C’est ma façon de résoudre, et vous ?

Algorithmique et programmation : les bases (C)
Corrigé

Résumé

Ce document décrit l’écriture dans le langage C des éléments vus en algorithmique.

Table des matières

Liste des exercices

1 Pourquoi définir notre langage algorithmique ?

2 Structure d'un algorithme

2.1 Exemple d'algorithme : calculer le périmètre d'un cercle

Un exemple d'algorithme/programme est donné ci-dessous. Il décrit comment obtenir le périmètre d'un cercle à partir de son diamètre. Cet exemple est volontairement très simple.

Listing 1 – Programme C pour calculer le périmètre d'un cercle

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

#define PI 3.1415

int main()
{
    double rayon;           /* le rayon du cercle lu au clavier */
    double perimetre;       /* le perimètre du cercle */
    /* Saisir le rayon */
    printf("Rayon = ");
    scanf("%lf", &rayon);
    /* Calculer le périmètre */
    perimetre = 2 * PI * rayon;                    /* par définition */
    /*{ perimetre == 2 * PI * rayon }*/
    /* Afficher le périmètre */
    printf("Le périmètre est : %4.2f\n", perimetre);

    return EXIT_SUCCESS;
}
```

2.2 Structure de l'algorithme

La structure d'un programme C est proche de celle d'un algorithme. Le fichier, qui doit avoir l'extension .c, commence par un cartouche faisant apparaître le nom des auteurs du programme, la version ou la date de réalisation et l'objectif du programme. Ces éléments sont mis dans des commentaires et sont donc ignorés par le compilateur.

Les #include correspondent à des directives qui indiquent au compilateur (en fait au pré-processeur) d'inclure les fichiers nommés stdio.h et stdlib.h. Ces fichiers font parties de

la bibliothèque standard du C et donne accès à des fonctions déjà définies. Par exemple les fonctions d'affichage (printf) et de lecture (scanf) sont définies dans stdio.h. La constante EXIT_SUCCESS est définie dans stdlib.h. Si on ne met pas ces #include, on ne peut pas utiliser ces fonctions ou constantes.

Le #define suivant permet de définir la constante PI. Il correspond donc à une définition.

Finalement, les déclarations et instructions sont regroupées entre les accolades qui suivent int main(), d'abord les déclarations, puis les instructions. main est la fonction principale, c'est-à-dire que c'est elle qui est exécutée quand le programme sera lancé. Les instructions sont les mêmes que cette présentées dans l'algorithme même si elles ont une forme un peu différente.

2.3 Identificateurs

Un identificateur est un mot de la forme : une lettre (y compris le souligné) suivie d'un nombre quelconque de lettres et de chiffres.

Attention, il n'est pas possible d'utiliser les lettres accentuées en C.

2.4 Commentaires

Les commentaires commencent par /* et se terminent par */. Attention, les commentaires ne peuvent pas être imbriqués.

Pour représenter une propriété du programme, nous utiliserons /*{ ... }*/.

Le langage C++ ajoute les commentaires qui commencent par // et se termine avec la fin de la ligne (comme --). Ils peuvent être utilisés la où on met -- en algorithmique.

3 Variables

3.1 Qu'est ce qu'une variable ?

3.2 Définition d'une variable

En C, on commence par mettre le type suivi du nom de la variable et un point-virgule.

```
double prix_unitaire;        /* prix unitaire d'un article (en euros) */
int quantite;                /* quantité d'articles commandés */
char nom[20];                /* nom de l'article */
```

Les types et leur signification seront présentés dans la suite du cours.

Il est possible de déclarer plusieurs variables du même type en les séparant par des virgules mais ceci est déconseillé sauf si le même commentaire s'applique à toutes les variables.

```
int a, b, c;          /* trois entiers */
```

4 Types fondamentaux

Les opérateurs de comparaison se notent : <, >, <=, >=, == et !=. Notez bien que l'égalité est notée avec deux fois le caractère =.

4.1 Les entiers

Le type entier se note int. Cependant, des qualificatifs peuvent venir préciser :

– sa taille, c'est-à-dire le nombre d'octets sur lequel il est représenté (2 octets pour short, 4 pour long). La taille d'un int est comprise entre celle d'un short int d'un long int. Notons que int est optionnel quand on utilise short et long.

