Chapitre 1 : Les sous programmes : fonctions et procédures

1 Introduction

Un programme est un ensemble d'instructions séquentielles pour résoudre un problème spécifique. Afin de trouver la méthode de résolution (algorithme), le problème doit être divisé en différents sous-problèmes dont la solution est moins compliquée. Les problèmes partiels peuvent être résolus à l'aide de sous programmes.

2 Définitions

2.1 Sous-programmes :

Est un ensemble d'instructions indépendantes qui ont un nom et qui est appelé pour l'exécution. L'appelant est soit le programme principal, soit un autre sous-programme. Lorsque le programme, pendant son exécution, atteint l'instruction qui appelle la procédure, le contexte d'exécution devient le contenu du sous-programme, et une fois qu'il a fini l'exécution du sous-programme, il revient à l'exécution de l'instruction qui suit immédiatement l'invocation.

Les sous-programmes sont également appelés : procédures, fonctions, méthodes, routines.

2.1.1 Procédure :

C'est un sous-programme qui ne renvoie aucune valeur dans son nom, mais il peut renvoyer des résultats via des arguments. Le nom de la procédure peut être utilisé comme instruction complète. par exemple:

algorithme	**C**
SomeProc	SomeProc();
OtherProc(x)	OtherProc(x);

2.1.2 Fonction :

Il s'agit d'un sous-programme qui renvoie obligatoirement un résultat dans son nom, car son nom est considéré comme une variable qui porte une certaine valeur. Par conséquent, l'appel de la fonction peut être utilisé comme variable dans les opérations d'affectation et d'autres expressions. Par exemple :

Algorithme	**C**
Y← SomeFunction(X)*5	Y= SomeFunction(X)*5 ;

Remarque : Toute procédure renvoyant un seul résultat en tant qu'argument peut être convertie en fonction.

2.1.3 Avantages de l'utilisation de sous-programmes :

- Lisibilité : l'utilisation de sous-programmes organise le programme et le simplifie, ce qui aide à comprendre le code du programme
- Rapidité dans la programmation : ne pas répéter plusieurs fois la même séquence d'instructions au sein du programme.
- Réduire la taille du programme
- Facilite le processus de maintenance
- Réutilisation : il peut être stocké dans des bibliothèques pour être réutilisé dans d'autres programmes.

2.2 Déclarations

Procédure : la déclaration d'une procédure prend la forme suivante :

algorithme	**C**

procedure nom_proc (liste paramètres)
   variables locales
Debut
   instructions
Fin.
	void nom_proc (liste paramètres){
	   variables locales ;
	   instructions ;
	}

Fonction : la déclaration est similaire à la déclaration d'une procédure, sauf que le type de la valeur de résultat renvoyée doit être spécifié. Elle prend la forme suivante :

Algorithme	**C**

fonction nom_fonc (liste paramètres) : **type** variables locales **Debut** instructions **Fin.**	**type** nom_fonc **(**liste paramètres**){** variables locales ; instructions ; **}**

- nom_proc, nom_fonc: des identificateurs valides.
- Liste des paramètres (facultatif) : un ensemble de variables par lesquelles les données sont transmis et les résultats sont récupérés, séparés par une virgule «,», et qui sont entre deux parenthèses et sont de la forme nomParam : type, tel que (a:entier, b:réel) et sont appelés « paramètres formels ».
  - En C, la liste des arguments prend la forme de type nomParam (int a, float b) Les parenthèses () sont nécessaires même si elles ne contiennent aucun argument.
- Déclarations locales (facultatif) : Une liste de variables locales de la forme : var varLoc : type
- Instructions : un ensemble d'instructions de tout type, qui seront exécutées lorsque le sous-programme sera appelé. Où toutes les variables déclarées dans la liste des paramètres ou dans la déclaration locale, qui sont appelées variables locales, et les variables déclarées dans le programme principal, appelées variables globales, peuvent être utilisées.
- Result_Type : Lorsque le programme est une fonction, le type de valeur que la fonction retournera au programme qui l'a appelée doit être spécifié, et une valeur doit être attribuée au nom de la fonction. Il s'agit généralement de la dernière instruction de la fonction et elle est de la forme nom_fonction← expression où le nom de la fonction agit comme une variable spéciale qui contient la valeur de retour par la fonction.
  - Dans le langage C, on peut se passer du type de résultat si le sous-programme est une procédure, mais certaines versions utilisent le mot **void**, qui signifie que la fonction ne renvoie rien, et le mot **return** est utilisé pour attribuer une valeur au nom de la fonction.

