Chapitre 1 - Concepts de Programmation

September 2, 2022

1 Chapitre 1: Concepts de Programmation, illustrés en C

1.1 Premiers Pas en C

1.1.1 La fonction main

Un programme C doit toujours contenir une fonction main, c'est le point d'entrée du programme. Voici un programme minimal qui ne fait rien:

[5]:
```c
int main ()
{
}
```

Voici deux exemples plus intéressants: le traditionnel "Hello World" et un programme qui manipule des entiers:

[6]:
```c
#include <stdio.h>

int main ()
{
    printf("Hello World !\n") ;
}

// Cette ligne ne devrait pas exister.
main () ;
```

```
Hello World !
```

[7]:
```c
#include <stdio.h>

int main ()
{
    int n = 14 ;
    printf("%d\n", n + 1) ;
}

// Cette ligne ne devrait pas exister.
main () ;
```

```
15
```

La ligne “#include<stdio.h>” sert à charger la librairie “standard input output” qui permet d’utiliser la fonction printf.

C’est ainsi que l’on communique des valeurs à l’utilisateur; la valeur de retour de la fonction main **n’est pas** retournée à l’utilisateur:

[8]:
```
int main ()
{
    return 3 ;
}

// Cette ligne ne devrait pas exister.
main () ;
```

L’instruction “return” sera utilisé pour les fonctions internes au programme pour communiquer entre elles:

[9]:
```
int f (int x)
{
    return (x+1) ;
}
```

[10]:
```
int main ()
{
    int n = 3 ;
    printf("%d\n", f(n)) ;
}
```

[11]:
```
// Cette ligne ne devrait pas exister
main () ;
```

```
4
```

On peut également communiquer des valeurs au programme (depuis l’utilisateur) avec la fonction scanf:

[12]:
```
int main ()
{
    int n ;
    printf("Quel est votre entier préféré ?\n") ;
    scanf("%d", &n) ;
    printf("Votre entier préféré est %d\n", n) ;
}
```

[13]:
```
// Cette ligne ne devrait pas exister
main () ;
```

```
Quel est votre entier préféré ?
Votre entier préféré est 32765
```

Ici, cela ne fonctionne pas car nous ne sommes pas dans le cadre d'utilisation normal du C. C'est la même chose pour la ligne "main ()" que je dois ajouter à chaque fois pour que la fonction "main" soit appelée: c'est normalement inutile, à l'exécution la première chose que fait le C est d'appeler la fonction "main". Pour pouvoir tester les programmes de ce cours en conditions "normales", vous pouvez utiliser https://www.onlinegdb.com/. Il faudra alors écrire des programmes C complets. Si par exemple vous souhaitez tester une fonction "fctCours" de ce cours, qui prend un seul argument entier, alors il faut écrire le code suivant dans "onlinegdb" (puis tapez un nombre pour tester le programme):

[2]:
```c
// On commence par charger les librairies nécessaires.
// Dans le doute incluez celles ci:
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <stdint.h>
#include <stddef.h>

// Copier ici la fonction du cours
int fctCours (int n)
{
// ...
    return 0 ;
}

// Puis ajoutez la fonction main:
int main ()
{
    int n ;
    printf("Entrez un argument: ") ;
    scanf("%d",&n) ;
    printf("\nfctCours(%d)=%d\n",n,fctCours(n)) ;
}
```

```
input_line_12:10:1: error: function definition is not allowed here
{
^
```

```
Interpreter Error:
```

Si vous avez linux, vous pouvez également vous placer dans les véritables conditions normales d'utilisation (que l'on mettra en place un peu plus tard dans l'année): 1. Ecrivez le code précédent dans un fichier d'extension .c (par exemple monFichier.c); 2. Dans un terminal, depuis le dossier dans lequel se trouve le fichier, tapez la commande: gcc monFichier.c 3. Exécutez votre programme avec la commande: ./a.out 4. Entrez un nombre pour tester votre programme.

1.1.2 Les types de base

Voici les types de base au programme de MP2I: - Les entiers signés: int8_t, int32_t, int64_t. - Les entiers non signés: uint8_t, uint32_t, uint64_t. - En pratique on utilisera: int et long. Le plus souvent, int est un raccourci pour int32_t et long pour int64_t.

[6]:
```
printf("Taille d'un int8_t: %lu\n", sizeof(int8_t)) ;
printf("Taille d'un int32_t: %lu\n", sizeof(int32_t)) ;
printf("Taille d'un int64_t: %lu\n", sizeof(int64_t)) ;
printf("Taille d'un int: %lu\n", sizeof(int)) ;
printf("Taille d'un long: %lu\n", sizeof(long)) ;
```

```
Taille d'un int8_t: 1
Taille d'un int32_t: 4
Taille d'un int64_t: 8
Taille d'un int: 4
Taille d'un long: 8
```

Les entiers intN_t sont compris entre -2^{N-1} et $2^{N-1}-1$.
Les entiers uintN_t sont compris entre 0 et 2^N-1.

