STRUCTURES DE CONTROLE

Déroulement séquentiel d'un programme

Un programme se déroule, jusqu'à présent, de manière totalement séquentielle, et certains problèmes se résolvent très bien de cette manière : reprenons un exemple pour bien se remémorer les points déjà rencontrés.

Enoncé : on veut obtenir le montant des intérêts d'un capital de N euros placé sur un compte avec un taux d'intérêts annuel de T %. On veut ensuite que ces intérêts soient additionnés au capital pour obtenir le capital total. La valeur du capital sera saisie par l'utilisateur, le taux T est de 4,25 %.

Algorithme :
Saisir le capital
Calculer les intérêts : capital * taux d'intérêt
Ajouter les intérêts au capital

Autre solution :
Saisir le capital
Mettre à jour le capital : capital *(1+taux d'intérêt)

```
programme capital_interet

réel cap, taux ,inter;

taux ← 4.25 / 100. // car c'est un pourcentage
saisir(cap);
inter ← cap * taux;  }   Ou : cap ← cap*(1.0+taux);
cap ← cap+inter;     }
afficher("au bout d'un an, le capital sera de :",cap);
```

Cependant, ce déroulement séquentiel n'est pas suffisant pour qu'un programme puisse résoudre des problèmes un peu plus sophistiqués. Un seul comportement ne suffit parfois pas pour traiter différents cas, et il faut alors utiliser des structures de contrôle qui permettent à un programme de ne pas être purement séquentiel. L'exemple le plus simple à traiter pour vous en convaincre est celui de la résolution d'une équation du second degré dans $\mathbb{R}$, donc pour laquelle on n'a pas toujours une solution, ce qui implique qu'un programme devant résoudre ce problème (bien simple) ne pourra pas systématiquement réagir de la même manière.

Alternative et test conditionnel

Essayons donc de traiter ce problème en rappelant la méthode utilisée : cela nous donnera déjà une bonne idée de l'algorithme.

Calcul des racines de l'équation $ax^2+bx+c = 0$

Méthode de résolution :

Calculer $\Delta = b^2-4ac$

Si $\Delta < 0$: pas de solutions dans $\mathbb{R}$

Si $\Delta \geq 0$: une ou deux solutions : $x1 = \frac{-b-\sqrt{\Delta}}{2a}, x2 = \frac{-b+\sqrt{\Delta}}{2a}$

Il y a donc deux possibilités, donc deux comportements différentes à avoir selon la valeur de la quantité Δ calculée, ou, en l'exprimant de manière différente, selon que la condition $\Delta < 0$ est vraie ou fausse. Il est impossible de traiter ce problème avec un programme purement séquentiel !

Il serait intéressant de disposer d'un moyen permettant d'agir selon une condition, et qui permettrait de faire certaines instructions si une condition est réunie (ou vraie), et de ne pas les faire si la condition n'est pas réunie (ou fausse). Fort heureusement, cela est prévu en informatique (et à plus forte raison avec le langage algorithmique et avec le langage C).

Structure si ... alors

La structure de contrôle si…alors est prévue à cet effet en algorithmique, et son fonctionnement est très simple. On l'écrit de la manière suivante :

```
si (condition) alors
     instruction;
```

Cette structure fonctionne de la manière suivante : si la condition est vraie, alors l'instruction est faite; si la condition est fausse alors, l'instruction n'est pas faite. Nous allons maintenant aborder le point suivant : qu'est-ce qu'une condition? Et comment écrire proprement une condition en langage algorithmique ?

Ecriture des conditions : expressions booléennes

Nous avons déjà rencontré le mot 'expression' lors du cours précédent : une expression est une formule calculable et compréhensible. Nous l'avons appliqué aux calculs mathématiques les plus simples : les calculs arithmétiques. Mais il existe, en logique et en

informatique, une autre manière de calculer que l'arithmétique classique : c'est le calcul booléen, pour lequel les valeurs prises par les variables sont : VRAI et FAUX. Une **condition** est donc une expression booléenne, c'est à dire une formule compréhensible et dont le calcul donne comme résultat : VRAI ou FAUX.

Le calcul booléen respecte un certain nombre de règles, qui sont très simples, et que l'on peut comprendre très rapidement (cela revient à résoudre des petits problèmes de logiques). Les opérateurs logiques sont eux aussi très simples à comprendre et à manipuler.