```
short int a;        /* un entier court */
short a;            /* également un entier court (int est implicite) */
long l;             /* un entier long (int est aussi implicite) */
```

– s'ils sont signés ou non. Par défaut, les entiers sont signés (positifs ou négatif). Si l'on précise unsigned devant le type, ils ne peuvent pas être négatifs.

```
unsigned int n;     /* un entier non signé */
unsigned short s;   /* un entier court non signé */
```

Le reste de la division entière se note % et la division entière se note tous simplement /. Il faut faire attention à ne pas la confondre avec la division sur les réels.

```
10 % 3      /* (le reste de la division entière de 10 par 3) */
10 / 3      /* (le quotient de la division entière de 10 par 3) */
1 / 2       /* (le quotient de la division entière de 1 par 2) */
abs(-5)     /* (l'entier est mis entre parenthèses (cf sous-programmes)) */
```

Notons que les débordement de capacité sur les opérations entières ne provoquent aucune erreur à l'exécution... mais le résultat calculé est bien sûr faux par rapport au résultat attendu !.

Remarque : Le type char (caractère, section 4.4) fait partie des entiers.

4.2 Les réels

Il existe deux types réels, les réels simple précision appelés float et les réels double précision appelés double.

La valeur absolue se note fabs. Elle prend en paramètre un double et retourne un double. Pour pouvoir l'utiliser, il faut ajouter en début de fichier #include <math.h>. Dans ce même module sont définies la racine carrée (sqrt), les fonctions trigonométriques (sin, cos, etc.)...

La partir entière d'un réel s'obtient en faisant un cast : (int) 3.14 correspond à 3. Ceci correspond à convertir en entier le réel 3.14.

4.3 Les booléens

Le type booléen n'existe pas. C'est le type entier qui remplace les booléens avec la convention suivante : 0 correspond à FAUX, tous le reste à VRAI. Il faut donc comparer par rapport à 0 et non par rapport à 1 ou un autre entier non nul !

Cependant, il existe un module standard (<stdbool.h>) qui définit un type booléen bool avec les deux valeurs true et false.

Les opérateurs logiques se notent && pour Et, || pour Ou et ! pour Non.

```
1  &&    /* ET logique        expr1 && expr2    */
2  ||    /* OU logique        expr1 || expr2    */
3  !     /* NON logique       ! expr1           */
```

Les expressions booléennes sont évaluées en court-circuit (on parle d'évaluation partielle), c'est-à-dire que dès que le résultat d'une expression est connu, l'évaluation s'arrête. Par exemple, true || expression sera évaluée à true sans calculer la valeur de expression.

4.4 Les caractères

Le type caractère se note char. Les constantes caractères se notent comme en algorithmique. Cependant, le type char est un fait un type entier et sa valeur est le code ASCII du caractère. Il n'y a donc pas de fonctions Chr et Ord.

```
char c;
int i;
c = 'A';          /* la valeur de c est 'A' */
i = c;            /* la valeur de i est 65, code ASCII de 'A' */
```

Enfin, si c est un caractère correspondant à un chiffre (c >= '0'&& c <= '9'), c - '0' est la valeur entière de ce chiffre (entre 0 et 9).

4.5 Les chaînes de caractères

Les constantes « chaînes de caractères » se notent comme en algorithmique.

```
1  "Une chaîne de caractères"
2  "Une chaîne avec guillement (\")"
```

5 Constantes

Les constantes sont définies en utilisant #define :

```
#define   PI 3.1415              /* Valeur de PI */
#define   MAJORITÉ 18            /* Âge correspondant à la majorité */
#define   TVA 19.6               /* Taux de TVA en vigueur au 15/09/2000 (en %) */
#define   CAPACITÉ 120           /* Nombre maximum d'étudiants dans une promotion */
#define   INTITULÉ "Algorithmique et programmation"   /* par exemple */
```

Attention : Ne surtout pas mettre de point-virgule (« ; ») après la déclaration d'une constante avec #define. #define n'est pas traitée par le compilateur mais par le préprocesseur qui fait bêtement du remplacement de texte. Le point-virgule provoquera donc des erreurs là où est utilisée la macro et non où elle est définie !

6 Expressions

La priorité des opérateurs est différente. Voici la table des priorités de C.