[8]:
```
int8_t n = 127 ;
printf("n = %hhd \n", n) ;
printf("n+1 = %hhd \n",n+1) ;
```

```
input_line_15:4:24: warning: format specifies type 'char' but the argument has
type 'int' [-Wformat]
printf("n+1 = %hhd \n",n+1) ;
              ~~~~     ^~~
              %d

n = 127
n+1 = -128
```

[ ]:
```
int n = ((1 << 30) * 2) - 1 ;
printf("%d",n) ;
```

- Les caractères: char (for character in english). On en reparlera au 3e cours.
- Les flottants: float (sur 32 bits) et double (sur 64 bits). On aura un TP dédié à la représentation des flottants et leur utilisation en fin de premier semestre.
- Les booléens: bool. Ils servent majoritairement dans les conditionnelles et les boucles while. Ci dessous un exemple de condition booléenne:

[9]:
```
void printBool (bool b)
{
    printf("%s", b ? "true" : "false") ;
    // Attention: cette syntaxe est hors programme.
}
```

[10]:
```
printBool(true && (false || (1 != 2)) && (1 < 3 || 1 >= 3) && !(false &&␣
↪false));
```

```
true
```

1.2 C (presque) comme du Python

Comme en Python, on a: des variables, des conditionnelles, des boucles FOR et WHILE, on peut définir des fonctions puis les appeler.

IMPORTANT: les codes qui suivent ne sont plus encapsulés dans une fonction "main" pour que ce soit plus lisible. C'est tout à fait impossible de faire cela dans un vrai environnement C ! Pour exécuter les exemples ci dessous dans un environnement C, il faut suivre les instructions en fin de paragraphe 1.1.1.

1.2.1 Les variables

En C, on peut déclarer des variables sans les initialiser, puis les initialiser plus tard.

[13]:
```
int a ;
a = 0 ;
a = 3*(a+1) - 25 ;
printf("La variable a vaut: %d",a) ;
```

```
La variable a vaut: -22
```

On peut également les déclarer et les initialiser en même temps. Le cadre précédent est équivalent à:

[14]:
```
int a = 0;
a = 3*(a+1) - 25 ;
printf("La variable a vaut: %d",a) ;
```

```
La variable a vaut: -22
```

Il faut impérativement préciser le type des variables à la déclaration, et le type ne pourra plus être changé. La principale raison est que le système doit connaitre la taille que prend la variable en mémoire, on verra cela au prochain Chapitre (aspects mémoire).

Une seconde raison en faveur du typage des variables est que cela permet de détecter des erreurs avant l'exécution du programme: si on tente de manipuler un booléen comme si c'était un flottant, on a probablement fait une erreur. Cependant, C n'interdit pas une telle utilisation mais signale (warning) une "conversion implicite":

[16]:
```
double x = 3.5;
printf("%lf", 1/(1+x*x)) ;
```

```
0.075472
```

[17]:
```
bool b = true ;
b = 3.5 ;
```

```
printf("%lf \n",b) ;
printBool(b) ;
```

```
input_line_24:3:5: warning: implicit conversion from 'double' to 'bool' changes
value from 3.5 to true [-Wliteral-conversion]
b = 3.5 ;
  ~ ^~~
input_line_24:4:17: warning: format specifies type 'double' but the argument has
type 'bool' [-Wformat]
printf("%lf \n",b) ;
        ~~~     ^

0.000000
true
```

C'est un point remarquable du C: même s'il détecte une erreur de type (on essaie d'affecter un flottant à une variable booléenne), il produit un avertissement (Warning) et non une erreur. Autrement dit, il sait que vous vous êtes trompés, mais exécute tout de même le code; et bien sûr le résultat n'est pas celui attendu.

Pour signaler à C que ce n'est pas une maladresse de votre part et que vous souhaitez réellement effectuer la **conversion**, on peut utiliser la syntaxe suivante:

```
(new_type)(var)
```

Dans ce cas, nous n'obtenons pas d'avertissement:

[19]:
```
bool t = true ;
bool f = false ;
printf("%lf %lf", (double)(t), (double)(f)) ;
// Ici plus de WARNING.
```

```
1.000000 0.000000
```

[19]:
```
17
```

Les conversions sont à utiliser avec précaution, nous ne l'utiliserons que dans des cas bien précis.

[20]:
```
double x = 3.5 ;
double y = -3.5 ;
int n = 12 ;
printf("Le flottant %lf est converti en entier: %d \n", x, (int)(x)) ;
printf("Le flottant %lf est converti en entier: %d \n", y, (int)(y)) ;
printf("L'entier %d est converti en flottant: %lf \n", n, (double)(n)) ;
printf("Les booleens TRUE et FALSE sont convertis en entier: %d %d \n",␣
 ↪(int)(true), (int)(false) ) ;
```

```
Le flottant 3.500000 est converti en entier: 3
Le flottant -3.500000 est converti en entier: -3
L'entier 12 est converti en flottant: 12.000000
Les booleens TRUE et FALSE sont convertis en entier: 1 0
```

Les conversions seront cependant bien pratiques dans certains cas:

[24]:
```
for (int i = 0; i < 26; i++)
    printf("%c ",'a'+i) ;
printf("\nNow I know my ABC, next time will you sing with me ?") ;
```

