** : elle est initialisée à 0, et sa valeur est conservée d'un appel à l'autre de la fonction.

Variables locales

Exemple

```c
#include <stdio.h>

int n = 10;

void fonction();

int main()
{
  int i;
  for (i = 0; i < 5; i++)
    fonction();
}

void fonction()
{
  static int n;
  n++;
  printf("appel numero %d\n",n);
  return;
}
```

Output

```
appel numero 1
appel numero 2
appel numero 3
appel numero 4
appel numero 5
```

Exemple

En revanche, il s'agit bien d'une variable **locale**, qui n'a aucun lien avec la variable **globale** déclarée à la ligne 3.

Transmission des paramètres dans une fonction

Transmission de paramètres d'une fonction

Paramètres effectifs

Les **paramètres** d'une fonction sont traités de la même manière que les variables locales : lors de l'appel de la fonction, les **paramètres effectifs** sont copiés dans la **pile d'appel**.

Cette copie disparait lors du retour du programme appelant. Cela implique en particulier que, si la fonction modifie la valeur d'un de ses paramètres, seule la copie sera modifiée ; la variable du **programme appelant**, elle, ne sera pas modifiée.

On dit que les paramètres d'une fonction sont **transmis par valeurs**.

Transmission de paramètres d'une fonction

Exemple

```c
#include <stdio.h>

void echange(int, int);

int main()
{
  int a = 2, b = 5;
  printf("debut programme principal :\t a = %d \t b = %d\n",a,b);
  echange(a,b);
  printf("fin programme principal :\t a = %d \t b = %d\n",a,b);
}

void echange(int a, int b)
{
  int t;
  printf("debut fonction :\t\t a = %d \t b = %d\n",a,b);
  t = a;
  a = b;
  b = t;
  printf("fin fonction :\t\t\t a = %d \t b = %d\n",a,b);
}
```

Output

```
debut programme principal :      a = 2   b = 5
debut fonction :                 a = 2   b = 5
fin fonction :                   a = 5   b = 2
fin programme principal :        a = 2   b = 5
```

Transmission de paramètres d'une fonction

Pointeurs et effets de bords

Pour qu'une fonction modifie la valeur d'un de ses arguments, il faut qu'elle ait pour paramètre l'**adresse** de cet objet et non sa valeur.

Dans ce cas, on dit que la fonction produit des **effets de bords**.

Transmission de paramètres d'une fonction

Exemple

```c
#include <stdio.h>

void echange(int *, int *);

int main()
{
  int a = 2, b = 5;
  printf("debut programme principal :\t a = %d \t b = %d\n",a,b);
  echange(&a,&b);
  printf("fin programme principal :\t a = %d \t b = %d\n",a,b);
}

void echange(int *adr_a, int *adr_b)
{
  int t;
  t = *adr_a;
  *adr_a = *adr_b;
  *adr_b = t;
}
```

Output

```
debut programme principal :      a = 2   b = 5
fin programme principal :        a = 5   b = 2
```

Transmission de paramètres d'une fonction

Tableaux et effets de bords

Rappelons qu'un **tableau** est un pointeur (sur le premier élément du tableau).

Lorsqu'un tableau est transmis comme paramètre à une fonction secondaire, ses éléments sont donc modifiés par la fonction.

Transmission de paramètres d'une fonction

Exemple

```c
#include <stdlib.h>

void init(int *, int);

int main()
{
  int n = 5;
  int *tab;
  tab = malloc(n * sizeof(int));
  init(tab,n);
}

void init(int *tab, int n)
{
  int i;
  for (i = 0; i < n; i++)
    tab[i] = i;
  return;
}
```

Exemple

La fonction ci-dessus initialise les éléments du tableau **tab**.

La fonction `main`

La fonction `main`

`main`

Le type de retour de la fonction **`main`** (celle qui est appelée lors de l'exécution du programme compilé) est **`int`**.

La valeur renvoyée par **`main`** est transmis à l'**environnement d'exécution** (le **terminal** par exemple) :

- la valeur de retour 0 indique que le programme s'est terminé sans erreur ;
- la valeur de retour 1 correspond à une terminaison sur une erreur.