Les opérateurs de comparaison

Un opérateur arithmétique +, *, donne comme résultat un nombre, et non pas VRAI ou FAUX. Il existe pour cela d'autres opérateurs, que l'on peut appliquer pour comparer deux valeurs :

Ces opérateurs sont :

nom	utilisation	rôle	résultat
=	valeur1 = valeur2	égalité	VRAI si les deux valeurs testées sont égales
≠	valeur1≠valeur2	Inégalité	VRAI si les deux valeurs testées sont différentes
>	valeur1 > valeur2	Supérieur strictement	VRAI si valeur1 strictement supérieure à valeur2
<	valeur1 < valeur2	Inférieur strictement	VRAI si valeur1 strictement inférieure à valeur2
≥	valeur1 ≥ valeur2	Supérieur ou égal	VRAI si valeur1 supérieure ou égale à valeur2
≤	valeur1 ≤ valeur2	Inférieur ou égal	VRAI si valeur1 inférieure ou égale à valeur2

Grâce à ces opérateurs, on peut donc obtenir des conditions élémentaires pour faire les tests. Reprenons l'exemple de notre problème de résolution d'équation du second degré : on commence par calculer le discriminant Δ, puis on fait un test suivant une certaine condition pour traiter le cas le plus simple : celui où il n'y a pas de solution. Voici donc ce que doit faire notre programme pour l'instant : afficher un petit texte de présentation, saisir les valeurs des coefficients de l'équation a, b et c, calculer Δ, puis afficher un message dans le cas où il n'y a pas de solutions à l'équation.

```
programme second_degre
réel coef_a, coef_b, coef_c;
réel delta; // attention à l'écrire en toute lettre

afficher("entrez les coefficients de l'equation :");
saisir(coef_a);
saisir(coef_b);
```

```
saisir(coef_c);

delta ← (coef_b*coef_b)-(4.0*coef_a*coef_c);
si (delta < 0.0) alors
     afficher("il n'y a pas de solutions a cette equation\n");
```

Quelques remarques sur ce début de programme :

- Les variables pour stocker les valeurs de a, b, et c (de l'énoncé) ne se nomment pas a, b et c, mais portent un nom plus explicite; cela ne change absolument rien aux calculs !
- Pour le calcul de Δ (que l'on nomme `delta` en toutes lettres, car les lettres de l'alphabet grec ne sont pas autorisées dans les identificateurs de variable), on utilise bien l'écriture **`4.0`** et non **`4`** pour bien préciser qu'il s'agit d'un nombre à virgules (de type `réel`).
- Pour le calcul de b^2, on utilise `coef_b *coef_b`, car l'ordinateur ne comprend pas l'écriture `coef_b²` (vous verrez qu'au niveau stupidité, un ordinateur peut battre tous les records), même si le symbole ² est accessible au clavier.
- Il est important de toujours faire précéder une saisie d'un message expliquant à l'utilisateur du programme la valeur qu'il est censé entrer au clavier.

Examinons maintenant la structure conditionnelle de plus près :

La condition à évaluer est : delta < 0.0, qui sera, selon la valeur de la variable delta, VRAIE ou FAUSSE. Si elle est vraie, l'instruction suivant le test est faite : l'ordinateur affichera :

```
il n'y a pas de solutions a cette equation
```

Si elle est fausse : alors delta est supérieure ou égal à 0.0, et l'instruction d'affichage n'est pas faite.

Ajoutons un autre test qui s'occupe maintenant du cas où il y a des solutions à l'équation : on sait que si la variable delta est ≥ 0.0, alors on doit faire le calcul des solutions (donc on ajoutera deux variables de type réel pour stocker les valeurs des solutions).

Ajoutons au début du programme (après la ligne `réel delta;` par exemple) :

```
réel sol_1, sol_2;
```

et écrivons le test correspondant au cas où il y a des solutions :

```
si (delta >= 0.0) alors
```

il faut maintenant calculer les deux solutions, mais cela nécessite 2 instructions (2 affectations avec les calculs des solution), or on ne peut faire suivre le si … alors que d'une instruction.

Il existe heureusement un moyen de grouper les instructions : les blocs d'instruction : un bloc d'instruction est constitué d'instructions encadrées par des accolades '{' et '}'. Ce bloc est alors considéré par les différentes structures de contrôle, comme une seule instruction.

Ainsi, si l'on écrit :

```
si (condition) alors
{
      instruction1;
      instruction2;
      …     ;
      instruction n;
}
```

c'est tout le bloc qui sera effectué si la condition est vérifiée (ou vraie), et aucune instruction d bloc ne sera faite si la condition n'est pas vérifiée (ou fausse).

La suite du programme est alors :

```
si (delta >= 0.0) alors
{
      sol_1 ← (-1.0 *b – racine(delta))/(2.0 *a);
      sol_2 ← (-1.0*b + racine(delta)) /(2.0 *a);
      afficher("les solutions sont ", sol_1," et ",sol_2);
}
```

pour conclure cette partie : le programme d'un seul tenant, pour qu'il soit bien lisible :

```
programme second_degre
réel coef_a, coef_b, coef_c;
réel delta; // attention a l'écrire en toute lettre
réel sol_1, sol_2;

afficher("entrez les coefficients de l'equation :");
saisir(coef_a);
saisir(coef_b);
saisir(coef_c);

delta ← (coef_b*coef_b)-(4.0*coef_a*coef_c);

si (delta < 0.0) alors
      afficher("il n'y a pas de solutions a cette equation\n");

si (delta >= 0.0) alors
{
      sol_1 ← (-1.0 *b – racine(delta))/(2.0 *a);
      sol_2 ← (-1.0*b + racine(delta)) /(2.0 *a);
      afficher("les solutions sont ", sol_1," et ",sol_2);
}
```

Structure si…alors…sinon

En regardant bien les deux tests effectués dans le programme précédent, on se rend compte qu'ils sont exclusifs : cela signifie que si l'un est vrai, alors l'autre est forcément faux ! delta est soit < 0, soit ≥ 0 !