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
Now I know my ABC, next time will you sing with me ?
```

1.2.2 Les conditionnelles et définitions de fonctions

[25]:
```
int min (int x, int y)
{
  if (x < y)
  {
    return x ;
  }
  else
  {
    return y ;
  }
}
```

[26]:
```
printf("min(%d,%d) = %d\n", 12,42,min(12,42)) ;
```

```
min(12,42) = 12
```

ATTENTION Même s'il est important de continuer à indenter votre code comme vous l'avez appris en Python, ce n'est pas du tout obligatoire en C. La première chose que le compilateur (programme qui lit votre programme et "prépare" son exécution) fait est d'enlever les retours à la ligne, les tabulations et les espaces inutiles.

Le code suivant est donc équivalent au précédent:

[27]:
```
int minMoche(int x,int y) {if(x < y){ return x ;}else{return y ;}}printf("%d",␣
↪min(12,42)) ;
```

```
12
```

Vous comprenez donc pourquoi les accolades (absentes de la syntaxe Python) sont ici fondamentales. Par exemple Le compilateur ne fait aucun différence entre les deux codes suivants:

[ ]:
```
int n = 12 ;
int m = 42 ;
if (m < n)
    n = 23 ;
    printf("%d",n) ;
```

[ ]:
```
int n = 12 ;
int m = 42 ;
```

```
if (m < n)
    n = 23 ;
printf("%d",n) ;
```

Lequel est correctement indenté ?

C'est le second: en effet, les accolades d'un bloc "if then else" (ou d'une boucle) sont optionnelles s'il n'y a qu'une seule instruction. On a le droit d'écrire la fonction "min" ainsi:

[ ]:
```
int min (int x, int y)
{
  if (x < y)
    return x ;
  else
    return y ;
}
```

Méfiez vous c'est une pratique dangereuse: il est fréquent de se rendre compte par la suite que l'on avait besoin d'une seconde instruction dans un des blocs et d'oublier d'ajouter les accolades (maintenant obligatoires !).

[2]:
```
int min (int x, int y)
{
  if (x < y)
  {
    printf("Le premier argument est plus petit\n") ;
    return x ;
  }
  else
    return y ;
}
```

1.2.3 Les boucles

[ ]:
```
int syracuse (int n)
{
    while (n != 1)
        if (n % 2 == 0)
            n = n/2 ;
        else
            n = 3*n+1 ;
    return n ;
}
```

[ ]:
```
printf("%d",syracuse(24));
```

Si l'on ajoute une instruction dans la boucle, les accolades ne sont plus optionnelles:

[ ]:
```
int syracuse (int n)
{
    int cpt = 0 ;
    while (n != 1)
    {
        if (n % 2 == 0)
            n = n/2 ;
        else
            n = 3*n+1 ;
        cpt = cpt + 1 ;
    }
    return cpt ;
}
```

[ ]:
```
printf("%d",syracuse(24)) ;
```

Et si on est prudent, on écrit toutes les accolades:

[ ]:
```
int syracuse (int n)
{
    int cpt = 0 ;
    while (n != 1)
    {
        if (n % 2 == 0)
        {
            n = n/2 ;
        }
        else
        {
            n = 3*n+1 ;
        }
        cpt = cpt + 1 ;
    }
    return cpt ;
}
```

Pour les boucles FOR:

[27]:
```
double exp (double x, int n)
{
  double result = 1 ;
  for (int i = 0; i < n; i++)
    result = result * x ;
  return result ;
}
```

[ ]:
```
printf("%lf",exp(3.5,5)) ;
```

En C, les boucles FOR attendent 3 paramètres: le premier est une instruction exécutée en début de boucle, le second est une condition d'arrêt de la boucle testée avant chaque itération (comme pour les boucles WHILE !) et le troisième est l'instruction exécutée après chaque itération de la boucle.

Le plus souvent, on utilisera des boucles FOR ayant la forme ci dessus: on y déclare une variable de boucle i, on itère tant qu'elle est inférieure à un entier n (le nombre d'itérations désirées) et on incrémente i à chaque itération. On obtient le même comportement qu'en Python.

Mais ce format de boucle FOR permet en fait des choses plus complexes. C'est un avantage, ou un danger:

[28]:
```
double exp (double x, int n)
{
  double result = 1 ;
  for (int i = 0; i < n; i++)
  {
    result = result * x ;
    n++ ;
  }
  return result ;
}
```

[29]:
```
printf("%lf",exp(3.5,5)) ;
```

```
inf
```

Observez le comportement du programme suivant et comparez avec le programme équivalent Python (ou autre langage):

[30]:
```
int n = 12 ;
for (int i = 0; i < n; i++)
{
    printf("%d\n",i) ;
    n = 3 ;
}