- **return** : l'instruction **return** fait quitter le sous-programme et reprendre le programme qui l'a appelé à l'instruction suivant immédiatement l'invocation. Il peut renvoyer une valeur au programme qui a appelé le sous-programme s'il s'agissait d'une fonction.

Format : `return [ <expression> ] ;`

Exemple :

`return 5*x ;` Si une fonction

`return ;` s'il s'agit d'une procédure (c'est-à-dire une fonction de type void)

Notes importantes :

- Pour trouver les arguments, nous posons la question qu'est-ce que nous donnons au sous-programme en entrée et qu'est-ce qu'il renvoie en résultat.
- La liste des paramètres de la partie définition du sous-programme doit être identique en nombre et en type à celle utilisée dans l'invocation du sous-programme.
- La première ligne d'une déclaration de fonction ou de procédure c'est à dire : type de fonction, nom de la fonction, type, ordre et nombre d'arguments, sauf leurs noms, est appelée entête ou prototype.
- Les arguments ne sont pas regroupés s'ils sont du même type comme dans (x, y:entier), mais on met (x:entier, y:entier) (int x, int y)
- Tout type de retour autre que void indique que le programme est une fonction et non une procédure.
- void main() ou simplement main() est une procédure, tandis que int main() est une fonction, vous devez donc utiliser return.
- scanf() et printf() sont deux fonctions déclarées dans la bibliothèque stdio

2.3 Où déclarer les sous-programmes :

Dans l'algorithme, il se trouve après la déclaration des variables et avant début du programme principal.
Dans un programme C, elle est déclarée avant la fonction main().

Remarque : L'ordre des sous-programmes est important car chaque fonction doit être définie avant de pouvoir être utilisée. Autrement dit, si la fonction f1() appelle la fonction f2(), alors la fonction f2() doit être définie avant la fonction f1().

2.4 L'invocation

Pour appeler et exécuter une procédure, on utilise son nom comme une instruction à part et on affecte des valeurs et/ou des variables aux arguments entre parenthèses, appelés paramètres effectifs. Les parenthèses peuvent être omis en l'absence de tout argument, mais en C, elles sont obligatoires.

La même chose pour l'appel d'une fonction, où son nom est considéré comme une variable qui porte une certaine valeur, donc l'appel de la fonction peut être utilisé comme une variable dans les opérations d'affectation et d'autres expressions.

Les paramètres effectifs doivent correspondre en nombres en type et en ordre avec les paramètres formels

2.5 Exemples

Exemples de procédures

- L'affichage sur l'écran des nombres inférieur à une certaine limite, il prend la limite supérieure et ne renvoie rien
  `procedure afficheNbs(n : entier)`
- Afficher les valeurs du tableau à l'écran prend un tableau et ne renvoie rien
  `procedure afficheTab(t :tableau de réel, n :entier)`
- Résoudre une équation quadratique qui prend trois coefficients et renvoie deux solutions
  `procedure eq2(a : entier, b : entier, c : entier, var x1 : entier, var x2 : entier)`

Exemples de fonctions

- Carré d'un nombre Prend un nombre et renvoie son carré
  `fonction carre(x :réel) : réel`
- L'aire d'un rectangle prend deux nombres et renvoie l'aire
  `fonction surface(long :réel, large :réel) : réel`
- La résolution d'une équation du premier ordre prend deux coefficients et renvoie une solution
  `fonction eq1(a :réel, b :réel) : réel`
- La somme d'un tableau prend un tableau et renvoie la somme
  `fonction som(t :tableau de réel, size :entier) : réel`
- savoir si le nombre est premier ou non
  `fonction estPremier(x : entier) : booléen`

Exemple Algorithme

Algorithme test	Nom du programme
var z : réel	Variable globale
procedure afficheNbrs (n:entire)	Le nom de la procédure qui prend une variable n de type entier comme argument
var i:entier	variable locale
Début	Le début de la procédure
pour i←1 **à** n **faire** Ecrire(i) **finpour**	Les instructions de procédure
Fin procédure	fin de la procédure
fonction sommeNbrs (x:entier, y:entier) : entier	Le nom de la fonction qui prend deux variables entières et renvoie un résultat entier. x et y ne sont pas groupés même s'ils sont du même type.
Début	Le début de la fonction
sommeNbrs ←x+y	Le nom de la fonction agit comme une variable et prend le résultat de la somme
Fin fonction	fin de la fonction

