Méthodologie de conception et de programmation

Cours 4

N. de Rugy-Altherre - Vincent Demange

Fonctions

Définition

Une fonction est composée de deux parties :

- **La signature** : définit le type du résultat, le nom de la fonction, les types et noms des paramètres.
- **Le corps** : contient une (ou plusieurs) instruction(s) **`return`**, des déclarations de variables locales, et les instructions de la fonction.

Fonctions

Exemple

```
int puissance(int n, int p) {
    int i, res = 1;

    for(i = 1; i <= p; i++) {
        res *= n;
    }
    return res;
}
```

Le **prototype** de cette fonction ($\simeq$ déclaration) est :

```
int puissance(int n, int p);
```

Fonctions

Fonctions particulières

- Une **procédure** est une fonction n'ayant pas de valeur de retour :

```
void procedure(int arg1, int arg2);
```

- Une fonction peut n'avoir aucun argument :

```
int fonction();
```

- La fonction `main` : point d'entrée du programme.

Rappel

Une instruction **`return`** provoque la sortie et fin de la fonction.

Fonctions : déclaration

Ordre de déclaration

Les fonctions doivent être déclarées **avant** leur utilisation (p. ex. par la fonction `main`).

Deux solutions :

1. écrire la fonction avant son utilisation ;
2. écrire seulement le prototype de la fonction avant son utilisation, et sa définition après.

Fonctions : déclaration

Exemple

```
int puissance(int n, int p);  // prototype

int main(int argc, char * argv[]) {
    printf("4**3=%d\n", puissance(4,3));
    exit(0);
}

int puissance(int n, int p) { // definition
    int i, res = 1;

    for(i = 1; i <= p; i++) {
        res *= n;
    }
    return res;
}
```

Portée des variables

Variables globales/locales

- **Variable locale** : déclarée dans une fonction, elle est détruite à la fin de cette fonction et ne peut pas être utilisée en dehors.
- **Variable globale** : déclarée en dehors de toute fonction (généralement au début avec les prototypes). Elle peut être utilisée par toutes les fonctions qui suivent.

Portée des variables

Exemple : variables locales

```
int f(int a) {
    int b = a+1;
    return b;
}

int main(int argc, char * argv[]) {
    int c = 0;
    c = f(c);
    printf("%d\n", b); // erreur
}
```

La variable b n'existe pas dans la fonction main, d'où une erreur à la compilation.

Portée des variables

Exemple : variables globales

```
int b;

int f(int a) {
    b = a+1;
    return b;
}

int main(int argc, char * argv[]) {
    int c = 0;
    c = f(c);
    printf("%d\n", b);
}
```

La variable b est globale donc accessible depuis la fonction main.

Passage par valeur/référence

Objectif

Calculer et retourner à la fois le quotient et le reste de la division de a par b :

```
? div(int a, int b) {
    int r = a%b;
    int q = a/b;
    return ?;
}
```

Solutions possibles

1. Retourner un tableau de deux éléments (difficile en C), ou un enregistrement.
2. Utiliser des variables globales (difficile à maintenir).
3. Passer en paramètres les adresses de deux variables à modifier.

Passage par valeur/référence

Exemple de passage par référence

```
void div(int a, int b, int * div, int * res) {
    *res = a%b;
    *div = a/b;
}

int main(int argc, char* argv[]) {
    int r, d;

    div(18, 3, &d, &r);
    printf("%d = %d * %d + %d\n", 18, 3, d, r);
}
```

Fonctions et tableaux

Tableau comme argument d'une fonction

Un tableau étant un pointeur, on peut considérer qu'il est toujours passé par référence à une fonction (i.e. ses éléments sont modifiables).

En pratique

Ne pas oublier de passer à la fonction une variable contenant la taille effective du tableau !

Exemple

```
void parcours(int tab[], int len) {
    int i;
    for(i = 0; i < len; i++){
        printf("%d\n", tab[i]);
    }
}
```

Fonctions et enregistrement

Un enregistrement fonctionne comme une variable standard : il est passé par défaut par valeur à une fonction. La modification d'un enregistrement dans une fonction ne le modifie donc pas hors de la fonction.

Fonctions et enregistrement

Exemple

```
void mystere(eleve_t e1, eleve_t* pE2) {
    e1.note = 1;
    pE2->note = 2;
}
int main(){
    eleve_t el1, el2;
    el1.note = 10;
    el2.note = 13;
    mystere(e1,&e2);
    printf("%i\n",e1->note) //affichera 10
    printf("%i\n",e2->note) //affichera 2
}
```

Fonction `main`

Signatures possibles

- `void main()`
- `int main(int argc, char * argv[])`

Valeur de retour

- nulle : le programme s'est arrêté normalement ;
- strictement positive : le programme a rencontré une erreur ;
- donnée par un appel à `exit(...)` ou `return` dans `main`.