/*
n = 12
for i in range(0,12):
    print(i)
    n = 3
*/
```

```
0
1
2
```

Gardez en tête que l'instruction du milieu sera exécutée **à chaque itération** !

Evitez donc d'écrire:

[31]:
```
int fctTresCouteuse(int n)
{
    if (n==0)
        return 1 ;
    else
        return (fctTresCouteuse(n-1)*n) ;
}
```

[ ]:
```
int n = 5 ;
int sum = 0 ;
for (int i = 0; i < fctTresCouteuse(n); i++)
    sum = sum + i ;
printf("%d",sum) ;
```

Et préférez:

[ ]:
```
int n = 5 ;
int sum = 0 ;
int max = fctTresCouteuse(n) ;
for (int i = 0; i < max; i++)
    sum = sum + i ;
printf("%d",sum) ;
```

Un avantage du comportement atypique de la boucle FOR en C : on peut l'interrompre avant la fin.

[ ]:
```
// Teste si syracuse(i) < ceil pour tout i <= N.
bool test (int ceil, int N)
{
    bool b = true ;
    for (int i = 0; i <= N; i++)
    {
        if (syracuse(i) >= ceil)
        {
            b = false ;
            i = N + 1 ;
        }
    }
    return b ;
}
```

Exercice (TD): simplifier le code cette fonction (sans modifier son comportement).

1.3 Exécution ligne à ligne de programmes

Un bel outil de visualisation de l'exécution ligne à ligne d'un programme:
https://pythontutor.com/visualize.html#mode=edit

[ ]:
```
int syracuse (int n)
{
    int cpt = 0 ;
    while (n != 1)
    {
        if (n % 2 == 0)
            n = n/2 ;
        else
            n = 3*n+1 ;
        cpt = cpt + 1 ;
    }
    return cpt ;
}
```

[ ]:
```
int mysteryFunction (int n)
{
    int result ;
    int mysteryVar = -1 ;
    int cpt ;
    for (int i = 1; i <= n; i++)
    {
        cpt = syracuse(n);
        if (cpt > mysteryVar)
        {
            result = i ;
            mysteryVar = cpt ;
        }
    }
    return result ;
}
```

Au tableau: on exécute “mysteryFunction(5)” ligne à ligne.

Exercice (TD): exécuter “min(12,42)”, “max(12,42)” et “sum(4)” ligne à ligne.

[ ]:
```
int min (int x, int y)
{
    if (x < y)
        return y ;
    else
        return x ;
}
```

[ ]:
```
int max (int y, int x)
{
    if (x > y)
        return y ;
    else
```

```
        return x ;
}
```

[ ]:
```
int sum (int i)
{
    int result = 0 ;
    for (int n = 0; n <= i; n++)
    {
        result = result + n ;
    }
    return result ;
}
```

Exercice (TD): exécuter la fonction suivante sur quelques valeurs. Que calcule cette fonction ?

Aide: n % 2 vaut 0 lorsque n est pair, 1 sinon.

[ ]:
```
int mysteryFct (int n)
{
    if (n == 0 )
        return 1 ;
    else if (n % 2 == 0)
        return (mysteryFct(n/2) * mysteryFct(n/2)) ;
    else
        return (2*mysteryFct((n-1)/2) * mysteryFct((n-1)/2)) ;
}
```

Exercice (TD): écrivez le programme ci dessous en ajoutant les accolades optionnelles. Que retourne ce programme sur les entrées (0,0), (1,0), (0,1), (1,1) ?

[ ]:
```
void danglingElse (int a, int b)
{
    if (a == 1)
        if (b == 1)
            printf("Passe dans le IF \n") ;
    else
        printf("Passe dans le ELSE \n") ;
}
```

[ ]:
```
danglingElse(0,0) ;
danglingElse(1,0) ;
danglingElse(0,1) ;
danglingElse(1,1) ;
```

1.4 Portée des variables

Exercice: exécuter le code suivant:

[ ]:
```
for (int i = 0; i < 12; i++)
{
  int myVar = i ;
}
printf("%d\n", myVar);
```

Cela génère une erreur: myVar n'existe pas ! Créons la:

[ ]:
```
int myVar = -1 ;
for (int i = 0; i < 12; i++)
{
  int myVar = i ;
}
printf("%d\n", myVar);
```

Bien qu'ayant le même nom, les variables créées dans la boucle ne sont pas les mêmes ! On verra l'explication au prochain cours.

Ce problème n'existe pas en Python puisque le langage ne permet pas de faire la différence entre définir une nouvelle variable ("int myVar = i") et mettre à jour une variable existante ("myVar = i"): ces deux instructions s'écrivent "myVar = i" en Python.

Notez bien qu'il est inélégant et rarement justifié de déclarer une variable à l'intérieur d'une boucle, les erreurs des deux cadres précédents ne devraient jamais vous arriver en pratique.

C'est la même chose pour les appels de fonctions, et là ça peut vous arriver:

[36]:
```
void f (int x)
{
  int yy = x+1 ;
}
```

[38]:
```
int main ()
{
  int a = 3 ;
  f(a) ;
  printf("%d",yy) ;
}
```

```
input_line_49:5:15: error: use of undeclared identifier 'yy'
  printf("%d",yy) ;
              ^
```

```
Interpreter Error:
```

Dans ce cas, "y" n'existe pas: c'est une variable **locale** de la fonction "f", elle n'est pas accessible depuis la fonction "main". Et si on définit une variable "y" **locale** dans la fonction "main", alors elle sera **locale**: ce ne sont pas les mêmes variables "y".

[39]:
```
int main ()
{
  int a = 3 ;
  int yy = 1;
  f(a) ;
  printf("%d",yy) ;
}
```

[40]:
```
main () ;
```