```
16G   -> .
15D   (unaires) sizeof ++ -- ~ ! + - * & (cast)
13G   * / %
12G   + -
11G   <<    >>
10G   <    <=    >    >=
 9G   ==    !=
 8G   &
 7G   ^
 6G   |
 5G   &&
 4G   ||
 3G   ?: (si arithmétique)
 2D   = *= /= %=    +=   -=   <<=   >>=   &= |= ^=
 1G   ,
```

```
a + b + c + d          // G : associativité à gauche ((a + b) + c) + d
x = y = z = t          // D : associativité à droite x = (y = (z = t))
```

Le dernier exemple correspond à l'affectation.

Solution : Pour l'exercice Parenthéser, on obtient donc un résultat différent :

```
(2 + (x * 3))                    -- ok si x Entier (ou Réel)
(((- x) + (3 * y)) <= (10 + 3))  -- ok si x et y entiers (ou réels)
((x == 0) Ou (y == 0))           -- ok si x et y entiers (ou réels)
```

7 Instructions d'entrée/sorties

7.1 Opération d'entrée

On utilise scanf qui est une fonction de saisie qui fonctionne avec un format décrivant la nature de l'information à lire et donc la conversion à effectuer.

```
char un_caractere;
int un_entier;
float un_reel;
double un_double;

scanf("%c", &un_caractere);
scanf("%d", &un_entier);
scanf("%f", &un_reel);
scanf("%lf", &un_double);
scanf("%c%d%lf", &un_caractere, &un_entier, &un_double);
```

Chaque % rencontré dans le format (la chaîne de caractères) est suivi d'un caractère indiquant la nature de l'information à lire (c pour caractère, d pour entier, etc.). À chaque % doit

correspondre une variable donnée après le format. Les variables sont séparées par des virgules et sont précédées du signe & (voir sous-programmes). Le & indique que l'on donne l'adresse de la variable de qui permet à scanf de changer la valeur de la variable.

En faisant #include <iostream>, on peut utiliser l'opérateur >> :

```
char un_caractere;
int un_entier;
float un_reel;
double un_double;
std::cin    >>  un_caractere;
std::cin    >>  un_entier;
std::cin    >>  un_reel;
std::cin    >>  un_double;
std::cin    >>  un_caractere >> un_entier >> un_double;
```

Notons que std::cin désigne l'entrée standard, le clavier par défaut.

Il n'y a plus à faire attention ni au format utilisé, ni au nombre de variables fournies.

7.2 Opération de sortie

On utilise printf qui est une fonction de saisie qui, comme scanf, fonctionne avec un format décrivant la nature de l'information à écrire et donc la conversion à effectuer.

```
char un_caractere = 'A';
int un_entier = 10;
float un_reel = 3.14;
double un_double = 1.10e-2;
printf("%c\n",          un_caractere);
printf("%d\n",          un_entier);
printf("%f\n",          un_reel);
printf("%f\n",          un_double);
printf("2 * %d         = %d\n", un_entier, un_entier * 2);
printf("c = %c         et nb = %f\n", un_caractere, un_double);
```

Notons que std::cout désigne la sortie standard, l'écran par défaut. std::cerr désigne la sortie en erreur (également reliée à l'écran par défaut).

Le programme affiche alors :

```
A
10
3.140000
0.011000
2 * 10 = 20
c = A et nb = 0.011000
```

Notons que l'on ne met pas de & devant l'expression.

En C++ : En faisant #include <iostream>, on peut utiliser l'opérateur << :

