TP Programmation C/C++

Structures fondamentales d'algorithmie

Éléments de cours

Sommaire

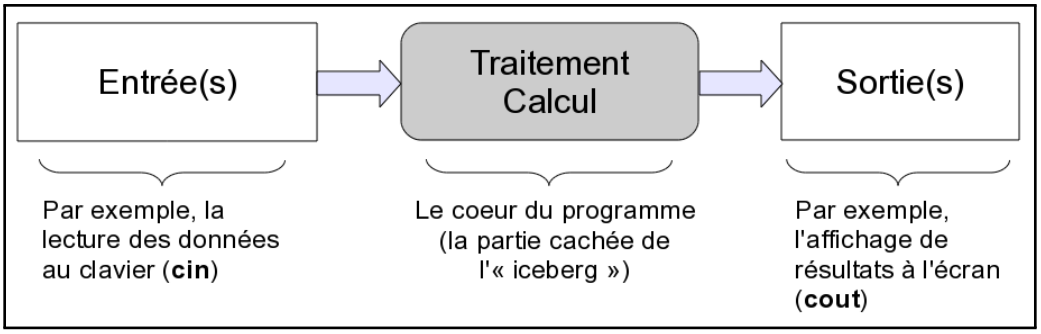
Les objectifs de ce tp sont de comprendre et mettre en oeuvre des variables dans les structures fondamentales d'algorithmie : enchaînements, alternatives, itérations. Les exercices sont extraits du site www.france-ioi.org.

Les critères d'évaluation de ce TP sont :

- le minimum attendu : on doit pouvoir fabriquer un exécutable et le lancer !
- le programme doit énoncer ce qu'il fait et faire ce qu'il énonce !
- le respect des noms de fichiers
- le nommage des variables ainsi que leurs types, l'utilisation de constantes
- le code est fonctionnel
- la simplicité du code

Un programme informatique

Un programme informatique (simple) est souvent structuré de la manière suivante :



Un **traitement est tout simplement l'action de produire des sorties à partir d'entrées** : les vrais programmes ont donc tendance à produire des résultats en fonction de l'entrée qu'on leur fournit.

Objectifs du programmeur

Le métier de programmeur consiste à écrire des programmes qui :

- donnent des résultats corrects
- sont simples
- sont efficaces

L'ordre donné ici est très important : peu importe qu'un programme soit rapide si ses résultats sont faux. De même, un programme correct et efficace peut être si compliqué et mal écrit qu'il faudra le jeter ou le récrire complètement pour en produire une nouvelle version. N'oubliez pas que les programmes utiles seront toujours modifiés pour répondre à de nouveaux besoins.

Un programme (ou une fonction) doit s'acquitter de sa tâche de façon aussi simple que possible.

Nous acceptons ces principes quand nous décidons de devenir des professionnels. En termes pratiques, cela signifie que nous ne pouvons pas nous contenter d'aligner du code jusqu'à ce qu'il ait l'air de fonctionner : nous devons nous soucier de sa structure. Paradoxalement, le fait de s'intéresser à la structure et à la "qualité du code" est souvent le moyen le plus facile de faire fonctionner un programme.

En programmation, les principes à respecter s'expriment par des **règles de codage** et des **bonnes pratiques**.

☞ Exemples :

- **Règle de codage** : les valeurs (comme `10` pour un MAXIMUM ou `3.14` pour PI) dans une expression doivent être déclarées comme des CONSTANTES. En `C`, les CONSTANTES s'écrivent en MAJUSCULES.
- **Bonne pratique** : Un programme (ou une fonction) ne doit pas dépasser 15 lignes de `C/C++` (accolades exclues).

Entrées et sorties

Pour l'instant, nos programmes se limiteront à lire des entrées en provenance du **clavier** de l'utilisateur et de produire des résultats en sortie sur l'**écran** de celui-ci.

Pour lire des entrées saisies au clavier (l'entrée standard `stdin`), on utilisera :

– `scanf()` en C ou C++
– `cin` en C++

✍ *Ne vous préoccuper pas des saisies invalides. Pour écrire des programmes qui fonctionnent et qui sont simples, il est plus prudent pour l'instant de supposer que l'on fait face à un utilisateur « parfait ». Par exemple, lorsque on souhaite lire un entier, l'utilisateur « parfait » saisit un entier !*

Pour afficher des résultats à l'écran (la sortie standard `stdout`), on utilisera :

– `printf()` en C ou C++
– `cout` en C++

✍ *Evidemment, il existe d'autres types d'entrées/sorties (fichier, réseau, base de données, ...) pour les programmes. Nous les verrons plus tard.*