Le résultat du premier test permet de connaître la résultat du deuxième. Il serait donc intéressant de disposer d'une structure de contrôle un peu plus riche que **si…alors** : c'est la structure **si…alors…sinon**, qui s'emploie de la manière suivante :

```
si (condition) alors
{
     bloc d'intruction 1;
}
sinon
{
     bloc d'instruction 2;
}
```

la partie si…alors est identique à la structure de contrôle simple : si la condition est vraie, alors le bloc d'instruction 1 est fait, et le bloc d'instruction 2 n'est pas fait. Le bloc d'instruction 2, concerné par le **sinon**, est fait si et seulement si la condition est fausse.

Résumé :

- Si la condition est vraie : le bloc d'instruction suivante le **alors** est fait, celui suivant le **sinon** n'est pas fait
- Si la condition est fausse : le bloc d'instruction suivante le **alors** n'est pas fait, celui suivant le **sinon** est fait.

Illustration : un programme déterminant si un nombre entier saisi par l'utilisateur est un nombre pair ou impair. Le programme devra afficher le nombre saisi, puis le message "est un nombre pair" ou "est un nombre impair" selon le cas.

```
programme pair_ou_impair

entier nb;

afficher ("entrez un nombre svp :");
saisir(nb);
// pour savoir si un nombre est pair ou impair, on utilise
// l'opérateur modulo %
```

```
si( nb%2 = 0) alors
{
     afficher(nb," est un nombre pair\n");
}
sinon
{
     afficher(nb," est un nombre impair\n");
}
```

on peut également reprendre le programme de résolution d'équation du second degré :

```
programme second_degre
réel coef_a, coef_b, coef_c;
réel delta; // attention a l'écrire en toute lettre
réel sol_1, sol_2;

afficher("entrez les coefficients de l'equation :");
saisir(coef_a);
saisir(coef_b);
saisir(coef_c);

delta ← (coef_b*coef_b)-(4.0*coef_a*coef_c);

si (delta < 0.0) alors
     afficher("il n'y a pas de solutions a cette equation\n");

sinon   // au lieu de : si (delta >= 0.0) alors
{
     sol_1 ← (-1.0 *b – racine(delta))/(2.0 *a);
     sol_2 ← (-1.0*b + racine(delta)) /(2.0 *a);
     afficher("les solutions sont ", sol_1," et ",sol_2);
}
```

Attention, on ne remplace pas systématiquement deux structures si…alors consécutives par une structure si…alors…sinon, on ne peut le faire que si les deux conditions testées sont exclusives l'une de l'autre, c'est à dire si l'une est vraie lorsque l'autre est fausse.

Structure si…alors…sinon si…

Il se peut qu'un problème ne puisse être résolu par le simple test de deux conditions exclusives l'une de l'autre. C'est le cas par exemple de l'équation du second degré, si le cas spécifique $\Delta = 0$ doit être traité. Ce sont alors 3 cas distincts que l'on doit traiter, les uns après les autres. Le fait de tester la condition $\Delta < 0$ soit vraie permet d'indiquer qu'il n'y a pas de solution réelle à l'équation proposée, mais le fait que cette même condition soit fausse ne permet pas de conclure sur le fait qu'il y ait une seule solution ou deux solutions distinctes.

Il faut dans ce cas refaire un test : si la condition $\Delta < 0$ est fausse, on doit tester si la condition $\Delta > 0$ est vraie, pour bien dissocier le cas où Δ est nul.

Cela est possible grâce à la structure conditionnelle :

```
si (condition1) alors
{
     bloc d'instructions 1;
}
sinon si (condition 2) alors
{
     bloc d'instructions 2;
}
sinon
{
     bloc d'instructions 3;
}

instructions suivantes;
```

Cette structure teste la condition 1. Si celle ci est vraie, le bloc d'instructions 1 est effectué, et l'ordinateur passe aux instructions qui se situent après la structure complète (donc aux `instructions suivantes`).

Si la condition 1 est fausse, la structure teste la condition 2 : si celle ci est vraie, le bloc d'instructions 2 est fait, et l'ordinateur passe aux `instructions suivantes`. Si la condition 2 est fausse (mais cela n'est testé que si la condition 1 était fausse également), l'ordinateur exécute le bloc d'instructions 3, puis passe aux `instructions suivantes`.

Il est possible d'enchaîner ces structures sans limite.

