Enseignant
Manuel clergue

3

Introduction au langage C
Les fonctions

2022

esiea

1 Modules, fonctions et procédures

Rappel :

Algorithme
=
expression d'une méthode de résolution par décomposition du traitement
à effectuer en actions élémentaires, en précisant l'ordre de leur exécution.

Analyse descendante : on décompose le problème en actions que l'on décomposera ensuite en sous-actions, puis en actions élémentaires.
Il est intéressant de considérer certaines actions comme des **modules**.

Par définition, un module désigne une entité de données et d'instructions qui fournissent une solution à une partie bien définie d'un problème plus complexe.
N.B. Dans une terminologie plus ancienne, on parlait de sous-programmes.

Avantages des modules (début)

- **Meilleure lisibilité** de la méthode de résolution du problème.
- **Suppression des codes dupliqués** : une même action qui doit être exécutée à plusieurs endroits ne sera écrite qu'une seule fois.
- **Possibilité de tests sélectifs** d'un module, indépendamment du problème complet ⇒ plus facile de cerner puis corriger une erreur.
- **Dissimulation des méthodes** : pour utiliser un module, il suffit de connaître son effet, sans s'occuper des détails de sa réalisation.
- **Réutilisation de modules existants** : il est facile d'utiliser des modules qu'on a écrits soi-même ou qui ont été développés par d'autres personnes (exemple : les bibliothèques ou "libraries").

Exemples de modules "recyclables" :

affichage des valeurs d'un tableau ;

saisie d'une valeur numérique, avec boucle pour interdire des valeurs hors limites.

Avantages des modules (fin)

- **Simplicité de la mise à jour** : un module peut être modifié ou remplacé indépendamment des autres modules du programme.
- **Favorisation du travail en équipe** : un algorithme peut être développé en équipe, en déléguant la conception des modules à différentes personnes. Une fois développés, les modules peuvent constituer une base de travail commune.
- **Hiérarchisation des modules** : un algorithme peut d'abord être décrit globalement au niveau du module principal. Les détails peuvent être reportés à des modules sous-ordonnés qui peuvent eux aussi être subdivisés en sous-modules et ainsi de suite.

Les modules communiquent entre eux, par le biais des appels ; un module peut faire appel à plusieurs sous-modules pour sa résolution.

Deux familles de modules

On peut distinguer deux familles de modules :

- ceux qui **réalisent des actions** : on les appelera des **procédures**
- ceux qui **calculent des résultats** : on les appellera des **fonctions**

N.B. La notion algorithmique de fonction généralise la notion de fonction mathématique (ex. $\cos(\pi)$ fournit le résultat -1), mais le résultat d'une fonction peut être une structure de données plus élaborée [voir suite du cours], pas seulement une valeur numérique.

Déclaration, définition et appel d'une procédure

Une procédure constitue un bloc dans l'algorithme. Elle doit être déclarée, donc recevoir un identificateur (i.e. un nom).
Ce nom est suivi d'une liste de **paramètres** placée entre des parenthèses : ce sont les **arguments** sur lesquels la procédure effectuera ses traitements.

La définition de la procédure est un code qui contient :

- une (éventuelle) liste de variables locales
- une liste d'instructions

L'appel de la procédure apparaît, dans le code qui est extérieur à son bloc, sous la forme d'une **instruction** à part entière.

Déclaration, définition et appel d'une fonction

Au même titre qu'une variable, une fonction est déclarée à l'aide d'un identificateur et **un type** lui est attribué : celui du résultat qu'elle fournira.
Ce nom est suivi d'une liste de **paramètres** placée entre des parenthèses : ce sont les **arguments** sur lesquels la fonction effectuera ses traitements.

La définition de la fonction est un code qui contient :

- une (éventuelle) liste de variables locales
- une liste d'instructions
- une instruction finale **retourne** suivie du résultat qu'elle renvoie

L'appel de la fonction apparaît, dans le code qui lui est extérieur, sous la forme d'une **variable** qui est utilisée dans l'évaluation d'une expression, ou dans un affichage, etc. . . .