Objets, types et valeurs

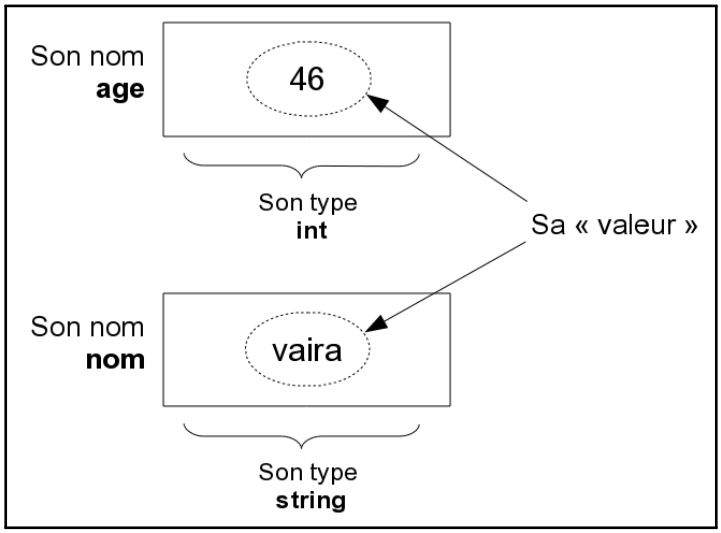
Pour pouvoir lire quelque chose, il faut dire où le placer ensuite. Autrement dit, il faut un "endroit" dans la mémoire de l'ordinateur où placer les données lues. On appelle cet "endroit" un **objet**.

Un objet est une région de la mémoire, dotée d'un **type** qui spécifie quelle sorte d'information on peut y placer. Un objet nommé s'appelle une **variable**.

☞ *Exemples :*

– les **entiers** sont stockés dans des objets de type `int`
– les **réels** sont stockés dans des objets de type `float` ou `double`
– les **chaînes de caractères** sont stockées dans des objets de type `string` en C++
– etc ..

Vous pouvez vous représenter un objet comme "une boite" (une case) dans laquelle vous pouvez mettre une **valeur** du **type** de l'objet :



Représentation d'objets

✍ *Il est fondamentalement impossible de faire quoi que ce soit avec un ordinateur sans stocker des données en mémoire (on parle ici de la RAM).*

Une instruction qui définit une variable est ... une **définition** !

✍ *Une déclaration est l'action de nommer quelque chose et une définition de la faire exister. Les déclarations (situées dans des fichiers* `.h`*) sont utilisées pendant la phase de compilation et les définitions (dans les fichiers* `.cpp`*) sont indispensables à l'édition des liens pour fabriquer un exécutable.*

Une définition peut (et généralement doit) fournir une **valeur initiale**. Trop de programmes informatiques ont connu des *bugs* dûs à des oublis d'initialisation de variables. On vous obligera donc à le faire systématiquement. On appelle cela "respecter une règle de codage". Il en existe beaucoup d'autres.

```
int nombreDeTours = 100; // Correct 100 est une valeur entière
string prenom = "Robert"; // Correct "Robert" est une chaîne de caractères

// Mais :
int nombreDeTours = "Robert"; // Erreur : "Robert" n'est pas une valeur entière
string prenom = 100; // Erreur : 100 n'est pas une chaîne de caractères (il manque les
   guillemets)
```

Initialisation de variables

✍ *Le compilateur se souvient du type de chaque variable et s'assure que vous l'utilisez comme il est spécifié dans sa définition.*

Le C++ dispose de nombreux types. Toutefois, vous pouvez écrire la plupart des programmes en n'en utilisant que cinq :

```
int nombreDeTours = 100; // int pour les entiers
double tempsDeVol = 3.5; // double pour les nombres en virgule flottante (double précision)
string prenom = "Robert"; // string pour les chaînes de caractères
char pointDecimal = '.'; // char pour les caractères individuels ou pour des variables
   entières sur 8 bits (un octet)
bool ouvert = true; // bool pour les variables logiques (booléenes)
```

Les types usuels en C++

✍ *Le type* `string` *n'existe pas en langage* `C`*. Pour manipuler des chaînes de caractères en* `C`*, il faudra utiliser des tableaux que l'on verra plus tard.*

```
#include <stdbool.h> /* pour le type bool en C */

int nombreDeTours = 100; // int pour les entiers
double tempsDeVol = 3.5; // double pour les nombres en virgule flottante (double précision)
char prenom[] = "Robert"; // un tableau de caractères pour stocker les chaînes de caractères
char pointDecimal = '.'; // char pour les caractères individuels ou pour des variables
   entières sur 8 bits (un octet)
bool ouvert = true; // bool pour les variables logiques (booléenes)
```

Les types usuels en C

Nommer une variable

Un nom de variable est un nom principal (surtout pas un verbe) suffisamment éloquent, éventuellement complété par :
– une caractéristique d'organisation ou d'usage
– un qualificatif ou d'autres noms

On utilisera la convention suivante : **un nom de variable commence par une lettre minuscule puis les différents mots sont repérés en mettant en majuscule la première lettre d'un nouveau mot**.