Illustration : achat d'un jouet avec un certain crédit. Noël approchant, vous décidez d'acheter un jouet à votre petit frère (ou petite sœur), histoire de l'éloigner de votre PC. Selon votre crédit, vous pouvez acheter certains jouets :

12 € : poupée Barbo ou camion de pompier sans la sirène

20 € : camion de pompier avec la sirène ou poupée Barbo et sa dînette

32 € : panoplie de spiderman ou panoplie de la poupée Barbo

Ecrivons un programme qui, en fonction du crédit, indique le jouet à acheter.

```
programme jouet

entier credit;

afficher("entrez votre budget en €:");
saisir(credit);

si (credit >= 32) alors
{
     afficher("panoplie : spiderman ou poupee Barbo");
}
```

```
sinon si (credit >= 20) alors
{
     afficher("poupee barbo+dinette ou camion de pompier avec sa
sirene");
}
sinon si (credit >= 12) alors
{
     afficher("camion de pompier sans sirene ou poupee Barbo");
}
sinon
{
     afficher("pas de cadeau cette année");
}
```

Il faut ici faire particulièrement attention à l'ordre dans lequel on écrit les différentes conditions, car toutes ne seront pas systématiquement testées ! Le test d'une condition peut ici dépendre du résultat du test d'une autre condition écrite précédemment.

Les opérateurs booléens

Grâce aux différents opérateurs conditionnels (ou de comparaison) rencontrés jusqu'ici, il est possible de faire des tests simples, mais on peut parfois faire des tests plus sophistiqués (et un peu plus complexe) pour écrire des programmes plus performants et plus faciles à comprendre (et c'est ce dernier point qui est le plus important pour nous). Les règles de calcul avec les conditions, que l'on appelle les règles de l'algèbre booléenne, permettent de réunir plusieurs conditions en une seule :

Imaginons un programme simple qui permet de choisir une activité en fonction du temps qu'il fait et en fonction du taux de remplissage de votre porte-monnaie. Ce programme se base sur un ensemble de choix prédéfinis, dont voici la liste :

S'il fait beau et que le porte-monnaie est rempli : piscine

S'il fait mauvais et que le porte-monnaie est rempli : cinéma

S'il fait beau et que le porte-monnaie est vide : vélo

Il nous faut donc, pour déterminer le programme de la journée, tester deux conditions. Cela nécessite donc deux tests. Voici un algorithme pour déterminer le premier choix (aller à la piscine) :

```
si (il fait beau) alors
{
     si (le porte monnaie est rempli) alors
     {
          aller à la piscine;
     }
}
```

On ne fait le deuxième test (si le porte monnaie est rempli) que si la première condition est vraie. Donc, l'instruction `aller à la piscine` ne sera faite que si les deux tests sont vrais.

On peut répéter cela avec les autres activités proposées, et le programme fonctionnera très bien. Mais que se passerait-il s'il fallait vérifier 2,4,10 conditions avant de pouvoir faire une instruction ? le programme ou l'algorithme deviendrait vite illisible, et inefficace.

Pour cela, on peut plutôt regrouper plusieurs conditions en une seule en utilisant des opérateurs logiques (ou opérateurs booléens). Ces opérateurs permettent de faire des calculs avec des valeurs de type VRAI / FAUX, tout comme les opérateurs arithmétiques permettent de faire des calculs avec des valeurs entières ou à virgule.

Ces opérateurs sont au nombre de 4 :

ET, OU, OUEX, NON

L'opérateur ET

Cet opérateur porte sur deux conditions c1 et c2 et donne un résultat VRAI si les deux conditions sont vraies, c'est à dire si c1 et c2 sont vraies.

Valeur de c1	Valeur de c2	Valeur de (c1 ET c2)
FAUX	FAUX	FAUX
FAUX	VRAI	FAUX
VRAI	FAUX	FAUX
VRAI	VRAI	VRAI

L'opérateur OU

Cet opérateur porte sur deux conditions c1 et c2 et donne un résultat VRAI si l'une des deux conditions est vraie, c'est à dire si c1 ou c2 est vraie.

Valeur de c1	Valeur de c2	Valeur de (c1 OU c2)
FAUX	FAUX	FAUX
FAUX	VRAI	VRAI
VRAI	FAUX	VRAI
VRAI	VRAI	VRAI

L'opérateur OUEX

Cet opérateur porte sur deux conditions c1 et c2 et donne un résultat VRAI si l'une seulement des deux conditions est vraie, c'est à dire si c1 ou c2 est vraie et que l'autre est fausse.

Valeur de c1	Valeur de c2	Valeur de (c1 ET c2)
FAUX	FAUX	FAUX
FAUX	VRAI	VRAI
VRAI	FAUX	VRAI
VRAI	VRAI	FAUX

L'opérateur NON

Il ne porte que sur une condition c, et donne un résultat VRAI si la condition est fausse, et un résultat FAUX si la condition est vraie.

Valeur de c	Valeur de NON c
FAUX	VRAI
VRAI	FAUX

Bien écrire des conditions et des opérateurs logiques

Comme pour les opérateurs arithmétiques, il vaut mieux utiliser des parenthèses de manière systématique pour chacune des conditions, et autour de chaque groupe de conditions.