La fonction `main`

stdlib.h

On peut utiliser comme valeur de retour les deux constantes **EXIT_SUCCESS** (égale à 0) et **EXIT_FAILURE** (égale à 1), qui sont définies dans **stdlib.h**.

Si **statut** est un entier spécifiant le type de terminaison du programme, on peut également remplacer l'instruction `return(statut);` par l'appel de fonction **exit(statut);** (la fonction **exit** étant définie dans **stdlib.h**).

La fonction `main`

Console

```
janson@mp2i:~$ ./executable argument1 argument2 ... argumentN
```

Paramètres de la fonction `main`

la fonction **`main`** peut également posséder des **paramètres formels**.

En effet, un programme C peut recevoir une liste d'arguments au lancement de son exécution, comme ci-dessus.

Ces arguments sont alors transmis à la fonction **`main`** par l'interpréteur de commandes.

La fonction `main`

Console

```
janson@mp2i:~$ ./executable argument1 argument2 ... argumentN
```

Paramètres de la fonction `main`

En fait, la fonction **`main`** peut posséder deux **paramètres formels**, appelés par convention **`argc`** (**argument count**) et **`argv`** (**argument vector**).

- **`argc`** est une variable de type **`int`** dont la valeur est égale au nombre de mots composant la ligne de commande (y compris le nom de l'exécutable). Elle est donc égale au nombre de paramètres effectifs de la fonction $+1$.

La fonction `main`

Console

```
janson@mp2i:~$ ./executable argument1 argument2 ... argumentN
```

Paramètres de la fonction `main`

En fait, la fonction **`main`** peut posséder deux **paramètres formels**, appelés par convention **`argc`** (**argument count**) et **`argv`** (**argument vector**).

- **`argv`** est un tableau de chaînes de caractères dont chaque case correspond à un mot de la ligne de commande. **`argv[0]`** contient donc le nom de la commande (du fichier exécutable), **`argv[1]`** contient le premier paramètre, ...

La fonction `main`

```
1  int main(int argc, char *argv[]);
```

Prototype de la fonction `main`

La fonction **`main`** peut donc avoir le prototype ci-dessus.

La fonction `main`

Attention

Chaque paramètre est donné sous forme d'une chaîne de caractères. Il faut donc les convertir dans le bon type avant de s'en servir.

La librairie **`stdlib.h`** fournit des fonctions pour convertir ces arguments, comme la fonction **`atoi`** (**ASCII to int**) qui convertie une chaîne de caractères en entier.

La fonction main

exemple.c

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
  int a, b;
  if (argc != 3)
  {
    printf("Erreur : nombre invalide d'arguments\n");
    printf("Usage: %s int int\n",argv[0]);
    return(EXIT_FAILURE);
  }
  a = atoi(argv[1]);
  b = atoi(argv[2]);
  printf("Le produit de %d par %d vaut : %d\n", a, b, a * b);
  return(EXIT_SUCCESS);
}
```

Console

```
janson@mp2i:~$ gcc exemple.c -o produit
janson@mp2i:~$ ./produit
Erreur : nombre invalide d'arguments
Usage: ./produit int int
janson@mp2i:~$ ./produit 12 8
Le produit de 12 par 8 vaut : 96
```

Fonction comme paramètre d'une autre fonction

Fonction comme paramètre d'une autre fonction

Fonction en argument

Il est parfois utile de passer une fonction comme paramètre d'une autre fonction.

Cette procédure permet en particulier d'utiliser une même fonction pour différents usages.

Dans ce cas, pour le **paramètre formel** en question, il faut indiquer la **signature** de la fonction attendue en argument.

Fonction comme paramètre d'une autre fonction

Exemple

```c
#include <stdio.h>

int somme(int a, int b)
{
  return a + b;
}

int produit(int a, int b)
{
  return a * b;
}

int op_binaire(int a, int b, int f(int, int))
{
  return f(a,b);
}

int main()
{
  printf("%d\n", op_binaire(2, 3, somme));
  printf("%d\n", op_binaire(2, 3, produit));
}
```

Output