En pratique

Utilisation des constantes `EXIT_SUCCESS` et `EXIT_FAILURE` définies dans `stdlib.h`.

Fonction `main`

Paramètres

- **`int`** `argc` : taille du tableau de chaînes de caractères `argv` ;
- **`char`** `* argv[]` : tableau dont le premier élément est le nom du processus, les suivants les paramètres donnés dans le terminal.

Exemple

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

    printf("Nom du processus: %s\n", argv[0]);
    for (i=1; i < argc; i++) {
        printf("Argument %d: %s\n", i, argv[i]);
    }
}
```

Programmation modulaire

Principes

Découper un programme en *modules*, i.e. ensemble de fonctions, indépendants.

Intérêts

- Améliorations, tests et développements indépendants.
- Réutilisabilité dans plusieurs programmes (bibliothèques).
- Lisibilité, documentation facilitées.

Un module en pratique

- Un fichier **d'en-tête** (*header*) : contient les prototypes des fonctions (p. ex. `fichier.h`).
- Un fichier **source** : contient le corps des fonctions (p. ex. `fichier.c`).

Module : exemple (1/3)

puissance.h

```
#ifndef PUISSANCE_H
#define PUISSANCE_H

// Calcule n a la puissance p
int puissance(int n, int p);

#endif
```

#ifndef PUISSANCE_H teste si PUISSANCE_H est définie, sinon exécute le code jusque **#endif**.

Évite les déclarations multiples de fonctions à la compilation.

Module : exemple (2/3)

puissance.c

```
#include "puissance.h"

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

#include "*puissance.h*" obligatoire : vérifie la cohérence des déclarations et des définitions.

Module : exemple (3/3)

main.c

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

int main(int argc, char* argv[]) {
    printf("%d**%d = %d\n", 24, 11, puissance(24, 11));
    printf("%d**%d = %d\n", 1 , 5 , puissance(1 , 5));
    printf("%d**%d = %d\n", 2 , 20, puissance(2 , 20));

    return EXIT_SUCCESS;
}
```

`main.h` inutile car le code de `main.c` n'a pas vocation à être réutilisé.

Module : compilation

Soit un programme décomposé en modules nommés `module_i.h` et `module_i.c`, et un fichier principal `main.c` contenant **la** fonction `main`.

Étapes

1. Compilation de chaque module :

```
$ gcc -c module_i.c -o module_i.o
```

2. Compilation du programme :

```
$ gcc main.c module_1.o module_2.o ... -o main
```

Exemple

```
$ gcc -c puissance.c -o puissance.o
$ gcc main.c puissance.o -o main
```

Module

Attention

- Le fichier source d'un module, `module.c`, ne doit pas définir de fonction `main` (sinon `module.c` pas réutilisable).
- Si le contenu d'un module change il doit être recompilé, ainsi que tous les modules en dépendant et le programme principal (dans cet ordre).

Exemple

- si `main.c` est modifié :

```
$ gcc main.c puissance.o -o main
```

- si `puissance.h` ou `puissance.c` est modifié :

```
$ gcc -c puissance.c -o puissance.o
$ gcc main.c puissance.o -o main
```

Tester un module

Comment tester les fonctions d'un module en l'absence de `main` ?

Principe

Pour chaque module (`module.c` et `module.h`), on associe un fichier de test `testModule.c` (ou `test_module.c`).

testModule.c

```c
#include "module.h"

// des fonctions utiles aux tests

int main(int argc, char * argv[]) {
    // les tests des fonctions du module
}
```

Ainsi les tests n'interfèrent pas avec le programme principal.

Tester un module : exemple

testPuissance.c

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

void test_puissance_params(int n, int p) {
    printf("%d**%d = %d\n", n, p, puissance(n, p));
}

void test_puissance() {
    test_puissance_params(24, 11);
    test_puissance_params(1 , 5 );
    test_puissance_params(2 , 20);
}

int main(int argc, char* argv[]) {
    test_puissance();

    return EXIT_SUCCESS;
}
```

Résumé : fichiers d'un programme

Fichiers d'un programme

Un programme complet est constitué de :

- modules : `module_1.h`, `module_1.c`, `module_2.h`, `module_2.c`, `...` ;
- un fichier source principal : `main.c` ;
- des tests : `testModule_1.c`, `testModule_2.c`, `...`

Éventuellement l'ensemble des modules est vide ainsi que les tests.