```
1
```

Et sur ce point, le comportement est identique en Python (et dans la plupart des langages de programmation).

1.5 Graphe de flot de contrôle d'un programme

Lorsque l'on exécute un programme sans machine, on écrit entre autre une suite de numéro de lignes du programme. Par exemple l'exécution de "syracuse(5)" donne la suite:

`L1,L3, L4,L6,L9,L10, L4,L6,L7,L10, L4,L6,L7,L10, L4,L6,L7,L10, L4,L6,L7,L10, L4,L12.`

[ ]:
```
int syracuse (int n)
{
    int cpt = 0 ;
    while (n != 1)
    {
        if (n % 2 == 0)
            n = n/2 ;
        else
            n = 3*n+1 ;
    cpt = cpt + 1 ;
    }
    return(cpt) ;
}
```

Mais "syracuse(2)" donne la suite:

`L1,L3, L4,L6,L7,L10, L4,L12.`

Cette suite est bien différente de la précédente. Pour autant, toutes les suites ne sont pas envisageables: il n'existe aucune valeur du paramètre d'entrée n telle que "syracuse(n)" ne donne la suite:

`L1,L4,L10,L12.`

En effet, une exécution de "syracuse(n)" commencera toujours (peu importe la valeur de n) par

`L1,L3`

et finira toujours par.

`L4,L12.`

On définit donc le **graphe de flot de contrôle** d'un programme. C'est une structure qui va nous permettre de visualiser simplement et efficacement toutes les exécutions possibles du programme.

Exercice 5.1 (TD): dessiner le graphe de flot de contrôle de plusieurs des programmes vus plus haut.

Vocabulaire: - *Sommet* du graphe de flot de contrôle: ce sont les cercles contenant des numéros de ligne du programme. - *Arc* du graphe de flot de contrôle: ce sont les flèches. - *Chemin* du graphe de flot de contrôle: ce sont les suites de sommets que l'on peut obtenir en parcourant le graphe depuis l'entrée jusqu'à la sortie, c'est à dire que deux sommets consécutifs du chemin doivent être reliés par un arc, le premier sommet du chemin correspond à L1 et le dernier sommet du chemin correspond à la dernière ligne du programme. - A toute exécution du programme, on peut associer un chemin dans le graphe de flot de contrôle. - A l'inverse, un chemin est dit *faisable* s'il correspond à une exécution du programme. C'est à dire qu'il existe une valeur du (ou des) paramètre(s) pour lequel l'exécution produit exactement ce chemin.

Quelques exemples de chemins non faisables:

[ ]:
```
int stupid (int n)
{
    int cpt = 0 ;
    while (false)
        cpt = cpt + 1 ;
    return cpt ;
}
```

[ ]:
```
int even ()
{
    int cpt = 1 ;
    while ((cpt % 2) != 0)
    {
        cpt = cpt + rand() ;
        printf("%d\n",cpt) ;
    }
    if (cpt % 2 == 1)
        cpt = cpt - 1 ;
    else
        cpt = cpt - 2 ;
    return cpt ;

}
```

[ ]:
```
printf("%d",even()) ;
```

[32]:
```
int notTrivial (int n)
{
    int a = 0 ;
    int s = 1 ;
```

```
    int t = 1 ;
    while (s <= n)
    {
        a++ ;
        s = s + t + 2 ;
        t = t + 2 ;
    }
    int inv1 = t/2+1 ;
    int inv2 = t+1 ;
    int inv3 = inv2/2 ;
    // printf("Ou est le furet ?") ;
    if (s != inv1 * inv1)
        printf("\nIl est passe par ici \n") ;
    if (s != inv2 * inv2 /4)
        printf("\nIl repassera par la \n") ;
    if (s != inv3 * inv3)
        printf("\nEst-il ici aussi ? \n") ;
    return a ;
}
```

[34]:
```
int r = 1000000 ;
for (int  i = 0; i < r; i++)
    notTrivial(i) ;
printf("Fini.\n") ;
```

```
Fini.
```

Ce dernier exemple montre que savoir si un chemin est faisable ou non peut être difficile: cela peut demander une analyse mathématique conséquente du programme. Nous reviendrons sur ce point dans le second chapitre: Analyse de Programmes.

Ci dessous, encore un exemple de chemin non faisable, de nature différente:

[ ]:
```
void triangle (int a, int b, int c)
{
    if (a + b < c || a + c < b || b + c < a)
        printf("Impossible: un triangle verifie l'inegalite triangulaire. \n") ;
    else
    {
        int nbEgalite = 0 ;
        if (a == b)
            nbEgalite++ ;
        if (a == c)
            nbEgalite++ ;
        if (b == c)
            nbEgalite++ ;
        if (nbEgalite == 0)
            printf("C'est un triangle scalene. \n") ;
```