```
5
6
```

Exemple

La fonction **op_binaire** prend en arguments deux entiers a et b, et une fonction $f : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$. Elle renvoie la valeur de $f(a, b)$.

Fonction comme paramètre d'une autre fonction

`strcmp`

La librairie **`string.h`** fournit une fonction **`strcmp`** qui prend en entrée deux chaînes de caractères **`s1`** et **`s2`**, et qui renvoie un entier tel que :

- **`strcmp(s1, s2)`** vaut 0 si **`s1`** et **`s2`** sont égales ;
- **`strcmp(s1, s2)`** est < 0 si **`s1`** est strictement inférieure à **`s2`** pour l'ordre lexicographique ;
- **`strcmp(s1, s2)`** est > 0 si **`s1`** est strictement supérieure à **`s2`** pour l'ordre lexicographique.

L'**ordre lexicographique** est une généralisation de l'**ordre alphabétique**.

Fonction comme paramètre d'une autre fonction

exemple.c (1/2)

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int somme(int a, int b)
{
  return a + b;
}

int produit(int a, int b)
{
  return a * b;
}

int op_binaire(int a, int b, int f(int, int))
{
  return f(a,b);
}

void usage(char *cmd)
{
  printf("Usage : %s int [plus|fois] int\n", cmd);
}
```

Exemple

Pour que notre programme puisse être exécuté en ligne de commande avec des paramètres, on peut utiliser les fonctions :

- **atoi** de **stdlib.h** ;
- **strcmp** de **string.h**.

Fonction comme paramètre d'une autre fonction

exemple.c (2/2)

```c
int main(int argc, char *argv[])
{
  int a, b;
  if (argc != 4)
  {
    printf("Erreur : nombre invalide d'arguments\n");
    usage(argv[0]);
    return(EXIT_FAILURE);
  }
  a = atoi(argv[1]);
  b = atoi(argv[3]);
  if (strcmp(argv[2], "plus") == 0)
  {
    printf("%d\n", op_binaire(a, b, somme));
    return(EXIT_SUCCESS);
  }
  if (strcmp(argv[2], "fois") == 0)
  {
    printf("%d\n", op_binaire(a, b, produit));
    return(EXIT_SUCCESS);
  }
  else
  {
    printf("Erreur : argument(s) invalide(s)\n");
    usage(argv[0]);
    return(EXIT_FAILURE);
  }
}
```

Console

```
janson@mp2i:~$ gcc exemple.c -o calcul
janson@mp2i:~$ ./calcul
Erreur : nombre invalide d'arguments
Usage : ./calcul int [plus|fois] int
janson@mp2i:~$ ./calcul 2 plus 3
5
janson@mp2i:~$ ./calcul 2 fois 3
6
janson@mp2i:~$ ./calcul 2 oups 3
Erreur : argument(s) invalide(s)
Usage : ./calcul int [plus|fois] int
```

Fonction comme paramètre d'une autre fonction

```
1  void qsort(void *tab, size_t nb_elem, size_t taille_elem, int comp(const void*, const void*));
```

Exemple : Fonctions de tris

Passer une fonction en argument est notamment utile lorsqu'on veut **trier** des éléments d'un tableau, mais qu'on veut spécifier pour **quel ordre** on veut le trier.

C'est le cas par exemple de la fonction **`qsort`** (pour **quick sort**) de la librairie **`stdlib.h`** dont le **prototype** est celui ci-dessus.

Le dernier paramètre **`comp`** doit être une fonction décrivant la relation d'ordre voulue, de la manière suivante :

- **`comp(a,b) == 0`** si **`*a == *b`** ;
- **`comp(a,b) < 0`** si **`*a < *b`** ;
- **`comp(a,b) > 0`** si **`*a > *b`**.

Renvoyer plusieurs valeurs avec une fonction

Renvoyer plusieurs valeurs avec une fonction

Limitation

En C, la notion de couple **n'existe pas**.

