Licence Maths-Info-SPI, informatique pour les scientifiques : cours 1
responsable de l'UE : Jean Lieber, année scolaire 2011-2012

Informations pratiques

Cours de 60H : 40H « cours intégré », 20H de TP. Présence obligatoire aux cours intégrés et aux TP.
5 premières semaines : que des cours intégrés. 10 dernières semaines : cours intégrés et TP.
Modalité de contrôle des connaissances :
– Note de l'examen intermédiaire (coefficient 20%) ;
– Note de l'examen final (coefficient 40%) ;
– Note d'examens courts à l'improviste ou pas en séance de cours intégrés (coefficient 20%) ;
– Note de TP sur la base de rapports et d'examens de TP (coefficient 20%).
Contrôle continu : pas de deuxième session.
Notes de cours : disponibles sur `http://www.loria.fr/~lieber/cours/ips`.
Hypothèse pédagogique : les étudiants travaillent durant les séances et entre les séances. Travail : beaucoup d'exercices (peu d'apprentissage « par cœur »). Acquisition d'un savoir-faire.

1 Introduction

Informatique pour les scientifiques. Savoir construire des programmes simples (ou moins simples) pour effectuer des **calculs**. Autrement dit : savoir « expliquer » à une machine comment faire certains calculs (qu'on doit savoir faire...). Objectif : un programme qui « marche » (et donc **testé**), **lisible**, documenté (avec des **commentaires**) et **réutilisable**. Démarche : partant d'une **spécification**, établir un **algorithme** puis « traduire » en **programme** (démarche affinée dans la suite du cours).
Algorithmique et programmation. Algorithmique : description destinée à un humain d'un processus de calcul (notion ancienne, cf. Euclide). Programme : description destinée à une machine d'un processus de calcul (notion plus récente, cf. les machines à calculer et les ordinateurs).
Langage de programmation utilisé : C. Langage de programmation impératif. Un programme écrit en C est *compilé* (grâce à un compilateur, tel que `gcc`) en code machine. C comme support de cours : ce cours *n'est pas* un cours de C.
Plan du cours.

Chapitre 1 Introduction
Chapitre 2 Premiers algorithmes et premiers programmes
Chapitre 3 Les booléens ($\{\texttt{vrai}, \texttt{faux}\}$)
Chapitre 4 Les entiers ($\mathbb{N}$ et $\mathbb{Z}$)
Chapitre 5 Les flottants ($\mathbb{R}$)
Chapitre 6 Les enregistrements ($A \times B \times C$)
Chapitre 7 Les tableaux ($A \times A \times \ldots \times A = A^n$)
Chapitre 8 Les listes (e.g., $(a\ b\ c\ d\ e)$)
Chapitre 9 Les ensembles finis, piles, files et autres structures linéaires simples
Chapitre 10 Les arbres
Chapitre 11 Conclusion

2 Premiers algorithmes et premiers programmes

Objectif du chapitre : se familiariser avec les notions de spécification, d'algorithme, de programmes C et de tests.
– La **spécification** est le point de départ, l'énoncé de l'exercice de programmation. Elle décrit ce que le programme attendu doit faire (cahier des charges).
– L'**algorithme** décrit un processus de calcul qui doit répondre à la spécification.
– Le **programme** découle d'une traduction de l'algorithme, en l'occurrence, en langage C.
– Les **tests** sont nécessaires pour vérifier que le programme fonctionne sans erreur et donne le résultat spécifié. Dans le cas contraire, il est nécessaire de remonter à l'étape de programmation, voire à l'étape d'algorithmique.

2.1 Premier exemple : « Bonjour tout le monde ! »

Spécification. Écrire un message de bienvenue.
Algorithme. L'algorithme suivant *répond* à cette spécification :

```
début
  afficher "Bonjour tout le monde !"
  retourner à la ligne
fin
```

Programme C. Le programme C traduit l'algorithme précédent :

```
/* Premier programme C */
#include <stdio.h>
#include <stdlib.h> /* pour le EXIT_SUCCESS */
```

```
int main ()
{
  printf ("Bonjour tout le monde !\n") ;
  return EXIT_SUCCESS ;
}
```