```
        else if (nbEgalite == 1)
            printf("C'est un triangle isocele. \n") ;
        else
            printf("C'est un triangle equilateral. \n") ;
    }
}
```

Exercice 5.2: proposez des valeurs d'entrée (a,b,c) telles que chaque ligne du programme soit exécutée dans l'une au moins des exécutions.

Pour les programmes "stupid", "even" et "notTrivial"; certaines lignes du programme ne sont **jamais** exécutée: peu importe l'entrée du programme, l'exécution ne passera pas par ces lignes. On dit alors que le sommet du graphe de flot de contrôle correspondant à une telle ligne est **non couvrable** (et couvrable dans le cas contraire). Dans tous les exemples précédents, les chemins étaient infaisables car il contenaient un sommet non couvrable. Mais dans le programme "triangle", vous venez de montrer que tous les sommets étaient couvrables. Pourtant il existe bien des chemins infaisable.

Exercice 5.3: donner un chemin infaisable du graphe de flot de contrôle du programme "triangle".

Vocabulaire: - Un sommet du graphe de flot de contrôle est *couvrable* s'il existe un chemin faisable passant par ce sommet. - Un arc du graphe de flot de contrôle est *couvrable* s'il existe un chemin faisable passant par cet arc.

Exercice 5.4 (TD): donner les sommets non couvrables et les arcs non couvrables des différents graphes de flot de contrôle précédents.

Propriété: s'il existe des sommets non couvrables ou des arcs non couvrables, alors on peut simplifier le programme (sans changer son comportement).

1.6 Tester un programme (TD)

Beaucoup des programmes écrit dans ce cours sont un peu artificiels: ils ont vocation à montrer un comportement spécifique du langage C, mais leur résultat est inutile. En pratique (dans le monde réel), on écrit un programme pour répondre à un problème bien précis, et il convient donc de s'assurer que le programme écrit répond effectivement au problème initial. Par exemple, si l'on doit écrire un programme qui trie une liste, on va s'assurer que le programme fonctionne en l'essayant sur quelques listes. Mais comment choisir les listes sur lesquelles tester son programme ?

Voici quelques principes de choix des entrées tests: 1. Choisir des entrées telles que tout sommet couvrable du graphe de flot de contrôle apparaisse dans au moins une exécution des tests. 2. Choisir des entrées telles que tout arc couvrable du graphe de flot de contrôle apparaisse dans au moins une exécution des tests. 3. Choisir des entrées "limites": la liste vide si l'entrée du programme est une liste, 0 si l'entrée du programme est un nombre entier positif, etc. 4. Choisir des entrées qui valident ou invalident les tests booléens du programme de toute les façons possibles. 5. *Programmation défensive*: choisir des entrées qui ne correspondent pas au format attendu. Exemple: -1 si le programme est conçu pour fonctionner sur des entiers positifs.

Pour illustrer la méthode, nous allons produire un *jeu de tests* pour le programme suivant:

[41]:
```
void triangle (int a, int b, int c)
{
    assert(a >= 0 && b >= 0 && c >= 0) ;
    if (a + b < c || a + c < b || b + c < a)
        printf("Impossible: un triangle verifie l'inegalite triangulaire. \n") ;
    else if (a == 0 && b == 0 && c == 0)
        printf("C'est un point.\n") ;
    else
    {
        int nbEgalite = 0 ;
        if (a == b)
            nbEgalite++ ;
        if (a == c)
            nbEgalite++ ;
        if (b == c)
            nbEgalite++ ;
        if (nbEgalite == 0)
        {
            if (a + b == c || a + c == b || b + c == a)
                printf("C'est un triangle plat.\n") ;
            else
                printf("C'est un triangle scalene. \n") ;
        }
        else if (nbEgalite == 1)
        {
            if (a + b == c || a + c == b || b + c == a)
                printf("C'est un triangle plat isocele.\n") ;
            else
                printf("C'est un triangle isocele. \n") ;
        }
        else
            printf("C'est un triangle equilateral. \n") ;
    }
}
```

1.7 Programmation Défensive: utilisation de ASSERT

1.7.1 Pour les fonctions internes au programme

On illustre le concept sur le programme suivant:

[35]:
```
#include <stdio.h>
#include <assert.h>
#include <math.h>
#include <stdlib.h>

// Mauvaise pratique:
```

```
int intSqrt (double x)
    // x est positif.
{
    return ((int)(sqrt(x))) ;
}
```

[36]:
```
printf("La racine carree entiere de %lf est: %d\n",37.56,intSqrt(37.56)) ;
printf("La racine carree entiere de %lf est: %d\n",-1.0, intSqrt(-1)) ;
printf("\n\nExplication: \n") ;
printf("La racine carree de %lf est: %lf\n",-1.0, sqrt(-1)) ;
printf("La conversion en entier de -nan donne: %d\n",(int)(sqrt(-1))) ;
```

```
La racine carree entiere de 37.560000 est: 6
La racine carree entiere de -1.000000 est: -2147483648


Explication:
La racine carree de -1.000000 est: -nan
La conversion en entier de -nan donne: -2147483648
```

Problème: parfois, l'argument ne respecte pas le format attendu par la fonction mais l'exécution ne produit pas d'erreur. Pourtant, le résultat est inutilisable:

[84]:
```
double x = 37.56 ;
printf("Quand on remet au carre:\n") ;
printf("%d < %lf\n", intSqrt(x)*intSqrt(x),x) ;
x = -15000 ;
printf("%d < %lf\n", intSqrt(x)*intSqrt(x),x) ;
```