On ne peut donc pas renvoyer facilement plusieurs valeurs dans un `return` (comme on pourrait le faire en **Python** par exemple).

Renvoyer plusieurs valeurs avec une fonction

Utilisation de structures

Pour renvoyer plusieurs valeurs, une première possibilité est de créer une **structure** adéquate, contenant toutes les valeurs qui nous intéresse.

Mais cette solution n'est souvent pas la plus naturelle, car nous risquons de créer des structures qui servent uniquement comme valeur de retour à une fonction...

Renvoyer plusieurs valeurs avec une fonction

exemple.c (1/2)

```c
#include <stdio.h>
#include <math.h> // sqrt est la fonction "racine carrée"

struct retour_racines
{
  int nb;
  float r1;
  float r2;
};
typedef struct retour_racines retour_racines;

retour_racines racines_reelles(float a, float b, float c);

int main()
{
  retour_racines r = racines_reelles(-1, 2, 3);
  printf("Nombre de racines : %d\n",r.nb);
  if (r.nb > 0)
  {
    printf("Racines : %f, %f\n", r.r1, r.r2);
  }
}
```

Renvoyer plusieurs valeurs avec une fonction

exemple.c (2/2)

```c
retour_racines racines_reelles(float a, float b, float c)
{
  retour_racines res;
  float delta = b*b - 4*a*c;
  if (delta == 0)
  {
    res.nb = 1;
    float r = -b / (2*a);
    res.r1 = r;
    res.r2 = r;
  }
  else if (delta > 0)
  {
    res.nb = 2;
    res.r1 = (-b - sqrt(delta)) / (2*a);
    res.r2 = (-b + sqrt(delta)) / (2*a);
  }
  else
  {
    res.nb = 0;
  }
  return res;
}
```

Renvoyer plusieurs valeurs avec une fonction

Console

```
janson@mp2i:~$ gcc exemple.c -o racines -lm
janson@mp2i:~$ ./racines
Nombre de racines : 2
Racines : 3.000000, -1.000000
```

Exemple

Le programme précédent renvoie ne nombre de solutions réelles d'une équation de la forme $aX^2 + bX + c = 0$, ainsi que les solutions éventuelles.

Pour compiler notre programme avec la librairie **`math.h`**, il faut donner l'option **`-lm`** à **`gcc`** (pour "*librairie math*").

Renvoyer plusieurs valeurs avec une fonction

Utilisation de pointeurs

Une solution plus naturelle en C est d'utiliser des **pointeurs** :

- avant d'appeler la fonction dont on veut récupérer plusieurs valeurs, on crée des variables qui vont acceuillir ces valeurs ;
- on donne ensuite des pointeurs vers ces variables à la fonction en question ;
- la fonction se sert de ces pointeurs pour écrire les valeurs voulues aux bonnes adresses.

L'avantage de cette approche est de ne pas avoir à créer une structure adhoc à chaque fois qu'on voudrait renvoyer plusieurs valeurs avec une fonction.

Renvoyer plusieurs valeurs avec une fonction

exemple.c (1/2)

```c
#include <stdio.h>
#include <math.h>

int racines_reelles(float a, float b, float c, float *r1, float *r2)
{
  float delta = b*b - 4*a*c;
  if (delta == 0)
  {
    float r = -b / (2*a);
    *r1 = r;
    *r2 = r;
    return 1;
  }
  else if (delta > 0)
  {
    *r1 = (-b - sqrt(delta)) / (2*a);
    *r2 = (-b + sqrt(delta)) / (2*a);
    return 2;
  }
  else
  {
    return 0;
  }
}
```

Renvoyer plusieurs valeurs avec une fonction

exemple.c (2/2)

```c
int main()
{
  float r1, r2;
  int nb = racines_reelles(-1, 2, 3, &r1, &r2);
  printf("Nombre de racines : %d\n", nb);
  if (nb > 0)
  {
    printf("Racines : %f, %f\n", r1, r2);
  }
}
```

Console

```
janson@mp2i:~$ gcc exemple.c -o racines -lm
janson@mp2i:~$ ./racines
Nombre de racines : 2
Racines : 3.000000, -1.000000
```

Fonctions avec un nombre variable de paramètres

Fonctions avec un nombre variable de paramètres

Nombre variable de paramètres

Il est possible en C de définir des fonctions qui ont un nombre **variable** de paramètres.