```
char un_caractere = 'A';
int un_entier = 10;
```

```
float un_reel = 3.14;
double un_double = 1.10e-2;
std::cout    <<    un_caractere << std::endl;
std::cout    <<    un_entier << std::endl;
std::cout    <<    un_reel << std::endl;
std::cout    <<    un_double << std::endl;
std::cout    <<    "2 * " << un_entier << " = " << un_entier * 2 << std::endl ;
std::cout    <<    "c = " << un_caractere
                   << " et nb = " << un_double << std::endl;
```

Le programme affiche alors :

```
A
10
3.14
0.011
2 * 10 = 20
c = A et nb = 0.011
```

Remarque : Malheureusement, le scanf et le printf de C ne sont pas polymorphes et c'est pour cette raison que le programmeur doit préciser le format. Les opérateurs << et >> de C++ sont, quant à eux, polymorphes.

Exercice 1 : Cube d'un réel

Écrire un programme qui affiche le cube d'un nombre réel saisi au clavier.

Solution :

R0 : Afficher le cube d'un nombre réel

Tests :

- 0 -> 0
- 1 -> 1
- 2 -> 8
- -2 -> -8
- 1.1 -> 1,331

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

```
int main()
{
    double x;        /* un nombre saisi par l'utilisateur */
    /* Saisir un nombre réel */
    printf("Nombre = ");
    scanf("%lf", &x);

    /* Afficher le cube de x */
    printf("Son cube est : %f\n", x * x * x);

    return EXIT_SUCCESS;
}
```

8 Affectation

L'affectation se note avec un signe =.

Remarque : L'affectation peut être enchaînée : `a = b = c = 0;` consiste à initialiser c, b puis a avec la valeur 0. C'est équivalent à `a = (b = (c = 0));`.

Attention : Il ne faut pas confondre = (affectation) et == (égalité).

Il exite des formes condensées de l'affectation. Par exemple, `x = x + y` peut se noter `x += y`. Ces formes condensées fonctionnent avec la plupart des opérateurs mais elles sont à éviter dans le cas général car elles peuvent nuire à la lisibilité.

```
x += y  /* équivalent à x = x + y */
x -= y  /* équivalent à x = x - y */
x %= y  /* équivalent à x = x % y */
x |= y  /* équivalent à x = x | y */
...
```

Enfin, il existe les opérateurs de pré-incrémentation et post-incrémentation (idem avec décrémentation).

```
int i = 10;
i++;        /* postincrémentation de i */
++i;        /* préincrémentation de i */
i--;        /* postdécrémentation de i */
--i;        /* prédécrémentation de i */
```

Ces opérateurs peuvent être utilisés dans des instructions (ce qui n'est généralement pas recommandé). On parle de post ou de pré car il sont respectivement exécutés avant et avant l'instruction elle-même.

Ainsi `x = ++y;` est équivalent à :

```
++y;
x = y;
```

Prenons un exemple plus compliqué :

```
int   x, y, z, t,        u;
x =   3;
y =   7;
z =   ++x;               /* x == 4, y == 7, z == 4, t == ?, u == ? */
t =   y--;               /* x == 4, y == 6, z == 4, t == 7, u == ? */
u =   x++ + --y;         /* x == 5, y == 5, z == 4, t == 7, u == 9 */
```

La dernière instruction est équivalente à :

```
--y;   /* décrémentation de y ==> y == 5 */
u = x + y;     /* y = 4 + 5  ==> y == 9 */
x++;   /* incrémentation de x ==> x == 5 */
```

Exercice 2 : Permuter deux caractères

Écrire un programme qui permute la valeur de deux variables c1 et c2 de type caractère.

Solution : Le principe est d'utiliser une variable intermédiaire (tout comme on utilise un récipient intermédiaire si l'on veut échanger le contenu de deux bouteilles).

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

int main()
{
      char c1, c2;              /* les deux caractères à permuter */
      char tmp;                 /* notre intermédiaire */
      /* initialiser c1 et c2 */
      c1 = 'A';
      c2 = 'Z';

      /* permuter c1 et c2 */
      tmp = c1;
      c1 = c2;
      c2 = tmp;

      /* afficher pour vérifier */
      printf("c1 = %c et c2 = %c\n", c1, c2);
      return EXIT_SUCCESS;
}
```

Exercice 3 : Cube d'un réel (avec une variable)

Reprenons l'exercice 1.