Pour tester si la condition c1 est vraie ou la condition c2 est vraie, on écrira :

```
si ((c1) OU (c2))
```

Cette technique permet d'obtenir un programme plus lisible, et de ne pas avoir à connaître par cœur les priorités des opérateurs.

Application : calcul de réduction pour un voyage en avion.

La société Mancha Air propose, sur la liaison Roissy – Orly, un vol à 99,99 € par passager. Cependant, si le vol décolle avant 7h00 **et** concerne un groupe de plus de 60 passagers, une réduction de 2 % est accordée. Il n'y a pas d'autres réductions possibles.

Ecrire un programme qui, en fonction de l'heure de départ et du nombre de passagers, donne le montant total à payer pour le vol.

```
programme mancha_air

entier heure, nb_pass;
réel prix, reduc, remise;
prix ← 99.99;
```

```
reduc ← 0.02; // ne pas écrire 2/100 !

afficher("entrez le nombre de passagers :");
saisir(nb_pass);
afficher("heure de decollage ? ");
saisir(heure);

prix ← prix * nb_pass;

si ((nb_pass >= 60) ET (heure <= 7)) alors
{
     remise ← prix * reduc;
     prix ← prix - remise;
}

afficher("le prix total est :", prix, "\n");
afficher("merci d'avoir choisi Mancha Air");
```

Relation VRAI/FAUX et nombres entiers

Traduction de VRAI et FAUX en valeurs entières :

Tout comme les caractères, les valeurs VRAI et FAUX sont représentées par l'ordinateur par des nombres entiers, car c'est ce qu'il sait manipuler le plus facilement. Ainsi, la valeur FAUX est associée à l'entier 0 :

Essayons cette instruction qui peut sembler bizarre :

```
afficher(1=2);
```

1=2 est un test, qui donne la valeur FAUX, que l'ordinateur stocke par la valeur 0.

Cette instruction fonctionne très bien et donne :

```
0
```

Lorsque l'ordinateur fait un test et que le résultat de ce test est VRAI, alors, il stocke cette valeur par un 1:

```
afficher(-1 ≠ 6);
```

donne

```
1
```

Car –1 ≠ 6 est une condition qui donne VRAI.

Traduction d'une valeur entière vers VRAI/FAUX :

Une valeur entière peut aussi être traduite en une valeur VRAI/FAUX selon la règle suivante :

- 0 est compris comme FAUX
- toute valeur non nulle est comprise comme VRAI

on peut donc écrire les instructions suivantes :

```
programme entier_vrai

si (4) alors
{
     afficher("condition vraie\n");
}
sinon
{
     afficher("condition fausse\n");
}
```

ce qui donnera :

```
condition vraie
```

Car la valeur 4 a été comprise comme VRAI

Autre test :

```
programme entier_faux
entier t;
t ← 0;
si (t) alors
{
     afficher("condition verifiee");
}
sinon
{
     afficher ("condition non verifiee");
}
```

donnera :

```
condition non verifiee
```

car t vaut 0, et 0 est compris comme FAUX pour le test. Donc, ce sont les instructions (ici, une seule) concernées par le **sinon** qui sont faites.

Structure selon…cas (structure de sélection)

Principe : cette structure est une simplification de la structure si…alors sinon si pour des cas particuliers. Elle permet d'associer un bloc d'instructions à une valeur entière que peut prendre une variable ou expression. Cette structure s'écrit de la manière suivante :

```
selon (variable ou expression entière)
{
      cas valeur1 : bloc d'instructions 1;
      cas valeur2 : bloc d'instructions 2;
      …
      cas valeurn : bloc d'instruction n;
      par defaut : bloc d'instruction n+1;
}
```

où valeur1, valeur2, valeur*n*, sont des valeurs entières que peut prendre la variable ou l'expression testée. Si la variable a une des valeurs listées, alors c'est le bloc d'instruction associé à cette valeur qui sera effectué, et seulement ce bloc; si la variable n'a aucune des valeurs listées, c'est le bloc d'instruction associé à la valeur `par defaut` (le bloc d'instructions n+1) qui sera effectué.

La variable où l'expression ne peut pas être de type réel !

Une utilisation classique de cette structure est la création de menus pour guider l'utilisateur. Nous allons prendre un autre exemple pour montrer le fonctionnement :

On vous demande de sélectionner un item (un élément dans une liste) en saisissant son numéro. On affiche préalablement la liste avec les numéros associés, et on saisit le numéro. C'est à partir de ce numéro que l'on fait la sélection.

Exemple de programme utilisant la structure de sélection :

Vous devez choisir un équipement dans une liste, chaque équipement est caractérisé par son prix et par sa puissance en W. On veut récupérer, dans deux variables `prix` et `puissance`, les valeurs associées à l'équipement choisi.