Début	Début du programme principal
afficheNbrs (5)	Appelez la procédure afficheNbrs, où 5 est affecté à n, et la procédure affiche les nombres de 1 à 5
z←sommeNbrs (5, 3)	Appelant sommeNbrs, le programme affecte la valeur 5 à x et la valeur 3 à y, puis calcule la somme et l'affecte à z
Ecrire("la somme est ", z)	Il affiche la somme est 8
Fin.	Fin du programme principal

Exemples C

#include <stdio.h>	tiliser la bibliothèque stdio
float z ;	Variable globale
void afficheNbrs (int n)	Le nom de la procédure qui prend une variable n de type entier comme argument
{	Le début de la procédure
int i ;	variable locale
for (i=1 ; i<=n; i++) printf("%d\t",i);	Les instructions de procédure
}	fin de la procédure
int sommeNbrs (int x, int y)	Le nom de la fonction qui prend deux variables entières et renvoie un résultat entier. x et y ne sont pas groupés même s'ils sont du même type.
{	Le début de la fonction
return x+y ;	Le nom de la fonction agit comme une variable et prend le résultat de la somme
}	fin de la fonction
int main()**{**	Début de la fonction principal
afficheNbrs (5);	Appelez la procédure afficheNbrs, où 5 est affecté à n, et la procédure affiche les nombres de 1 à 5
Z=sommeNbrs (5, 3);	Appelant sommeNbrs, le programme affecte la valeur 5 à x et la valeur 3 à y, puis calcule la somme et l'affecte à z
printf("la somme est %d", z);	Il affiche la somme est 8
return 0 ;**}**	Fin de la fonction principal

3 Variables locales et variables globales

Une **variable globale** (global variable) est une variable déclarée en dehors du corps de tout sous-programme, et donc utilisable n'importe où dans le programme. Puisqu'une variable est globale, il n'est pas nécessaire de la passer comme paramètre pour l'utiliser dans des sous-programmes. Quant à sa durée de vie, c'est-à-dire son existence en mémoire, elle est créée lors du chargement du programme en mémoire, et elle n'est supprimée qu'à la fin de l'exécution du programme.

Une variable locale est une variable qui ne peut être utilisée que dans le sous-programme ou le bloc où elle est définie, la variable est créée lorsque la fonction est appelée et supprimée lorsque son exécution est terminée.

- Il est recommandé d'utiliser des variables locales et des paramètres plutôt que des variables globales pour éviter les erreurs et garantir l'indépendance des fonctions.

Exemple Algorithme:

Algorithme glob_loc	
Var glob, b : entier	variables globales
Procedure tst	

Var b, loc : entier	variables locales
Début	
glob←11	Les variables globales sont accessibles à l'intérieur de la procédure
b←22	La variable b locale masque la variable globale b
loc←33	
Ecrire(“dans tst : glob=”, glob, “b=”, b, “loc=”, loc)	
End	
Début	
glob←1	
b←2	La variable b est une variable globale
Ecrire(“avant tst : glob=”, glob, “b=”, b)	Les variables locales telles que loc ne sont pas accessibles
tst	Appel de la procédure
Ecrire(“après tst : glob=”, glob, “b=”, b)	
end	

Exemple en C

#include <stdio.h>	
int glob, b ;	variables globales
tst(){	
int b, loc ;	variables locales
glob=11;	Les variables globales sont accessibles à l'intérieur de la procédure
b=22;	La variable b locale masque la variable globale b
loc=33;	
printf(“dans tst : glob=%d b=%d loc=%d”, glob, b, loc);	
}	
int main(){	
glob=1;	
b=2;	La variable b est une variable globale
printf(“avant tst : glob=%d b=%d”, glob, b);	
Les variables locales telles que loc ne sont pas accessibles	
tst();	Appel de la procédure
printf(“après tst : glob=%d b=%d”, glob, b);	
return 0 ;}	

L'écran :

avant tst : glob=1 b=2
dans tst : glob=11 b=22 loc=33
après tst : glob=11 b=2

Explication :

avant d'appeler tst	Durant l'appel de tst	après avoir appelé tst
glob: 1 b: 2	glob: 11 b: 2 — tst: b: 22 loc: 33	glob: 11 b: 2

Avant l'appel, il n'y a que deux variables glob et b, mais lorsque la procédure tst est appelée, le processeur réserve deux autres variables, loc et b. La procédure peut accéder aux variables globales, mais la variable

locale b masque la variable globale b, et lorsque la procédure est terminée, le processeur supprime toutes les variables locales.