Signature d'une procédure ou d'une fonction

On appelle signature d'une procédure ou d'une fonction la liste de ses paramètres d'appel :

leur nombre, leur ordre et le type de chaque paramètre.

La signature doit être la même lors de la déclaration et lors des appels.

En revanche, les **noms** des variables n'ont aucune raison d'être les mêmes lors de la déclaration et lors des appels.

Variables globales vs locales

Une variable globale est déclarée dans le module principal.
Sa **durée de vie** sera toute la durée de l'exécution.

Elle sera **connue** par toutes les instructions du module principal, ainsi que par toutes celles de toutes les procédures et fonctions.

Une variable locale est déclarée à l'intérieur d'une procédure (ou fonction).
Sa **durée de vie** sera limitée à la durée de l'exécution du module (procédure ou fonction), lors d'un appel. Si le module est appelé une seconde fois, ses variables locales sont à nouveau créées.

Les paramètres

Les variables dont les noms figurent dans la liste des paramètres à la déclaration d'une procédure ou d'une fonction existent **uniquement pendant** que ce module est exécuté (donc suite à un appel).

Par défaut, la correspondance entre les paramètres et les noms des variables passées en argument lors d'un appel se fait **par valeur**,
⇒ on peut considérer les paramètres comme des variables locales dont la valeur initiale est celle reçue lors d'un appel.
Par conséquent, les éventuelles modifications effectuées sur ces variables-là, à l'intérieur du code du module, n'ont pas d'effet sur les variables du module appelant.

Passage par adresse

Si l'on souhaite néanmoins qu'un module puisse modifier les valeurs de certaines variables du module qui l'appelle, il faut alors passer ces variables **par adresse**.

Pour cela, il faudra transmettre comme paramètre l'**adresse** d'une variable et non plus sa valeur. On pourra réaliser cela à l'aide de pointeurs [voir suite du cours].

Conseil pratique : **EVITER** les variables **globales** !

En effet :
on risque de perdre le contrôle des valeurs d'une variable si elle peut être modifiée aussi bien par un sous-module (lors d'un ou... plusieurs appels) que par le module principal.

⇒ utiliser au maximum des variables locales et des paramètres.

2 Les fonctions en C

Les “fonctions” en langage C

Le langage C ne fait pas de distinction clairement explicite entre une procédure et une fonction.

- Les deux types de modules se déclarent de la même manière. La seule différence est que le **type** d'une procédure sera **void**, c'est-à-dire vide.
- Au niveau de la définition, une fonction devra nécessairement avoir comme dernière instruction

```
return (variable ou expression);
```

Et, bien sûr, l'appel d'une procédure se comporte comme une instruction, tandis que celui d'une fonction s'utilise comme une variable.

Prototype - Déclaration - Appel

On distingue trois *moments* dans la vie d'une fonction :

- La déclaration du prototype (optionel) : on signale au compilateur l'existence de la fonction en déclarant sa signature (identifiant - valeur de retour - nombre et types des paramètres). Notez bien le ; à la fin.

```
type_retour nom_fonction(type_arg1 nom_arg1, ... , type_argn nom_arg_n);
```

- la définition de la fonction : c'est là qu'elle prend corps. Si un prototype a été déclaré pour une fonction du même nom, il est obligatoire d'avoir la même signature.

```
type_retour nom_fonction(type_arg1 nom_arg1, ... , type_argn nom_arg_n){
        //  le corps de la fonction :
        //  déclaration des variables locales
        //  instructions
}
```

Prototype - Déclaration - Appel

- l'appel (ou les appels) de la fonction : le module en cours d'exécution est interrompu, la fonction est exécutée, puis quand cette dernière est terminée, on retourne au module appelant.

```
1 // si on ne fait rien de la valeur de retour
2 nom_fonction(val_arg1, ...,val_argn);
```

```
1 // la valeur de retour est utilisée dans une expression
2 a = z + nom_fonction(val_arg1, ...,val_argn);
```