La liste des équipements vous est donnée :

Equipement	Prix (€ TTC)	Puissance (W)
Séchoir Mincèche	220	2600
Aspirateur Tornadeau	125	1500
Lave Linge Vendetta	699	1200
Four Polbokuz	930	2300

Pour choisir un équipement, on peut saisir son nom, mais la moindre faute de frappe ou d'orthographe se révèle vite insurmontable pour un ordinateur (pensez donc, plusieurs

caractères à la suite, c'est bien trop compliqué à traiter pour une machine !), aussi on choisit de matérialiser le choix par la sélection d'un numéro, qui sera bien plus aisé à gérer informatiquement.

L'utilisateur saisira le numéro de l'équipement, et le programme affichera les caractéristiques de l'appareil sélectionné

```
programme select_equip
entier numero;
entier prix, puissance;

afficher("selectionnez votre equipement parmi :\n");
afficher("1 . Sechoir Minceche\n 2 . Aspirateur Tornadeau \n3 . Lave
Linge Vendetta\n 4 . Four Polbocuz\n");

saisir(numero);

selon (numero)
{
     cas 1 :    prix ← 220;
                puissance ← 2600;

     cas 2 :    prix ← 125;
                puissance ← 1500;

     cas 3 :    prix ← 699;
                puissance ← 1200;

     cas 4 :    prix ← 930;
                puissance ← 2300;

     par defaut :    prix ← 0;
                     puissance ← 0;
}

si (prix ≠ 0) alors
{
     afficher("caracteristiques de l'appareil :\n");
     afficher("prix : ",prix," € , puissance :",puissance," W\n");
}
sinon
{
     afficher("choix incorrect");
}
```

Répétitions et boucles

Il existe d'autres structures de contrôle que la structure conditionnelle (si…alors…sinon, ou selon…cas), car certains problèmes ne peuvent se résoudre uniquement avec cette première structure. Parfois, on doit répéter des actions ou instructions plusieurs fois avant d'obtenir le résultat voulu.

Prenons l'exemple d'une suite $(U_n)_{n\in\mathbb{N}}$ (au sens mathématique), où l'on a besoin de connaître un terme U_n pour calculer U_{n+1}. Une telle suite peut par exemple être définie par :

$$\begin{cases} U_0 = 4; \\ U_{n+1} = \dfrac{Un}{2} - \dfrac{3}{Un}; \end{cases}$$

On cherche à faire un programme qui saisit un entier (nommé rang par exemple) et qui permette d'obtenir la valeur de la suite U au terme dont le numéro est égal au rang demandé : si l'utilisateur entre 8, on veut la valeur de U_8, s'il rentre 30, la valeur de U_{30}, etc…

Sans structure appropriée, il est délicat de faire le programme : essayons tout de même pour bien se rendre compte de la difficulté à laquelle on se trouve confronté dans ce cas :

```
programme calcul_de_suite

entier rang;
réel terme_u;   // cette variable contiendra les valeurs successives
du terme de la suite

afficher("entrez la valeur du rang pour le calcul du terme :");
saisir(rang);

terme _u ← 4.0;

terme_U ← (terme_U / 2.0) - (3.0 / terme_U);
// terme_U contenait U0, il contient maintenant U1
terme_U ← (terme_U / 2.0) - (3.0 / terme_U);
terme_U ← (terme_U / 2.0) - (3.0 / terme_U);
```

Mais quand s'arrêter ? on peut penser à une structure selon…cas, puisque le rang est une entier, mais le programme risquerait d'être très long !

Connaissant le rang, et la formule permettant de passer d'un rang au suivant, on pourrait plutôt répéter (ou **réitérer**) le calcul tant que le rang n'a pas été atteint.

Pour programmer un accès par mot de passe, on peut aussi demander à l'utilisateur de saisir des caractères, et si le mot saisi n'est pas correct, l'utilisateur doit recommencer.

Cependant, une structure conditionnelle ne suffit pas, car il faut recommencer la saisie et le test !

Il existe une structure de contrôle permettant de répéter une instruction ou un bloc d'instructions tant qu'une condition est vérifiée.

La boucle tant que

Ecriture et fonctionnement

Ecriture :

```
tant que (condition)
      instruction;

instructions suivantes;
```

Ou

```
tant que (condition)
{
      instruction 1;
      …
      instruction n;
}

instructions suivantes;
```

Fonctionnement :

lorsque l'ordinateur rencontre cette structure, il procède systématiquement de la manière suivante :

1) la condition est testée (on dit aussi évaluée).
2) Si la condition est fausse, l'instruction ou les instructions du bloc ne sont pas faites et on passe aux instructions suivantes (après la structure de contrôle)
3) Si la condition est vraie, l'instruction ou les instructions du bloc sont faites, et on recommence à l'étape 1) : test de la condition

Une illustration avec le calcul de la suite $U_{n+1} = \frac{Un}{2} - \frac{3}{Un}$

Le rang *n* pour lequel on veut le résultat sera stocké dans une variable entière nommé `rang`. On pourra par exemple faire saisir cette valeur par l'utilisateur.

Il faudra compter le nombre de fois où l'on fait le calcul : pour cela, on utilisera une variable entière nommé `compteur`. Cette variable sera augmentée de 1 à chaque fois que l'on fera le calcul du terme U_{n+1} à partir du terme U_n.