```
Quand on remet au carre:
36 < 37.560000
0 < -15000.000000
```

Normalement, si votre code est bien conçu, la fonction *interne* intSqrt est toujours appelée (par les autres fonctions de votre programme) avec des arguments positifs. Néanmoins, il est difficile de s'en assurer simplement en parcourant le code. Par sécurité, on encourage donc à vérifier que les arguments de la fonctions satisfont le format d'entrée attendu par la fonction:

[2]:
```
// Bonne pratique:
#include <assert.h>

int intSqrt (double x)
{
    assert (x >= 0) ;
    return ((int)(sqrt(x))) ;
}
```

[4]:
```
// Fonctionnement (à tester dans un véritable environnement):
printf("La racine carree entiere de %lf est: %d\n",37.56,intSqrt(37.56)) ;
printf("(ASSERT ne fonctionne pas ici) La racine carree entiere de %lf est:␣
 ↪%d\n",-1.0, intSqrt(-1)) ;
```

```
La racine carree entiere de 37.560000 est: 6
(ASSERT ne fonctionne pas ici) La racine carree entiere de -1.000000 est:
-2147483648
```

Remarques: - L'utilisation d'assert améliore la lisibilité du code: le mot clé "assert" informe immédiatement un autre développeur des conditions d'utilisation de la fonction. - Très bon outil de debuggage, affiche notamment la ligne de l'erreur. - Pour des raisons de performances, on peut désactiver l'exécution des lignes ASSERT.

1.7.2 Pour les saisies de l'utilisateur

Si la fonction intSqrt n'est utilisée que par d'autres fonctions de votre programme, alors vous "pouvez" contrôler que les arguments reçus ont le bon format. Cependant, si l'un des argument provient de l'utilisateur, il est impossible de certifier qu'il aura le bon format. Même si le format attendu est spécifié à l'utilisateur, on ne peut pas supposer qu'il le vérifiera (illustration dans Emacs).

Par ailleurs, il est inélégant que l'utilisateur reçoive une erreur destinée au développeur. Dans le cadre d'une saisie utilisateur, on peut plutôt proposer ceci:

[7]:
```
// Solution #4:
int main ()
{
    printf("Entrez un réel positif: ") ;
    double x ;
    scanf("%lf",&x) ;
    while (x < 0)
    {
        printf("J'ai dit POSITIF ! Essaie encore: ") ;
        scanf("%lf", &x) ;
    }
    printf("La racine carre entiere est: %d", (int)(sqrt(x))) ;
    // Suite du code ici
}
```

Cela permet à l'utilisateur de se corriger, plutôt que d'interrompre tout le programme.

1.7.3 Propriété des sorties d'une fonction

Il peut aussi être intéressant de tester des propriétés en sortie de fonction:

[38]:
```
#include <stdio.h>
#include <assert.h>
#include <math.h>
```

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

int maxIndexTab (int longueur, int t[longueur])
{
    assert(longueur > 0) ;
    int max = t[0];
    int maxIndex = 0 ;
    for (int i = 1; i < longueur; i++)
    {
        if (t[i] > max)
        {
            max = t[i] ;
            maxIndex = i ;
        }
    }
    assert(0 <= maxIndex && maxIndex < longueur) ;
    return maxIndex ;
}
```

La fonction $maxIndex$ retourne un indice du tableau t, cet indice doit donc être compris entre 0 et $longueur - 1$.

De manière générale, lorsqu'on accède à une case de tableau $t[i]$, il convient toujours de se demander si i est effectivement un indice valide. S'il y a un doute, autant mettre un *assert*.

[40]:
```c
void initTabRand(int t[],int longueur)
{
    for (int i = 0; i < longueur; i++)
        t[i] = rand() ;
}
```

[46]:
```c
#include <stdlib.h>
#include <time.h>

int main ()
{
    srand(time(NULL));   // Initialise le generateur de nombres␣
 ↪pseudo-aleatoires (hors programme).
    int t[100] ;
    initTabRand(t,100) ; // remplit le tableau de valeurs aleatoires
    int maxIndex = maxIndexTab(100,t) ;
    printf("Le plus grand elt du tableau se situe a l'indice %d et vaut %d\n",␣
 ↪maxIndex,t[maxIndex]) ;
}
main () ;
```

```
Le plus grand elt du tableau se situe a l'indice 88 et vaut 2121015879
```

1.8 Instructions de modification du flot de contrôle (TD ?)

Il existe en C plusieurs instructions qui modifie le flot de contrôle tel que nous venons de le voir.

1.8.1 Instruction “Return”

[10]:
```
// Teste si syracuse(i) < ceil pour tout i <= N.
bool test (int ceil, int N)
{
    for (int i = 0; i <= N; i++)
    {
        if (syracuse(i) >= ceil)
            return false ;
    }
    return true ;
}
```

```
input_line_19:6:13: error: use of undeclared identifier 'syracuse'
        if (syracuse(i) >= ceil)
            ^
```

```
Interpreter Error:
```

1.8.2 Instruction “Assert”

Illustration sur $maxIndexTab$.