Dissection du programme :
- La première ligne est un commentaire : tout texte entre `/*` et `*/` en est un : il n'est pas destiné au compilateur C, mais est utile pour rendre le programme plus facile à comprendre et à réutiliser.
- La deuxième ligne indique l'appel à la bibliothèque standard `stdio.h` (`std`=standard, `io`=input/output). Cela permettra en l'occurrence d'utiliser l'instruction `printf` (voir plus loin).
- La troisième ligne indique l'appel à la bibliothèque standard `stdlib.h`. Cela permettra en l'occurrence de définir la constante `EXIT_SUCCESS` utilisée en fin de programme.
- La quatrième ligne est vide, elle n'est pas utile au programme mais permet d'aérer le code C.
- Ce programme contient une seule fonction : la fonction `main`. Elle est constituée d'une signature et d'un corps :
  - La **signature** est la ligne `int main ()` qui indique le type de sortie (`int`), le nom de la fonction (`main`) et la liste des paramètres (en l'occurrence, la liste vide, `()`).
  - Le **corps** de la fonction est un **bloc**, i.e., une partie du code située entre `{` et `}`.

  Notons que la fonction `main` est toujours celle qui est appelée en premier, à l'exécution du programme (*main* = principale). Par convention, son type de sortie est `int`.
- Le corps de la fonction `main` contient deux instructions
  - `printf ("Bonjour tout le monde !\n") ;`.
    `printf` indique que le contenu de la parenthèse, la chaîne de caractères `"Bonjour tout le monde !\n"`, doit être affiché à l'écran, sachant que `'\n'` est le caractère de retour à la ligne.
    Toute instruction C finit par un point-virgule.
  - `return EXIT_SUCCESS;`
    renvoie un entier (comme demandé dans la signature) qui indique que la fonction s'est terminée avec succès.

Tests. Le programme précédent ayant été édité et sauvé dans le fichier `bienvenue.c`, il peut être compilé, par exemple à partir d'un terminal Linux utilisant la syntaxe suivante :

```
% gcc -o bienvenue bienvenue.c
```

(le `%` est l'invite du terminal). Le résultat est le fichier exécutable `bienvenue` qui donne à l'exécution ceci :

```
% ./bienvenue
Bonjour tout le monde !
%
```

Il s'exécute sans erreur et répond bien à la spécification.

2.2 Deuxième exemple : périmètre d'un cercle

Spécification. Afficher à l'écran une valeur approchée du périmètre d'un cercle de rayon 2 cm.
Algorithmes. On rappelle que le périmètre d'un cercle de rayon r est $2\pi r$. Deux algorithmes sont proposés.

```
 /* Algorithme 1 */
début
  afficher 4 * π
fin
```

```
/* Algorithme 2 */
début
  afficher périmètre_cercle (2.)
fin

fonction périmètre_cercle (rayon : réel) : réel
début
  retourner 2 * π * rayon
fin
```

Exercice 1 *Lequel de ces deux algorithmes est-il préférable ? Pourquoi ?*

Programme C. Le programme C suivant traduit le deuxième algorithme :

```
/* Deuxième programme C    auteur : J. Gustedt et J. Lieber    date : 08/02/11 */
#include <stdio.h>
#include <math.h> // Contient la constante M_PI (nombre pi)
```

```
double perimetre_cercle (double rayon)
{
  return 2.0 * M_PI * rayon ;
}

int main ()
{
  double r ;
  r = 2.0 ;
  printf ("Le périmètre d'un cercle de rayon %f cm est de %f cm.\n",
          r, perimetre_cercle (r)) ;
  return EXIT_SUCCESS ;
}
```

Dissection du programme (seules les nouveautés sont décrites) :
- `#include <math.h>` contient entre autres la constante `M_PI` qui représente une valeur approchée de π.
- La fonction `perimetreCercle` :
  - A un seul paramètre : `rayon`, de type `double` (c'est ce qu'indique l'expression `double rayon`) ;
  - Retourne une valeur de type `double` (cf. le début de la signature, avant le nom de la fonction) ;
  - Le corps de la fonction retourne la valeur résultant du calcul du produit `2. * PI * rayon`.
- La fonction `main` est à la fin du code. Elle fait appel aux autres fonctions et, en C, si la fonction f fait appel à la fonction g, f doit être décrite après g (sauf cas particuliers...). Le corps de cette fonction est constitué :
  - D'une *déclaration de variable* : la variable `r` de type `double`.
  - D'une *affectation de variable* : la variable `r` prend la valeur `2.`
  - D'une instruction d'affichage (`printf`) ; à noter que `%f` fait appel aux valeurs indiquées dans la suite qui doivent être des « réels » (type `double` ou `float`). Pour des entiers, on utilise `%d`. Par exemple `printf ("a = %d et b = %d", 2, 4) ;` équivaut à `printf ("a = 2 et b = 4") ;`.