Le calcul à effectuer pour passer de U_n à U_{n+1} est le suivant :

```
terme_U ← (terme_U / 2.0) - (3.0 / terme_U);
```

rappel : l'ordinateur calcule l'expression situé à droite de l'opérateur d'affectation, puis range le résultat dans la variable situé à gauche.

Il faudra donc effectuer ce calcul jusqu'à ce que `compteur` soit égal à `rang`, ce que l'on peut dire également : tant que `compteur` est inférieur à `rang`.

Exemple de programme :

```
programme calcul_suite

entier compteur, rang;
réel terme_U;

compteur ← 0;
terme_U ← 4.0;
afficher("entrez le rang auquel vous voulez obtenir Un :");
saisir(rang);

tant que (compteur < rang)
{
     terme_U ← (terme_U / 2.0) - (3.0 / terme_U); // effectue le
calcul du terme suivant
     compteur ← compteur + 1; // indique que l'on passe au terme
suivant
}
```

Le principe du : faire 0 fois ou plus

Cette boucle ou répétition tant que est parfois appelée boule 0 fois ou plus, car les instructions concernées par la condition peuvent être faites 0 fois (si la condition est fausse lors du premier test), ou plus si la condition est vraie lors du premier test.

Autre exemple traité : saisie d'un mois. On demande de faire un petit programme qui vérifie si des dates entrées au format classique *jj*/*mm*/*aa* sont valides, et il faut notamment contrôler que le nombre *mm*, correspondant au numéro du mois, soit bien compris entre 1 et 12. L'instruction saisir ne permet pas de vérifier si la valeur qui a été saisie répond à ce genre de conditions. Il est donc possible que l'utilisateur puisse se tromper une première fois sur la

valeur du mois, voire même une deuxième ou troisième fois s'il est borné (ou s'il a confondu avec la valeur du jour, ce qui peut arrivé s'il s'agit d'un logiciel anglais mal adapté, car les valeurs de *jj* et *mm* sont inversées).

Il faudra donc probablement répéter la saisie un certain nombre de fois…mais jusqu'à quand ?

On arrêtera de proposer la saisie lorsque la valeur demandée sera correcte, donc comprise entre 1 et 12. Ce qui peut également être écrit sous la forme : on continuera la saisie tant que la valeur demandée sera incorrecte.

```
programme saisie_valide

entier mois_saisi; // contiendra la valeur du mois saisi
entier correct;    // vaut 0 si la saisie est incorrecte, 1 sinon

correct ← 0; // pour l'instant pas de saisie correcte

tant que (correct = 0)
{
      afficher("entrez le mois (mm) :");
      saisir(mois_saisi);
      si ((mois_saisi > 0) ET (mois_saisi <= 12)) alors
      {
            correct ← 1;
      }
}
```

si la valeur du mois saisi n'est pas valide, la variable `correct` reste égale à 0 et la condition `(correct=0)` est vraie, le bloc d'instruction de saisie sera donc de nouveau effectué.

La boucle faire … tant que

Cette structure de contrôle est très proche de la boucle ou répétition tant que. La seule différence réside dans l'ordre dans lequel sont faites les tests et les instructions. Cette structure s'utilise de la manière suivante :

```
faire
      instruction;
tant que (condition);

instructions suivantes;
```

ou

```
faire
{
      instruction 1;
      intruction2;
      …
      instruction n;
}
tant que (condition);

instructions suivantes;
```

Fonctionnement :

lorsque l'ordinateur rencontre cette structure, il procède systématiquement de la manière suivante :

1) Exécution de l'instruction ou du bloc d'instruction concerné
2) la condition est testée (on dit aussi évaluée).
3) Si la condition est vraie, l'instruction ou les instructions du bloc sont faites, et on recommence à l'étape 1) : exécution de l'instruction ou des instructions du bloc. Si la condition est fausse, le programme passe aux instructions suivantes.

On effectuera donc au minimum une fois l'instruction ou les instructions du bloc, puisque la condition de boucle est testée après ces instructions. Par opposition à la boucle tant que, que l'on qualifie de boucle 0 fois ou plus, la boucle faire…tant que est qualifiée de : 1 fois ou plus.

Cette boucle n'est pas un gadget par rapport à la boucle tant..que, car elle permet d'écrire des programmes plus courts à certaines occasions. Reprenons le programme de saisie de mois que nous avons vu précédemment avec la boucle tant…que. Nous devions employer une variable nommé correct qu'il fallait initialiser avant de tester la condition. Cette variable devient inutile avec la boucle faire…tant que. En voici l'illustration :

```
programme saisie_valide

entier mois_saisi; // contiendra la valeur du mois saisi

faire
{
     afficher("saisissez le mois (mm) :");
     saisir(mois_saisi);
}
tant que ((mois_saisi<1) OU (mois_saisi >12));
```

la différence tient à ce que la condition est testée après les instructions : on dispose donc de la valeur du mois saisie avant de faire le test : le test peut donc être fait directement avec la variable `mois_saisi`, qui a une valeur entrée par l'utilisateur, ce qui n'était pas le cas avec la boucle tant que. D'autre part, la condition n'est pas la même : dans la boucle tant que, la variable correct était initialisée à 1 si le mois était valide, donc compris entre 1 et 12.