3.1 Utiliser une variable intermédiaire pour le résoudre.

Solution : On reprend le même R0 et les mêmes tests. En fait, seule la manière de résoudre le problème change.

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

int main()
{
    double x;                   /* un nombre saisi par l'utilisateur */
    double cube;                /* le cube de x */
    /* Saisir un nombre réel */
    printf("Nombre = ");
    scanf("%lf", &x);

    /* Calculer le cube de x */
    cube = x * x * x;

    /* Afficher le cube */
    printf("Son cube est : %f\n", cube);

    return EXIT_SUCCESS;
}
```

3.2 Quel est l'intérêt d'utiliser une telle variable ?

Solution : L'intérêt d'utiliser une variable intermédiaire est d'améliorer la lisibilité du programme car elle permet de mettre un nom sur une donnée manipulée. Ici on nomme `cube` la donnée `x * x * x`.

De plus, ceci nous a permis, au niveau du raffinage, de découpler le calcul du cube de son affichage. Il est toujours souhaitable de séparer calcul des opérations d'entrées/sorties car l'interface avec l'utilisateur est la partie d'une application qui a le plus de risque d'évoluer.

3.3 Exécuter à la main l'algorithme ainsi écrit.

Solution : À faire soi-même ! On peut également l'exécuter sous le débogueur. La seule différence c'est que le débogueur n'indique pas les variables qui ont une valeur indéterminée.

9 Structures de contrôle

9.1 Enchaînement séquentiel

La séquence s'exprime comme en algorithmique. Pour bien mettre en évidence une séquence d'instructions, on peut la mettre entre accolades. On parle alors de bloc d'instructions. L'intérêt des accolades est alors double :

– il permet de considérer l'ensemble des instructions dans les accolades comme une seule instruction ;
– il permet de déclarer des variables (locales à ce bloc).

9.2 Instructions conditionnelles

9.2.1 Conditionnelle Si ... Alors ... FinSi

```
if (condition)
        une_seule_instruction;

if (condition) {
        instruction1;
        ...
        instructionn;
}
```

Cette deuxième forme est largement préférable car dans la première on ne peut mettre qu'une seule instruction contrôlée par le if alors que dans la seconde, on peut en mettre autant qu'on veut, grâce aux accolades.

Dans la suite, nous utiliserons pour toutes les structures de contrôle la forme avec les accolades mais il existe la forme sans accolades (et donc avec une seule instruction) que nous déconseillons fortement d'utiliser !

Exercice 4 : Une valeur entière est-elle paire ?

Écrire un algorithme qui lit une valeur entière au clavier et affiche « paire » si elle est paire.

Solution :

```
R0 : Afficher « paire » si une valeur entière saisie au clavier est paire

tests     :
        !   -> paire
        ¡   -> -----
        |   -> paire
R1 : Raffinage De « Afficher ... »
    | Saisir la valeur entière n
    | Afficher le verdict de parité

R2 : Raffinage De « Afficher le verdict de parité »
    | Si n est paire Alors
    |     | Écrire("paire")
```