Exemples : `distance`, `distanceMax`, `consigneCourante`, `etatBoutonGaucheSouris`, `nbreDEssais`, ...

Certaines abréviations sont admises quand elles sont d'usage courant : `nbre` (ou `nb`), `max`, `min`, ...

Les lettres `i`, `j`, `k` utilisées seules sont usuellement admises pour les indices de boucles.

Un nom de variable doit être uniquement composé de lettres, de chiffres et de "souligné" (_). Les noms débutant par le caractère "souligné" (_) sont réservés au système, et à la bibliothèque C.

Les noms débutants par un double "souligné" (__) sont réservés aux constantes symboliques privées (`#define` ...) dans les fichiers d'en-tête (`.h`).

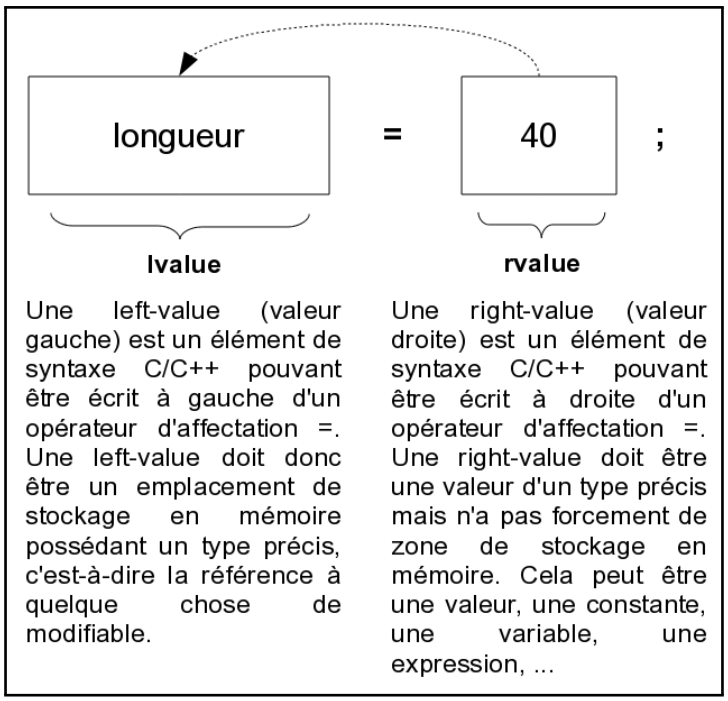
Il est déconseillé de différencier deux identificateurs uniquement par le type de lettre (minuscule/majuscule). Les identificateurs doivent se distinguer par au moins deux caractères, parmi les 12 premiers, car pour la plupart des compilateurs seuls les 12 premiers symboles d'un nom sont discriminants.

Les mots clés du langage sont interdits comme noms.

✍ *l'objectif de respecter des règles de codage est d'augmenter la lisibilité des programmes en se rapprochant le plus possible d'expressions en langage naturel.*

Expressions et instructions

La brique de base la plus élémentaire d'un programme est une **expression**. Une expression calcule une **valeur** à partir d'un certain nombre d'opérandes. Cela peut être une **valeur littérale** comme 10, 'a', 3.14, "rouge" ou le **nom d'une variable**.



On utilise aussi des opérateurs dans une instruction :

```
// calcule une aire
int longueur = 40;
int largeur = 20;
int aire = longueur * largeur; // * est l'opérateur multiplication
```

✍ *Il existe de nombreux opérateurs : les opérateurs arithmétiques (`+`, `-`, `*`, `/` et `%`), les opérateurs relationnels (`<`, `<=`, `>`, `>=`, `==` et `!=`), les opérateurs logiques && (et), `||` (ou) et `!` (non), les opérateurs bits à bits & (et), `|` (ou) et `~` (non) ...*

De manière générale, un programme informatique est constitué d'une **suite d'instructions**.

Il y a plusieurs sortes d'instructions : les déclarations, les instructions d'expression, les instructions conditionnelles, les instructions itératives (boucles), etc ...

Une **instruction d'expression** est une expression suivie d'un point-virgule (`;`). Le point-virgule (`;`) est un élément syntaxique permettant au compilateur de "comprendre" ce que l'on veut faire dans le code (comme la ponctuation dans la langue française).

Dans les programmes comme dans la vie, il faut souvent choisir entre plusieurs possibilités. Le C/C++ propose plusieurs **instructions conditionnelles** : l'instruction `if` (choisir entre deux possibilités) ou l'instruction `switch` (choisir entre plusieurs possibilités).