Tests. Le programme précédent ayant été édité et sauvé dans le fichier `perimetre.c`, il peut être compilé et exécuté :

```
% gcc -o perimetre perimetre.c
% ./perimetre
Le périmètre d'un cercle de rayon 2.000000 cm est de 12.566360 cm.
%
```

Ce qui est conforme à la spécification.

Exercice 2 *Modifier le programme pour l'affichage du périmètre d'un cercle de rayon 50 mètres.*

Exercice 3 *Spécification : afficher les valeurs* 2^2, 2^4, 2^8 *et* 2^{16}.
Écrire l'algorithme, le programme C et les instructions pour tester ce programme.
Comme chaque valeur est le carré de la précédente (la première étant le carré de 2*), on écrira une fonction qui calcule le carré d'un entier donné en paramètre.*

2.3 Troisième exemple : factorielle

Spécification. Écrire une fonction qui calcule la factorielle d'un entier naturel n. Rappel : cette valeur est notée $n!$ et vaut $\prod_{i=1}^{n} i = 1 \times 2 \times \ldots \times n$ pour $n \geqslant 1$ et $0! = 1$ On peut aussi définir cette fonction de façon récursive :

$$0! = 1 \qquad \text{pour tout } n \in \mathbb{N},\ (n+1)! = (n+1) \cdot n!$$

Algorithme. L'algorithme ci-dessous s'appuie sur la formulation récursive (c'est un algorithme récursif) :

```
fonction fact (a : entier_naturel) : entier_naturel
début
  si a = 0
     alors retourner 1
  fsi
  retourner a * fact (a - 1)
fin
```

Programme C.

```c
/* Troisième programme C */
#include <stdio.h>

unsigned int fact (unsigned int n)
{
  if (n == 0)
    return 0 ;  /* (sic) */
  return n * fact (n - 1) ;
}

void test_fact_param (unsigned int a)
{
  printf ("%u! = %u\n", a, fact (a)) ;
}

void test_fact ()
{
  unsigned int i ;
  for (i = 0 ; i <= 5 ; i = i + 1)
    test_fact_param (i) ;
}

int main ()
{
  testFact () ;
  return EXIT_SUCCESS ;
}
```

Dissection du programme (seules les nouveautés sont décrites) :
- La condition du `if` est nécessairement entre parenthèses. Le test d'égalité est noté `==` (le symbole `=` est réservé pour l'affectation de variables).
- Une fois une instruction `return` effectuée, on quitte la fonction. Ainsi, si `n` est nul, la dernière ligne de la fonction `fact` ne sera pas exécutée.
- Le type `unsigned int` est celui des entiers naturels.
- La fonction `testFact` est introduire pour tester la fonction `fact`. Son type de sortie est `void`, autrement dit, il n'y a pas de type de sortie : on parlera d'une **procédure** plutôt que d'une fonction.
- La procédure `testFact` utilise une **boucle for** qui contient une seule instruction, un `printf`. Cela signifie que cette instruction sera exécutée pour les valeurs successives de la variable `i` : `0`, `1`, `2`, `3`, `4` et `5`.

Tests. La compilation et l'exécution du programme donnent :

```
% gcc -o fact fact.c
% ./fact
0! = 0
1! = 0
2! = 0
3! = 0
4! = 0
5! = 0
```

Le programme est manifestement erroné.

Exercice 4 *Corriger le programme (voire l'algorithme) pour qu'il donne le résultat attendu :*

```
% gcc -o fact fact.c
% ./fact
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
```

Exercice 5 *Que faudrait-il changer au programme pour qu'il calcule* $\sum_{i=0}^{n} i = 0 + 1 + \ldots + n$ *?*

Exercice 6 *Donner un autre algorithme et un autre programme pour* `fact` *qui n'utilise pas la récursivité, mais qui utilise la boucle* `pour` *(en algorithmique) et la boucle* `for` *(en C), sachant que la syntaxe de la boucle* `pour` *est (*i *: nom de variable de type entier,* a *et* b *deux entiers tels que* $a \leqslant b$*,* instructions *: une liste d'instructions) :*

```
pour i de a à b faire
  instructions
fin-pour
```

ce qui se traduit en C par

```
for (i = a ; i <= b ; i = i + 1)
  {
    instructions
  }
```