```
    | FinSi

R3 : Raffinage De « n est paire »
    | Résultat <- n Mod 2 = 0
```

Dans le raffinage précédent un point est à noter. Il s'agit du raffinage R2 qui décompose « Afficher le verdict de parité ». Nous n'avons pas directement mis la formule « n Mod 2 = 0 ». L'intérêt est que la formulation « n est paire » est plus facile à comprendre. Avec la formule, il faut d'abord comprendre la formule, puis en déduire sa signification. « n est paire » nous indique ce qui nous intéresse comme information (facile à lire et comprendre) et son raffinage (R3) explique comment on détermine si n est paire. Le lecteur peut alors vérifier la formule en sachant ce qu'elle est sensée représenter.

Raffiner est quelque chose de compliquer car on a souvent tendance à descendre trop vite dans les détails de la solution sans s'arrêter sur les étapes intermédiaires du raffinage alors que ce sont elles qui permettent d'expliquer et de donner du sens à la solution.

Dans cet exercice, vous vous êtes peut-être posé la question : « mais comment sait-on que n est paire ». Si vous avez trouvé la solution vous avez peut-êre donnée directement la formule alors que le point clé est la question. Il faut la conserver dans l'expression de votre algorithme ou programme, donc en faire une étape du raffinage.

Si vous arrivez sur une étape que vous avez du mal à décrire, ce sera toujours une indication d'une étape qui doit apparaître dans le raffinage. Cependant, même pour quelque chose de simple, que vous savez faire directement, il faut être capable de donner les étapes intermédiaires qui conduisent vers et expliquent la solution proposée. Ceci fait partie de l'activité de construction d'un programme ou algorithme.

Remarque : Il est généralement conseillé d'éviter de mélanger traitement et entrées/sorties. C'est pourtant ce qui a été fait ci-dessus. On aurait pu écrire le premier niveau de raffinage différemment en faisant.

```
R1 : Raffinage De « Afficher ... »
    | Saisir la valeur entière                    n: out Entier
    | Déterminer la parité de n                   n: in ; paire: out Booléen
    | Afficher le verdict de parité              paire: in Booléen
R2 : Raffinage De « Déterminer la parité de n »
    | parité <- (n Mod 2) = 0

R2 : Raffinage De « Afficher le verdict de parité »
    | Si paire Alors
    |     | Écrire("paire")
    | FinSi
```

On constate ici que la variable intermédiaire « paire » permet d'avoir un programme plus lisible car on a donné un nom à la quantité (n Mod 2) = 0.

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

int main()
{
    int n;          /* valeur saisie au clavier */
    /* Saisir la valeur entière n */
    printf("Valeur = ");
    scanf("%d", &n);

    /* Afficher le verdict de parité */
    if (n % 2 == 0) {  /*{ n est paire }*/
        printf("paire\n");
    }

    return EXIT_SUCCESS;
}
```

9.2.2 Conditionnelle Si ... Alors ... Sinon ... FinSi

```
if (condition) {                    /* séquence1 */
    instruction1,1;
    ...
}
else { /* ! condition */            /* séquence2 */
    instruction2,1;
    ...
}
```

Exercice 5 : Maximum de deux valeurs réelles

Étant données deux valeurs réelles lues au clavier, afficher à l'écran la plus grande des deux.

Solution :

```
R0 : Afficher le plus grand de deux réels saisis au clavier
tests :
    ! et 2 -> 2
    ! et 1 -> 1
    ! et 3 -> 3
R1 : Raffinage De « Afficher le plus grand de deux réels ... »
  | Saisir les deux réels                x1, x2 : out Réel
  | Déterminer le maximum                x1, x2 : in ; max : out Réel
  | Afficher le maximum
R2 : Raffinage De « Déterminer le maximum »
  | Si x1 > x2 Alors
  |   | max <- x1
```

```
| Sinon
|    | max <- x2
| FinSi
```

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

int main()
{
    double x1, x2;          /* les deux réels saisis au clavier */
    double max;             /* le plus grand de x1 et x2 */
    /* Saisir les deux réels */
    printf("Deux réels : ");
    scanf("%lf%lf", &x1, &x2);

    /* Déterminer le maximum */
    if (x1 > x2) {
        max = x1;
    }
    else {
        max = x2;
    }

    /* Afficher le maximum */
    printf("max(%f, %f) = %f\n", x1, x2, max);

    return EXIT_SUCCESS;
}
```

9.2.3 La clause SinonSi

```
if (condition1) {               /* séquence1 */
    instruction1,1;
    ...
}
else if (condition2) {          /* séquence2 */
    instruction2,1;
    ...
}
...
else if (conditionn,1) { /* séquencen */
    instructionn,1;
    ...
```


Contact : abbassi@info.net
Idée : Maria.