Il est rare de faire quelque chose une seule fois. C'est pour cela que tous les langages de programmation fournissent des moyens pratiques de faire quelque chose plusieurs fois (on parle de traitement itératif). On appelle cela une **boucle** ou une **itération**.

Le C/C++ offrent plusieurs **instructions itératives** : la boucle `while` (et sa variante `do ... while`) et la boucle `for`.

Conditionner une action

La célèbre attraction du train fou est interdite aux moins de 10 ans. On souhaite écrire un programme qui demande à l'utilisateur son âge et qui, si la personne a moins de 10 ans, affiche le texte « Accès interdit » ; ce qui peut se rédiger comme cela :

```
Variable age : Entier

age <- Lire un entier
Si age < 10
   Ecrire "Accès interdit"
```

Cela se traduit en C :

```c
int age;

scanf("%d", &age);
if (age < 10)
{
   printf("Accès interdit\n");
}
```

On écrit donc le mot-clef `if`, la traduction en anglais de « **si** », puis on met entre parenthèses la condition à tester, à savoir `age < 10`. On n'oublie pas de de mettre des accolades.

Ainsi, l'accès est interdit à un enfant de 8 ans :

```
$ ./tester-age
8
Accès interdit
```

À l'opposé, le programme n'affiche rien pour un âge de 13 ans :

```
$ ./tester-age
13
```

Pour exprimer la condition du « si » dans le programme, on a utilisé le symbole `<`, qui est l'opérateur de comparaison strictement inférieur. De manière symétrique, l'opérateur `>` permet de tester si un nombre

est strictement supérieur à un autre. Lorsqu'on veut tester si un nombre est inférieur ou égal à un autre, on utilise le symbole `<=`. De manière symétrique, le symbole `>=` permet de tester si un nombre est supérieur ou égal à un autre.

Par exemple, le code suivant permet de tester si la température de l'eau a atteint 100 degrés :

```
int temperature;

scanf("%d", &temperature);
if (temperature >= 100)
{
   printf("L'eau bout !");
}
```

Pour finir, le symbole == permet de tester l'égalité et la différence avec != . Evidemment, il ne faut surtout pas confondre avec l'opérateur = qui permet d'effectuer une affectation.

La notion de base est donc simple mais il est également facile d'utiliser `if` de façon trop simpliste.

Voici un exemple simple de programme de conversion *cm/inch* qui utilise une instruction `if` :

```
int main()
{
    const double conversion = 2.54; // nombre de cm pour un pouce (inch)
    int longueur = 1;             // longueur (en cm ou en in)
    char unite = 0; // 'c' pour cm ou 'i' pour inch
    cout << "Donnez une longueur suivi de l'unité (c ou i):\n";
    cin >> longueur >> unite;

    if (unite == 'i')
        cout << longueur << " in == " << conversion*longueur << " cm\n";
    else
        cout << longueur << " cm == " << longueur/conversion << " in\n";
}
```

Conversion cm/inch (version 1)

En fait cet exemple semble seulement fonctionner comme annoncé. Ce programme dit que si ce n'est pas une conversion en *inch* c'est forcément une conversion en cm. Il y a ici une dérive sur le comportement de ce programme si l'utilisateur tape '`f`' car il convertira des cm en *inches* ce qui n'est probablement pas ce que l'utilisateur désirait. Un programme doit se comporter de manière sensée même si les utilisateurs ne le sont pas.

Voici une version améliorée en utilisant une instruction `if` imbriquée dans une instruction `if` :

```
if (unite == 'i')
   cout << longueur << " in == " << conversion*longueur << " cm\n";
else if (unite == 'c')
   cout << longueur << " cm == " << longueur/conversion << " in\n";
else
   cout << "Désolé, je ne connais pas cette unité " << unite << endl;
```

Conversion cm/inch (version 2)

De cette manière, vous serez tenter d'écrire des tests complexes en associant une instruction `if` à chaque condition. Mais, rappelez-vous, le but est d'écrire du code simple et non complexe.

En réalité, la comparaison d'unité à 'i' et à 'c' est un exemple de la forme de sélection la plus courante : une sélection basée sur la comparaison d'une valeur avec plusieurs constantes. Le C/C++ fournit pour cela l'instruction `switch`.


```
int main()
{
    const double conversion = 2.54; // nombre de cm pour un pouce (inch)
    int longueur = 1;             // longueur (en cm ou en in)
    char unite = 0; // 'c' pour cm ou 'i' pour inch
    cout << "Donnez une longueur suivi de l'unité (c ou i):\n";
    cin >> longueur >> unite;

    switch (unite)
    {
      case 'i':
        cout << longueur << " in == " << conversion*longueur << " cm\n";
        break;
      case 'c':
        cout << longueur << " cm == " << longueur/conversion << " in\n";
        break;
      default:
        cout << "Désolé, je ne connais pas cette unité " << unite << endl;
        break;
    }
}
```

Conversion cm/inch (version 3)

L'instruction `switch` utilisée ici sera toujours plus claire que des instructions `if` imbriquées, surtout si l'on doit comparer à de nombreuses constantes.