4 Passage de paramètres

Les arguments sont les variables par lesquelles les informations peuvent être échangées entre les programmes, c'est-à-dire l'introduction de données du programme appelant au sous-programme et/ou la sortie des résultats du sous-programme au programme appelant.

Il existe deux façons de passer des paramètres ou des arguments

Passage par valeur :

Dans ce mode, la valeur de la variable d'origine est copiée dans le paramètre (formel), et cette copie est utilisée (une variable locale), et la variable d'origine n'est pas modifiée. Dans ce mode, une valeur constante ou une expression peut être passé, et il n'est pas nécessaire qu'il s'agisse d'une variable.

Ce mode est utilisé uniquement pour entrer des informations dans le sous-programme et n'est pas utilisé pour recevoir les résultats.

Passage par référence, adresse ou variable :

Non seulement la valeur est passée, mais la place de la variable d'origine (adresse) est passée à la variable formelle, elles deviennent donc une seule variable, et toute modification du paramètre dans le sous-programme appelé entraîne la modification de la variable d'origine qui était passé en paramètre.

Dans ce mode, il n'est pas possible de passer une valeur constante ou une expression, mais il est nécessaire qu'il s'agisse d'une variable, cela s'appelle donc passer par variable.

Ce mode est utilisé pour entrer des informations pour le sous-programme, surtout s'il s'agit de variable de grande taille, pour éviter de le copier, comme des tableaux et des matrices. Il est également utilisé pour recevoir des résultats.

Le mot var est utilisé avant de déclarer le nom de l'argument pour indiquer que le passage est un passe par variable ou passage par référence.

Pour passer les arguments avec l'adresse en C, on utilise les pointeurs que l'on verra dans le troisième chapitre de ce cours, où le nom du paramètre formel est précédé de * lors de la déclaration et lors de l'utilisation, mais lors de l'appel de la fonction, la variable effective est précédée de &

Déclaration int f(int *x)
Utilisation *x=5;
Appel f(&a);

En C++, la gestion des pointeurs est masquée en utilisant le symbole $ lors de la déclaration uniquement, et cela s'appelle une référence

Déclaration int f(int $x)
Utilisation x=5;
Appel f(a);

Remarque : Nous n'utilisons pas le mot **var** (* en C) pour entrer des données et afficher des résultats.

Exemple Algorithme :

passage par valeur	passage par référence, adresse ou variable

Algorithme passage_valeur	Algorithme passage_variable

var a, c: réel
Procedure carre (x: réel, y: réel)
Debut
y← x*x
fin
debut
c←0
a←3
ecrire("avant carre c=", c)
carre(a ,c)
// on peut utiliser carre(3,c)
ecrire("aprés carre c=", c)
fin
	var a, c: réel
	Procedure carre (x: réel, **var** y: réel)
	Début
	y← x*x
	fin
	début
	c←0
	a←3
	ecrire("avant carre c=", c)
	carre(a,c)
	ecrire("aprés carre c=", c)
	fin

l'écran	
avant carre c=0 aprés carre c=0	avant carre c=0 aprés carre c=9

Exemple C :

passage par valeur	passage par référence, adresse ou variable

#include <stdio.h>
void carre (float x, float y){
y= x*x;
}
int main(){
float a, c;
c=0;
a=3;
printf ("avant carre c=%f ", c);
carre(a ,c);
// on peut utiliser carre(a,5)
printf ("aprés carre c=%f ", c);
return 0 ;}
	#include <stdio.h>
	void carre (float x, float ***y) {
	*y=x*x;
	}
	int main(){
	float a, c;
	c←0;
	a←3;
	printf ("avant carre c=%f ", c);
	carre(a,&c);// on ne peut pas utiliser carre(a,5)
	printf ("aprés carre c=%f", c);
	return 0 ;}

l'écran	
avant carre c=0 aprés carre c=0	avant carre c=0 aprés carre c=9

Passer d'une procédure à une fonction :

Toute procédure qui renvoie un seul résultat peut être convertie en une fonction, où nous changeons le mot Procédure en fonction et nous transformons l'argument que la procédure renvoie en une variable locale et définissons le type de la fonction comme étant le type de cet argument et avant de terminer la fonction, nous attribuons la valeur de la variable au nom de la fonction.