En pratique, il existe souvent des méthodes plus simples pour gérer ce type de problème ; toutefois, cette fonctionnalité est indispensable dans certains cas, notamment pour les fonctions **`printf`** et **`scanf`**.

Remarque

Cette section est hors programme, mais cela peut vous servir dans certains cas, par exemple dans le cadre de votre TIPE.

Fonctions avec un nombre variable de paramètres

Nombre variable de paramètres

Une fonction possédant un nombre variable de paramètre doit posséder au moins un **paramètre formel fixe**.

La notation `...` (obligatoirement à la fin de la liste des paramètres d'une fonction) spécifie que la fonction possède un nombre quelconque de paramètres (éventuellement de types différents) en plus des paramètres formels fixes.

Fonctions avec un nombre variable de paramètres

Exemple

```
1  int f(int a, char c, ...);
```

Exemple

La fonction ayant le prototype ci-dessus prend comme paramètres un entier, un caractère, et un nombre quelconque d'autres paramètres.

Fonctions avec un nombre variable de paramètres

Exemple

```
1  int printf(char *format, ...);
```

Exemple

La fonction **printf** de la librairie standard a le prototype ci-dessus, puisqu'elle a pour argument une chaîne de caractères spécifiant le format des données à afficher, et un nombre quelconque d'autres arguments qui peuvent être de types différents.

Fonctions avec un nombre variable de paramètres

Utilisation

Un **appel** à une fonction ayant un nombre variable de paramètres s'effectue comme un appel à n'importe quelle fonction.

Pour accéder à la liste des paramètres dans le **corps** de la fonction, on utilise les macros définies dans la librairie standard **`stdarg.h`**.

Fonctions avec un nombre variable de paramètres

```
1  va_list liste_parametres;
2  va_start(liste_parametres, dernier_parametre);
3  /* ... */
4  type param = va_arg(liste_parametres, type);
5  /* ... */
6  va_end(liste_parametres);
```

Utilisation

Il faut :

- déclarer une variable de type **va_list** qui va pointer sur la liste des paramètres de l'appel ;
- initialiser sa valeur avec la fonction **va_start** qui prend en paramètres la variable mentionnée ci-dessus, et le nom du **dernier paramètre formel fixe**.

Fonctions avec un nombre variable de paramètres

```
1  va_list liste_parametres;
2  va_start(liste_parametres, dernier_parametre);
3  /* ... */
4  type param = va_arg(liste_parametres, type);
5  /* ... */
6  va_end(liste_parametres);
```

Utilisation

- Ensuite, on accède aux différents paramètres de la liste à l'aide de la fonction **va_arg**, qui prend en arguments la liste des paramètres et le type du prochain paramètre de la liste (à chaque appel de cette fonction, on avance d'un cran dans la liste).
- Enfin, il faudra libérer la variable pointant sur les paramètres avec la fonction **va_end**.

Fonctions avec un nombre variable de paramètres

Remarque

Nous devons gérer nous-mêmes le nombre de paramètres de la liste.

Pour cela, on utilise généralement un **paramètre formel** qui correspond au nombre de paramètres de la liste, ou une valeur particulière qui indique la fin de la liste.

Fonctions avec un nombre variable de paramètres

Exemple

```c
#include <stdlib.h>
#include <stdio.h>
#include <stdarg.h>

int add(int,...);

int main()
{
  printf("%d\n", add(4,10,2,8,5));
  printf("%d\n", add(6,10,15,5,2,8,10));
  return(EXIT_SUCCESS);
}

int add(int nb,...)
{
  int res = 0;
  int i;
  va_list liste_parametres;
  va_start(liste_parametres, nb);
  for (i = 0; i < nb; i++)
    res += va_arg(liste_parametres, int);
  va_end(liste_parametres);
  return(res);
}
```

Output

```
25
30
```