Vous devez garder en mémoire ces particularités quand vous utilisez un `switch` :

- la valeur utilisée pour le `switch()` doit être un entier, un `char` ou une énumération (on verra cela plus tard). Vous ne pourrez pas utiliser un `string` par exemple.
- les valeurs des étiquettes utilisées dans les `case` doivent être des expressions constantes (voir plus loin). Vous ne pouvez pas utiliser de variables.
- vous ne pouvez pas utiliser la même valeur dans deux `case`
- vous pouvez utiliser plusieurs `case` menant à la même instruction
- l'erreur la plus fréquente dans un `switch` est l'oubli d'un `break` pour terminer un case. Comme ce n'est pas une obligation, le compilateur ne détectera pas ce type d'erreur.

Les variables booléennes

En C/C++ le programme suivant :

```
if (prix < 10)
{
   printf("Pas cher");
}
```

peut aussi s'écrire :

```
#include <stdbool.h> /* nécessaire en C */

bool estPasCher = (prix < 10);

if (estPasCher)
{
```



```
   printf("Pas cher");
}
```

La variable `estPasCher` est appelée une **variable booléenne** ou un **booléen** de type `bool` car elle ne peut être que vraie ou fausse, ce qui correspond en `C/C++` aux valeurs `true` (pour vrai) et `false` (pour faux).

✍ *N'oublier pas que le type* `bool` *est natif en* `C++`*. Par contre en* `C`*, il faut inclure* `stdbool.h` *pour pouvoir l'utiliser.*

Une maladresse classique avec les booléens est de faire quelque chose comme ceci :

```
int prix;
scanf("%d", &prix);
bool estCher = (prix > 100)

if (estCher == true)
{
   printf("C'est cher !");
}
```

Le code est correct mais on n'a pas besoin de tester si quelque chose est égal à `true` ou `false` , si ce quelque chose est lui même déjà `true` ou `false`!

On écrira donc :

```
if (estCher)
{
   printf("C'est cher !");
}

// ou :

if (!estCher)
{
   printf("Pas cher"));
}
```

Il est bien sûr possible d'utiliser des opérateurs booléens (les opérateurs && et `||`) pour combiner des conditions et les valeurs booléennes sont également utilisables.

Voici quelques extraits de code à titre d'exemple :

```
bool estSenior = (age >= 60);
bool estJeune = (age <= 25) && (age >= 12);
bool reductionPossible = (estSenior || estJeune);

if (reductionPossible)
{
   printf("Réduction!");
}

while (motDePasse != secret || agePersonne <= 3)
{
   printf("Accès refusé : mauvais mot de passe ou personne trop jeune\n");
   scanf("%d %d", &agePersonne, &motDePasse);
}
```



```
while (nbPersonnes <= nbMax && temperature <= 45)
{
   printf("Portes ouvertes\n");
   nbPersonnes = nbPersonnes + 1;
   scanf("%d", &temperature);
}
```

Itérer une action

Le premier programme jamais exécuté sur un ordinateur à programme stocké en mémoire (l'EDSAC) est un exemple d'itération. Il a été écrit et exécuté par David Wheeler au laboratoire informatique de Cambridge le 6 mai 1949 pour calculer et afficher une simple liste de carrés comme ceci :

```
0  0
1  1
2  4
3  9
4  16
...
98 9604
99 9801
```

Ce premier programme n'a pas été écrit en C/C++ mais le code devait ressembler à ceci :

```
// Calcule et affiche une liste de carrés

#include <iostream>

using namespace std;

int main()
{
   int i = 0; // commencer à 0

   // tant que i est inférieur strict à 100 : on s'arrête quand i a atteint la valeur 100
   while (i < 100)
   {
      cout << i << '\t' << i * i << '\n'; // affiche i et son carré séparés par une
         tabulation
      ++i; // on incrémente le nombre et on recommence
   }

   return 0;
}
```

Le premier programme jamais écrit (version while)

Les accolades (`{}`) délimitent le **corps de la boucle** : c'est-à-dire le bloc d'instructions à répéter. La condition pour la répétition est exprimée directement dans le `while`. La boucle `while` s'exécutera **0 ou n fois**.