On peut parfois définir des structures ou constantes (préprocesseur) dans des fichiers d'en-têtes (p. ex. `data.h`) sans fichier source correspondant (p. ex. sans `data.c`).

Résumé : compilation d'un programme

Étapes

1. Compilation des modules (option `-c`) :

```
$ gcc -c module1.c -o module1.o
$ gcc -c module2.c -o module2.o
```

2. Compilation des tests :

```
$ gcc testModule1.c module1.o -o testModule1
$ gcc testModule2.c module2.o -o testModule2
```

3. Exécution des tests :

```
$ ./testModule1
$ ./testModule2
```

4. Compilation du programme :

```
$ gcc main.c module1.o module2.o ... -o main
```

Documentation

Présentation

La documentation d'un programme ou d'un module est généralement composé d'un ou plusieurs fichiers `Html` ou `Pdf` contenant des informations sur :

- les fonctions : une description, le prototype, les entrées/sorties, les préconditions à satisfaire, et éventuellement un exemple d'utilisation ;
- les structures de données, les constantes ;
- les paramètres du programme, son utilisation ;
- etc.

Ce type de documentation permet d'utiliser un programme ou une bibliothèque sans avoir à lire le code.

Documentation

En pratique : Doxygen

- C'est un programme qui génère automatiquement une documentation à partir de commentaires spécifiques inclus dans les fichiers sources.
- Il analyse des fichiers sources écrits en C, C++, C#, PHP, Java, Python, ...
- Il produit des documentations en Html, LaTeX, Pdf, XML, ...

Exemple

```
/**
 * Ceci est un commentaire lu par Doxygen
 */

/* Ceci est un commentaire non lu par Doxygen */
// Ceci est un commentaire non lu par Doxygen
```

Doxygen

Documentation obligatoire

Dans ce cours, il faudra documenter avec des commentaires compréhensibles par `Doxygen` :

- les fichiers ;
- les fonctions ;
- les structures.

Doxygen : fichiers

Documentation pour un fichier

Dans un commentaire Doxygen, en début de fichier on indique :

- `\file` : nom du fichier (obligatoire) ;
- `\author` : auteur du fichier ;
- `\brief` : brève description du fichier (obligatoire).

puissance.h

```
/**
 * \file puissance.h
 * \author M. Le Prof
 * \brief Bibliotheque de calcul de puissances.
 */
#ifndef PUISSANCE_H
...
```

Doxygen : fonctions

Documentation pour une fonction

Juste avant le prototype d'une fonction, dans un fichier d'en-tête :

- \fn : prototype exact de la fonction ;
- \brief : brève description de la fonction (obligatoire) ;
- \param : nom et description d'un paramètre (obligatoire) ;
- \return : description de l'élément retourné (obligatoire).

puissance.h

```
...
/**
 * \brief Calcul d'une puissance entiere d'un entier.
 * \param n Base de la puissance.
 * \param p Puissance.
 * \return \a n a la puissance \a p.
 */
int puissance(int n, int p);
```

Doxygen : structures

Documentation pour une structure

- \struct : nom de la structure (obligatoire) ;
- \brief : brève description (obligatoire) ;
- description des champs (voir exemple).

Exemple

```
/**
 * \struct tab_t
 * \brief Tableau de taille variable.
 */
struct tab_t {
    int taille; /*!< Taille effective. */
    int tmax;   /*!< Taille maximale. */
    int * tab;  /*!< Tableau d'entiers. */
};
```

Doxygen : divers

Attention

Pour que les fonctions d'un fichier `fichier` soient lues et les commentaires analysés par Doxygen, il faut que le fichier soit référencé par l'utilisation de `\file fichier`.

Autres marquages

Des marques de mise en forme sont disponibles :

- \a : fait ressortir les variables ;
- \b : mise en gras ;
- \bug : indique un bug ;
- \date : indiquer une date (p. ex. de création) ;
- \e : mise en italique.

Doxygen : génération

Commande

```
$ doxygen fichier
```

où `fichier` est le fichier dont il faut générer la documentation, ainsi que toutes ses dépendances.

Exemple

```
$ doxygen main.c
```

Doxiwizard

Utilitaire graphique de construction de fichier de configuration :

```
$ doxywizard  # construit un fichier Doxyfile
$ doxygen
```