Exemple

```c
// Fonction réalisant l'addition de deux entiers
// Paramètres : a, b de type int
// Valeur de retour : somme des deux entiers passés en paramètres
int addition (int a, int b) {
        a = a+b;
        return a;
}

// Programme principal
int main (){
        int x=2, y=3, res;
        printf("Avant x=%d, y=%d.\n", x, y);
        res = addition(x, y); // Appel de la fonction
        printf("x+y = %d", res);
        printf("Après x=%d, y=%d.\n", x, y);
}
```

Instruction return

- L'instruction `return` termine immédiatement la fonction
- Le cas échéant `return` doit être suivi d'une valeur correspondant au type de retour de la fonction
- Une fonction peut contenir plusieurs `return`

```
1  int max (int i, int j){
2          if (i>j)
3                  return i; // Premier return. Atteint si i > j
4          return j; // Second return. Atteint si j >= i
5  }
```

On écrirait la fonction précédente plutôt comme ceci :

```
1 int max (int i, int j){
2                 return (i>=j)i:j;
3 }
```

Points à retenir

- Avant d'utiliser une fonction, il faut qu'elle est été définie, ou bien que son prototype ait été déclaré
- En général, on déclare tous les prototypes en début de fichier (ou dans un fichier d'entête, voir plus loin)
- Il n'est pas autorisé de déclarer une fonction à l'intérieur d'une autre fonction
- Les variables déclarées dans une fonction sont locales à la fonction. Elles ne sont connues que de la fonction
- Les seuls canaux de communication entre la fonction et son contexte appelant sont :
  - En entrée : les paramètres
  - En sortie : la valeur de retour

La vérité sur la fonction *main*

- C'est la fonction qui est appelée par le programme principal
- Elle retourne un entier, qui correspond à un code d'erreur :
  - 0 si tout c'est bien passé
  - autre chose sinon
- Des codes d'erreur normalisés (comme `EXIT_SUCCESS` ou `EXIT_FAIL`) sont définis dans `stdlib.h`
- Son prototype exact est :

```
1  int main (int argc, char ** argv)
```

- BTW, `stdlib.h` défini aussi une fonction `exit(...)`, qui termine l'exécution du programme.

Exemple

```c
int main (int argc, char ** argv){
        if (argc<3){ // Vérification des arguments de la ligne de commande
                printf("Pas assez d'arguments sur la ligne de commande!\n");
                return EXIT_FAILURE;
        }
        printf("Bonjour, %s, vous avez %d ans.\n", argv[1], atoi(argv[2]));
        return EXIT_SUCCESS;
}
```

- Quel est le rôle de la fonction `atoi` ?
- Par quelle autre fonction on pourrait la remplacer ?

3 Bibliothèques (*libraries*

Fichiers d'entête

- Directives préprocesseurs :
  - Pour les fichiers du langage : `#include <header.h>`
  - Pour les fichiers utilisateur : `#include "header.h"`
- Les fichiers header contiennent :
  - Prototypes des fonctions
  - Constantes
  - Les déclarations de types (`typedef`, `struct`, `union`, `enum`, ...)
- Penser aux balises ! Elles empêchent l'inclusion multiple d'un fichier
- Ne pas oublier les ; à la fin des prototypes

Un fichier d'entête (`addition.h`) :

```
1  #ifndef ADDITION_H // balise  : correspond au nom du fichier
2  #define ADDITION_H // balise
3
4  #define N 10 // Constante
5  int addition (int i, int j); // Prototype
6
7  #endif              // balise
```

Nota bene : certains utilisent la directive `#pragma once` comme balise ...

et son utilisation :

```c
#include <stdio.h>
#include "fichier.h"
int addition (int i, int j){
        return i+j;
}

int main (int argc, char ** argv){
        addition(3, N);
        return 0;
}
```

Portée des variables

- Définition : La portée d'une variable indique la zone de code dans laquelle elle est définie.
- Exemples :
  - Variables définies en début de fonction : fonction
  - Variables dans un bloc : bloc
  - Variables globales (définie entre 2 fonctions) : fichier
  - Arguments : fonction
  - Constantes d'un header : tout fichier appelant le header