Il existe aussi une boucle `do ... while` à la différence près que cette boucle sera exécutée **au moins une fois**.


```
// faire
do
{
   cout << i << '\t' << i * i << '\n'; // affiche i et son carré séparés par une tabulation
   ++i; // on incrémente le nombre et on recommence
}
while (i < 100); // tant que i est inférieur strict à 100
```

La boucle do .. while

Donc écrire une boucle est simple. Mais cela peut s'avérer dangereux :
- Que se passerait-il si `i` n'était pas initialisé à `0` ? Voilà une première raison qui démontre que les variables non initialisées sont une source d'erreurs courante.
- Que se passerait-il si on oubliait l'instruction `++i` ? On obtient une boucle infinie (un programme qui ne "répond" plus). Il faut éviter au maximum d'écrire des boucles infinies. Il est conseillé d'éviter ce type de boucle : `while(1)` ou `while(true)` qui sont des boucles infinies.

Itérer sur une suite de nombres est si courant en C/C++ que l'on dispose d'une instruction spéciale pour le faire. C'est l'instruction `for` qui très semblable à `while` sauf que la gestion de la variable de contrôle de boucle est concentrée sur une seule ligne plus facile à lire et à comprendre. Il existe toujours une instruction `while` équivalente à une instruction `for`.

L'instruction `for` concentre : **une zone d'initialisation, une zone de condition et une zone d'opération d'incrémentation**. N'utilisez `while` que lorsque ce n'est pas le cas.

```
// Calcule et affiche une liste de carrés

#include <iostream>
#include <cmath>

using namespace std;

int main()
{
   // exécute le bloc d'instructions de la boucle :
   // avec i commençant à 0 (initialisation)
   // tant que i est inférieur strict à 100 (condition)
   // en incrémentant i après chaque exécution du bloc d'instruction (opération d'
      incrémentation)
   for (int i = 0; i < 100; i++)
      cout << i << '\t' << pow(i, 2) << '\n'; // affiche i et son carré séparés par une
         tabulation

   return 0;
}
```

Le premier programme jamais écrit (version for)

☞ *Ici, on utilise la fonction puissance (`pow`) de la bibliothèque mathématique. Pour cela, il faut inclure `math.h` en C ou `cmath` en C++ puis effectuer l'édition des liens avec l'option `-lm` (ce qui est fait par défaut maintenant).*

Bilan

Vous avez découvert les notions suivantes :

– lire des entrées et produire des résultats en sortie
– enchaîner des suites d'instructions
– répéter (itérer) des instructions
– manipuler des variables et faire des opérations avec
– effectuer des actions sous conditions

Conclusion : parce qu'il est clair que vous débutez à peine votre carrière de programmeur, n'oubliez pas qu'écrire de bons programmes, c'est écrire des programmes corrects, simples et efficaces.

Rob Pike (ancien chercheur des Laboratoires Bell et maintenant ingénieur chez Google) :

> « *Règle n°4 : Les algorithmes élégants comportent plus d'erreurs que ceux qui sont plus simples, et ils sont plus difficiles à appliquer. Utilisez des algorithmes simples ainsi que des structures de données simples.* »

Cette règle n°4 est une des instances de la philosophie de conception KISS (*Keep it Simple, Stupid* dans le sens de « Ne complique pas les choses ») ou Principe KISS, dont la ligne directrice de conception préconise de rechercher la simplicité dans la conception et que toute complexité non nécessaire devrait être évitée.

Annexe 0 : printf() et scanf()

Il exste trois flux de base pour chaque programme :

- `stdin` (0) : Entrée standard (par défaut le clavier)
- `stdout` (1) : Sortie standard (par défaut l'écran)
- `stderr` (2) : Sortie erreur (par défaut l'écran également)

✍ *En C, un flux est un canal destiné à transmettre ou à recevoir de l'information. Il peut s'agir de fichier ou de périphériques.*

Pour afficher des résultats à l'écran, il faudra écrire sur le flux `stdout`. Pour cela, on utilise généralement la fonction `printf()` (déclarée dans `stdio.h`) :

```
#include <stdio.h>

printf("Hello World!\n");

// ce qui correspond à :
fprintf(stdout, "Hello World!\n");

// ou :
write(1, "Hello World!\n"); // 1 étant le descripteur de stdout
```

Pour afficher une erreur, il est recommandé de la faire dans le flux `stderr` :

```
// En C/C++
fprintf(stderr, "Ceci est une erreur.\n");

// En C++
cerr << "Ceci est une erreur.\n";
```

✍ *Si on ne désire pas l'affichage des messages d'erreur d'un programme, il suffit de rediriger son flux* ***stderr*** *vers le périphérique* ***null****. Par exemple pour le programme* ***a.out*** *:*

```
$ ./a.out 2> /dev/null
```