Par exemple, le sous-programme qui calcule la valeur absolue d'un nombre réel :

Sous la forme d'une procédure	Sous la forme d'une fonction
Procedure abs (x: réel, **var** y: réel) **Début** si x<0 alors y← -x sinon y← x finsi **fin**	**fonction** abs (x: réel) : réel **var** y: réel **Début** si x<0 alors y← -x sinon y← x finsi abs←y **fin**
appel	
abs(-5, z)	z←abs (-5)

En C

<table>
<tr><td><pre>void abs (float x, float *y){
   if (x<0)
      *y= -x;
   else
      *y= x;
}</pre></td><td><pre>float abs (float x){
   float y;
   if (x<0)
      y= -x;
   else
      y= x;
   return y ;
}</pre></td></tr>
<tr><td>La variable y peut être omise
Comme on peut omettre else qui vient après return.</td><td><pre>float abs (float x){
   if (x<0)
      return -x;
   return x;
}</pre></td></tr>
<tr><td colspan="2">appel</td></tr>
<tr><td>abs(-5, &z);</td><td>z=abs (-5);</td></tr>
</table>

5 La récursivité

La récursivité est un moyen simple et élégant de résoudre certains problèmes de nature récurrente.

Un programme récursif est tout programme qui se rappelle lui-même. Alors que le programme défini est utilisé pour se définir. Concrètement, un programme récursif est un programme qui fait une partie du travail et se rappelle ensuite pour terminer le reste.

Remarque : Toute boucle **pour** ou **tantque** peut être transformé en programme récursif.

Condition d'arrêt

Puisque le programme récursif s'appelle lui-même, il est nécessaire de fournir une condition pour arrêter la récursivité, ce qui est le cas où le programme ne s'appelle pas ou il ne s'arrêtera jamais.

Il est préférable de tester d'abord la condition d'arrêt, puis, si la condition n'est pas remplie, de rappeler le programme au fur et à mesure que l'appel conduit à la condition d'arrêt.

Exemple :

<table>
<tr><td><pre>Procédure affiche (i :entier)
début
   ecrire(i)
   affiche (i +1)
Fin.</pre></td><td><pre>void affiche (int i)
{
   printf("%d",i);
   affiche (i +1);
}</pre></td></tr>
</table>

Par exemple, on invoque affiche (1), donc il affiche 1, puis il invoque affiche pour i=i+1=2, donc il affiche 2, puis à l'infini, donc l'algorithme doit être muni d'une condition d'arrêt, par Exemple :

<table>
<tr><td><pre>Procédure affiche (i :entier)
début
  si (i<10) alors
    ecrire(i)
    affiche (i +1)
  fsi
Fin.</pre></td><td><pre>void affiche (int i)
{
   if (i<10) {
      printf("%d",i);
      affiche (i +1);
   }
}</pre></td></tr>
</table>

La forme générale du programme récursif :

Procédure récursive (paramètres)
début
   si (condition d'arrêt) **alors**
      <instructions du point d'arrêt>
   Sinon
      <instructions>
      Appel récursif (paramètres changés)
      <Instructions>
   Fsi;
Fin.
	void recursive(paramètres) **{**
	   if (condition d'arrêt)
	      <instructions du point d'arrêt>
	   else
	   {
	      <instructions>
	      Appel récursif (paramètres changés)
	      <Instructions>
	   }
	}

Exemple :

1. Factoriel

$$\text{fact}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot \text{fact}(n-1) & \text{if } n > 0 \end{cases}$$

La fonction peut s'écrire sous la forme d'une relation récursive :

$$b_0 = 1$$

$$b_n = nb_{n-1}$$

itératif	récursif

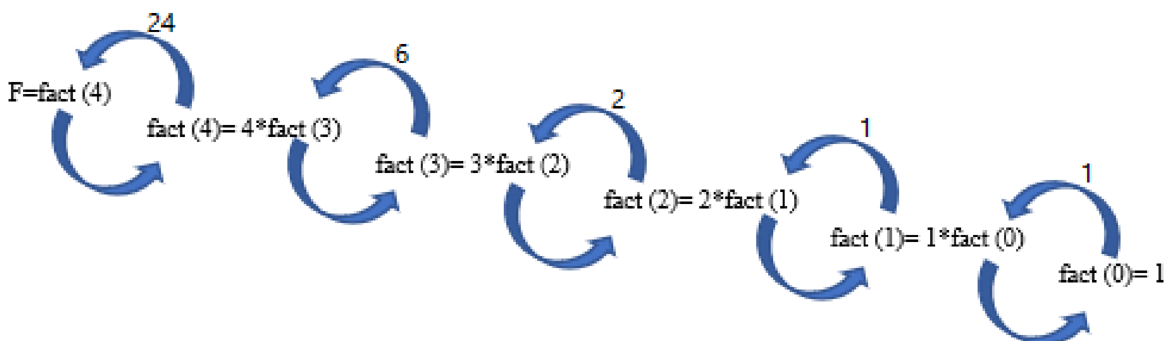
Fonction fact (n : Entier) : Entier
var i, f: Entier
début
   f←1
   pour i←2 **à** n **faire**
      f← f * i
   fpour
   fact←f
fin.
	Fonction fact (n : Entier) : Entier
	début
	   si (n = 0) alors
	      fact←1
	   sinon
	      fact←n*fact (n-1)
	   fsi ;
	fin.