1.8.3 Instruction “Break”

[ ]:
```
void sum41 ()
{
    int n ;
    int result  = 0 ;
    printf("Entrez des entiers pour en faire la somme, -1 pour terminer.\n") ;
    for (int i = 0; i < 41; i++)
    {
        scanf("%d",&n) ;
        if (n < 0)
            break ;
        result = result + n ;
    }
    printf("La somme totale est de %d",result) ;
}
```

1.8.4 Instruction “Continue”

L’instruction CONTINUE n’est pas explicitement au programme mais son comportement est complémentaire à celui de l’instruction BREAK:

[ ]:
```
void sum (int max)
{
    int n ;
    int result  = 0 ;
    printf("Entrez %d entiers pour en faire la somme, les negatifs sont ignores.
 ↪\n",max) ;
    for (int i = 0; i < max; i++)
    {
        scanf("%d",&n) ;
        if (n < 0)
            continue ;
        result = result + n ;
    }
    printf("La somme totale est de %d",result) ;
}
```

1.8.5 Instruction "GOTO" (Hors Programme)

L'instruction GOTO permet d'ajouter un arc au graphe de flot contrôle entre deux sommets quelconques. Dans un programme qui utilise l'instruction GOTO, il devient difficile de calculer les exécutions de tête; c'est pour cela que l'utilisation de cette instruction est déconseillée (et hors programme pour vous).

1.8.6 Exercice:

Exercice: réécrire les programmes précédents sans les instructions BREAK et CONTINUE et en utilisant l'instruction RETURN qu'en fin de programme.

1.9 Graphes de flot de contrôle de programmes récursifs (TD)

[ ]:
```
double power (double x, int n)
{
    if (n == 0 )
        return 1 ;
    else if (n % 2 == 0)
        return (power(x,n/2) * power(x,n/2)) ;
    else
        return (x*power(x,n/2) * power(x,n/2)) ;
}
```

[ ]:
```
printf("%lf",power(2,5)) ;
```

1.10 Organisation du code: structures en C

En C, on peut créer des types: - enum: type énumération, contiennent un nombre fini d'objet. - struct: type enregistrement (record en anglais), ou type produit. Contiennent un certains nombre de champs.

[5]:
```
enum lyceeFR { MASSENA_NICE, VHUGO_BESANCON, MONTAIGNE_BORDEAUX,␣
↪SAINTLOUIS_PARIS } ;
```

[6]:
```
enum lyceeFR monLycee = SAINTLOUIS_PARIS ;
```

[7]:
```
struct profilEleve {
   char* nom ;
   lyceeFR lycee ;
   double moyMath ;
   double moyPhy ;
   double moyInfo ;
   double moyFr ;
   double moyEng ;
   char* comment ;
 };
```

[8]:
```
double moyGen (struct profilEleve eleve)
{
  return ((10*eleve.moyMath + 7*eleve.moyPhy + 7*eleve.moyInfo + 3*eleve.moyFr␣
 ↪+ 4*eleve.moyEng) / 31) ;
}
```

Les types struct permettent d'améliorer la lisibilité et l'organisation du code.
Cela permet aussi d'écrire une fonction qui renvoit plusieurs valeurs (on verra une autre façon de faire au prochain cours):

[12]:
```
struct profilEleve inputProfil ()
{
    printf("Entrez votre nom: ") ;
    char* nomEleve ;
    nomEleve = (char*)(malloc(sizeof(char)*50));
    scanf("%s", nomEleve) ;
    printf("\n") ;
    printf("Entrez vos moyennes de Math, Physique, Informatique, Francais et␣
 ↪Anglais, separees par des espaces: \n") ;
    double math, phys, info, fr, eng ;
    scanf("%lf %lf %lf %lf %lf",&math, &phys, &info, &fr, &eng) ;
    char comment[100] = "RAS" ;
    printf("\n") ;
    struct profilEleve eleve =
      { nomEleve, SAINTLOUIS_PARIS, math, phys, info, fr, eng, comment } ;
    return eleve ;
} ;
```

[13]:
```
int main ()
{
    profilEleve eleve = inputProfil () ;
```

```
    printf("L'eleve %s a une moyenne generale de: %lf \n", eleve.nom,␣
 ↪moyGen(eleve)) ;
    free(eleve.nom) ;
}
```

Dernière chose: on peut modifier le nom d'un type existant. C'est pratique pour les type STRUCT ou ENUM:

[21]:
```
typedef struct profilEleve PE ;
char nom[6] = "Simon" ;
PES a = { nom, 12 } ;
```

Afin de conserver un nom intelligibile, on écrira le plus souvent:

[22]:
```
typedef struct profilEleve profilEleve ;
```

Cela fonctionne aussi avec les type de base (mais vous n'en aurez sans doute pas l'utilité):

[1]:
```
typedef int NB ;
NB b = 4 ;
printf("%d", b) ;
```

```
4
```

[ ]:
```

```