La fonction `printf()` (ainsi que `sprintf()` et `fprintf()` prend en argument une chaîne de format qui spécifie les types de ses autres arguments. La chaîne de format de `printf()` est riche en possibilités, car elle indique non seulement le type des arguments mais aussi la manière de les écrire. Par exemple, vous pouvez choisir le nombre de chiffres après la virgule d'un nombre flottant. Chaque code spécial de la chaîne de format est précédé d'un `%`. Si vous voulez écrire le symbole `%`, vous devez l'écrire deux fois : `%%`.

```
printf("%d\n", 42); // affichage d'un entier
printf("Taux de réussite : %.1f %%\n", 92.9); // Taux de réussite : 92.9 %
```

Chaque code spécial est découpé en plusieurs sections (les sections entre crochets sont facultatives) :

```
%[largeur][.precision]type
```

1. Largeur : nombre minimum de caractères à écrire, `printf()` complètera si nécessaire par des espaces sur la gauche, la valeur est alignée à droite.
2. Précision : nombre de chiffres après la virgule à écrire pour les nombres flottants, précédé d'un point. La valeur est automatiquement arrondie : le dernier chiffre est incrémenté s'il y a besoin.
3. Type : le type de la valeur à écrire

Voici les différents types utilisables :

%d	int	entier 32 bits en décimal
%u	unsigned int	entier 32 bits non signé en décimal
%x	int	entier 32 bits en hexadécimal (en minuscules et %X en majuscules)
%lld	long long int	entier 64 bits
%f	float ou double	avec par défaut 6 chiffres après la virgule
%g	float ou double	en utilisant la notation 1.234e5 quand cela est approprié
%c	char	caractère unique
%s	char*	chaîne de caractères
%p	void*	l'adresse mémoire en hexadécimal

☞ *man 3 printf*

La fonction `scanf()` permet de lire des données sur l'entrée standard (`stdin`). Cette fonction prend comme premier argument une chaîne de format, qui indique combien de variables vous souhaitez lire, et de quel type. Ensuite vous donnez passer en arguments des pointeurs sur les variables qui recevront le résultat, dans l'ordre de la chaîne de format.

Prenons le cas le plus simple :

```c
#include <stdio.h>

// lire un entier et le placer dans la variable n :
int n;
scanf("%d", &n); // ne pas oubliez le & qui donne l'adresse de la variable n, c'est
   nécessaire ici pour que la fonction puisse modifier la variable passée en argument

// lire deux entiers :
int a, b;
scanf("%d %d", &a, &b);
```

✍ *La lecture des chaînes de caractère (avec* `scanf()` *en C/C++ ou* `cin` *en C++) se termine sur ce qu'on appelle un espace blanc (whitespace), c'est-à-dire le caractère espace, une tabulation ou un caractère de retour à la ligne (généralement la touche* Enter *). Notez que les espaces sont ignorés par défaut. Sous Linux, vous pouvez indiquer la fin d'un flux (EOF) en combinant les touches* Ctrl + d *qui indiquera qu'il n'y a plus de saisie. Vous pouvez aussi stopper le programme avec* Ctrl + z *ou l'interrompre avec* Ctrl + c .

La chaîne de format se compose de codes spéciaux indiquant le type des valeurs à lire. Chaque code est précédé du symbole %. Il en existe une multitude, nous nous contenterons ici de ceux dont vous aurez besoin :

%d	int	entier 32 bits en décimal
%lld	long long int	entier 64 bits
%f	float	nombre réel simple précision
%lf	double	nombre réel double précision
%c	char	caractère unique
%s	char*	chaîne de caractères

☞ *man 3 scanf*

Annexe 1 : erreurs du débutant

✌ *Le bug du débutant n°1 :* il ne faut pas mettre un ; après un `if` car cela exécuterait une instruction nulle si expression est vraie. Cela donne un comportement défectueux :

```
int a = 0;
if(a != 0); /* c'est sûrement un bug involantaire */
   printf("bug : a n'est pas nul et pourtant a = %d !\n", a);
```

✌ *Le bug du débutant n°2 :* il ne faut pas confondre l'opérateur '=' (d'affectation) avec l'opérateur '==' (comparaison d'égalité). Cela risque de donner un comportement défectueux :

```
int a = 0;
if(a = 0) /* c'est sûrement un bug involantaire */
     printf("a = %d : a est égal à 0\n", a);
else  printf("bug : a n'est pas égal à 0 et pourtant a = %d !\n", a);
```

✌ *Le bug du débutant n° 3 :* il faut penser à déterminer si un `break` est nécessaire dans un `case`. Le code suivant le prouve :

```
a = 1;
switch (a)
{
   case 1: printf("a = %d : a est égal à 1\n", a); /* ne faudrait-il pas un break ? */
   case 2: printf("bug : a = %d : a est égal à 2 !\n", a);
}
```