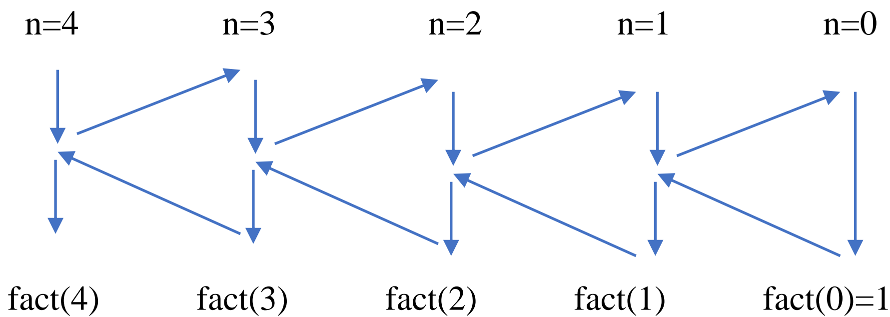
int fact (int n){
   int f=1;
   for (i=2 ;i<= n , i++)
      f← f * i
   return f;
}
	int fact (int n){
	   if (n == 0) **return** 1;
	   return n*fact(n-1);
	}

Comment elle fonctionne ?

On appelle la fonction fact pour n=4 pour calculer 4!

Nous appelons F=fact(4) qui à son tour appelle fact(3) qui appelle fact(2) jusqu'à ce qu'il appelle fact(0) qui se termine et renvoie 1 permettant à fact(1) d'être calculé ce qui permet à fact(2) d'être calculé jusqu'à ce que lfact(4) soit calculé fact(4). Voir ci-dessous.





La pile d'exécution :

Un emplacement en mémoire désigné pour contenir les paramètres et les variables locales et où le résultat est stocké pour chaque sous-programme en cours d'exécution.

Habituellement, la programmation en mode récursif est plus facile et plus lisible, mais elle consomme beaucoup de mémoire, par exemple pour calculer 4! On réserve dans la pile une place pour mettre le résultat, une autre pour mettre le paramètre n=4, puis une autre place pour mettre le résultat de 3! Et le paramètre n = 3 et ainsi de suite jusqu'à ce que 0! soit calculé Le paramètre n=0 est supprimé, puis les paramètres et les résultats sont supprimés dans l'ordre inverse de celui dans lequel ils ont été créés.

Régression mutuelle : un programme récursif peut s'appeler lui-même directement ou indirectement, car il appelle un autre programme, qui à son tour appelle le premier programme.

Exemple :

Pour calculer π, on utilise la relation suivante π/4=1-1/3+1/5-1/7+1/9... On crée deux fonctions régressives, la première additionnant 1/n, appelant la seconde pour n=n-2, puis en soustrayant 1/n qui à son tour appelle le premier à additionner et ainsi de suite jusqu'à ce que n devienne nul.

```
fonction f1(n: entier)
début
   si n<=0 alors
      f1←0
   else
      f1←1/n+f2(n-2)
   fsi
fin

fonction f2(n: entier)
début
   si n<=0 alors
      f2←0
   else
      f2←-1/n+f1(n-2)
   fsi
fin
```

```c
#include <stdio.h>

float f2(int n);

float f1(int n) {
  if (n <= 0) return 0;
  return 1. / n + f2(n - 2);
}

float f2(int n) {
  if (n <= 0) return 0;
  return -1. / n + f1(n - 2);
}

void main() {
  printf("%f\n", 4*f1(2*100+1) *
4);
}
```

La fonction f1 calcule π/4, et pour calculer π, on multiplie le résultat par 4

Note importante : Comme la fonction f1 appelle la fonction f2 qui n'est pas encore définie en langage C, l'en-tête de la fonction f2 doit être ajouté sans son corps (la première ligne) avant de définir la fonction f1, sachant que sa définition vienne après .