Vous obtiendrez :

```
a = 1 : a est égal à 1
bug : a = 1 : a est égal à 2 !
```

✌ *Le bug du débutant n°4 :* il ne faut pas mettre un ; après le `while` car cela exécuterait une instruction nulle si expression est vraie. Le code suivant sera dans une **boucle infinie** sans afficher aucun message "Salut" :

```
long compteur = 15;
while (compteur > 0); // c'est sûrement un bug involantaire
{
    printf("Salut\n");
    compteur = compteur - 1;
} // ici non plus ne pas mettre de ;
```

✌ *Le bug du débutant n°5 :* il ne faut pas mettre un ; à la fin d'un `for` car cela exécuterait une instruction nulle si expression est vraie. Le code suivant n'affichera qu'un seul message "Salut" car le `for` boucle dans le vide :

```
long compteur;
for (compteur = 0; compteur < 15; compteur++); // c'est sûrement un bug involantaire
{
    printf("Salut\n");
}
```

✌ *Le bug du débutant n°6 :* il ne faut pas oublier les accolades `{}` à la suite d'une instruction conditionnelle `if/else` ou itérative `for/while` sinon vous n'exécuterez qu'un seule instruction au lieu de plusieurs. Par exemple :


```
Si temperature >= 100
   Écrire "L'eau bout !"
   Écrire "Préparons le thé"
```

Il ne faut pas coder :

```
if (temperature >= 100)
   printf("L'eau bout !\n");
   printf("Préparons le thé\n");
```

Si la température est égale ou supérieure à 100, ce code affichera :

```
L'eau bout !
```

La bonne traduction de l'algorithme est la suivante :

```
if (temperature >= 100)
{
   printf("L'eau bout !\n");
   printf("Préparons le thé\n");
}
```

Annexe 2 : opérations sur les nombres réels

La bibliothèque C standard contient une série de fonctions de calcul sur les flottants (racine carrée, logarithme, sinus, etc.), que l'on peut utiliser en incluant le fichier `math.h` : `fabs()`, `floor()`, `fmod()`, `exp()`, `log()`, `log10()`, `sqrt(x)`, `pow()`, `sin(x)`, `cos(x)`, `tan()`, ...

Une des différences majeures entre le traitement des entiers et des flottants est la perte de précision. Avec les entiers, les calculs sont toujours exacts, du moment que l'on ne dépasse pas la capacité des entiers. Avec les flottants, chaque opération (addition, multiplication, etc.) induit une **perte de précision**.

Il existe deux types de flottants en C/C++ : `float` et `double`. Les `double` (codés sur 8 octets) ont une meilleur précision que les `float` (codés sur 4 octets). Il donc est conseillé de toujours utiliser des `double` afin de garder la meilleure précision possible, le seul intérêt des `float` est qu'ils prennent deux fois moins de place en mémoire.

Si vous calculez : 1e10 + 1e-10 = 1.00000000000000000001e10, l'ordinateur arrondira à 1.00000000000000e10 = 1e10 (15 chiffres pour un `double`). On a donc des cas où A + B = A même si B != 0. Il faut également faire attention à l'ordre des calculs. Par exemple A + (B - C) peut être différent de (A + B) - C. Si A = 1e-10, B = 1e10, C = 1e10, on a A + (B - C) = A + 0 = 1e-10 mais (A + B) - C = B - C = 0.

À cause des pertes de précision, les flottants sont rarement exactement égaux à leur valeur approchée, surtout si l'on utilise des fonctions comme la racine carrée, le sinus, ...

Prenons par exemple le code suivant :

```
double x = 2.;
x = sqrt(x);  // Racine carrée de x
x = x * x;    // x^2

if (x == 2.)
    printf("OK\n");
```

On s'attend à voir ce programme écrire `OK`. Il n'en est rien ! À cause des pertes de précision, `x` n'est pas égal à 2 à la fin du programme, mais à un nombre presque égal à 2 et différent de 2, comme 2.0000000000000004.

On ne peut donc pas tester l'égalité entre deux nombres réels.

Pour déterminer si deux nombres réels sont égaux, il faut commencer par calculer la différence des deux nombres. Si la différence est très proche de zéro, c'est-à-dire si la valeur absolue de la différence est très petite, alors on considère que les nombres sont égaux.

Concrètement, pour comparer `x` et `y`, on commence par définir une constante très petite que l'on nomme traditionnellement *epsilon*, et on teste si |x - y| < epsilon. La fonction `fabs()` calcule la valeur absolue.

```
#include <math.h>

const double epsilon = 1e-10;

double x = 2.;
x = sqrt(x);  // Racine carrée de x
x = x * x;    // x^2

if (fabs(x - 2.) < epsilon)
    printf("OK\n");

// pour tester si x est égal à zéro, il faut faire :
if (fabs(x) < epsilon)
    printf("x est égal à 0 !\n");
```