Pour la boucle faire … tant que, on doit indiquer la condition à laquelle on recommence la saisie : cette condition est que la valeur n'est pas valide, c'est à dire qu'elle est inférieure à 1 OU supérieure à 12 (attention à bien utiliser les opérateurs logiques à l'écriture d'une condition composée).

L'exemple de contrôle de saisie est l'exemple type de l'emploi de cette boucle : vous aurez à l'employer lors des TPs et projets pour empêcher l'utilisateur d'entrer n'importe quoi au clavier.

La boucle pour

Il existe enfin un dernier type de répétition que l'on emploie lorsque l'on connaît à l'avance le nombre de fois où une instruction (ou un bloc d'instructions) doit être fait. Les boucles tant que et faire…tant que ne sont pas prévues à cet usage. Il s'agit de la boucle pour, dont l'usage principal est de faire la gestion d'un compteur (un entier) qui évolue d'une valeur à une autre.

En réalité, la boucle pour est un peu plus sophistiquée, et se rapproche de la boucle tant que, mais nous l'utiliserons pour l'instant d'une manière simplifiée.

Ecriture et fonctionnement

```
pour variable de valeur1 à valeur2
     instruction

instructions suivantes;
```

Ou

```
pour variable de valeur1 à valeur2
{
      bloc d'instructions;
}

instructions suivantes;
```

fonctionnement :

1) la variable, jouant le rôle de compteur, est initialisée à la valeur1
2) l'ordinateur teste si la variable est inférieure ou égale à la valeur2 :
   - si c'est le cas, l'instruction ou le bloc d'instruction est effectué, la variable jouant le rôle de compteur est augmentée de 1, et retour à l'étape 2), et non à l'étape 1) qui initialise la variable;
   - si ce n'est pas le cas, l'instruction ou le bloc d'instruction n'est pas effectué, et l'ordinateur passe aux `instructions suivantes`.

Le principe du : faire n fois

D'après le fonctionnement, on sait par avance le nombre de fois où l'instruction (ou les instructions) seront faites, car cela ne dépend que de valeur1 et valeur2.

valeur1 et valeur2 sont des valeurs entières qui peuvent être données par : des constantes, des variables, ou plus généralement par des expressions entières.

Exemples d'utilisation de la boucle pour :

Saisir 5 valeurs à virgule, et afficher leur valeur absolue juste après la saisie :

```
programme cinq_saisies_affich

reel var_reel;
entier compteur;

pour compteur de 1 à 5 faire
{
      afficher ("saisir une valeur a virgule :");
      saisir(var_reel);
      si(var_reel <0) alors
      {
            var_reel ← -1. * var_reel;
      }

      afficher("valeur absolue :",var_reel,"\n");
}
```

on effectue donc 5 fois le bloc d'instruction concerné par la boucle pour. La valeur de la variable compteur ne joue aucun rôle dans le bloc d'instructions, mais on peut améliorer simplement le programme de manière à bien faire apparaître son évolution :

```
programme cinq_saisies_affich

reel var_reel;
entier compteur;

pour compteur de 1 à 5 faire
{
      afficher("saisie numero ",compteur,"\n");
      afficher ("saisir une valeur a virgule :");
      saisir(var_reel);
      si(var_reel <0) alors
      {
            var_reel ← -1. * var_reel;
      }
      afficher("resultat numero ",compteur,"\n");
      afficher("valeur absolue :",var_reel,"\n");
}
```

le résultat de ce programme apparaît sous la forme :

```
saisie numero 1
saisir une valeur a virgule :-1.E-7
resultat numero 1
valeur absolue : 1e-007
saisie numero 2
saisir une valeur a virgule :6.12E+23
resultat numero 2
valeur absolue : 6.12e+023
saisie numero 3
saisir une valeur a virgule :-1.732
resultat numero 3
valeur absolue : 1.732
saisie numero 4
saisir une valeur a virgule :6
resultat numero 4
valeur absolue : 6
saisie numero 5
saisir une valeur a virgule :-120000000
resultat numero 5
valeur absolue : 1.2e+008
```

exemple : valeur2 donné par une variable entière.

On demande maintenant à l'utilisateur combien de valeurs il veut traiter.

L'infâme aller à (goto)

Les langages de programmation dits de 3ème génération (comme le langage C) ont été créés il y a une trentaine d'années, lorsque les technologies de l'informatique n'étaient pas aussi avancées qu'aujourd'hui. Il y avait d'autres contraintes à gérer, notamment au niveau de l'efficacité et de la taille des programmes, puisque la mémoire disponible excédait à peine les quelques ko. Il existe donc des instructions qui n'ont plus de raison d'être utilisées aujourd'hui, car elles ne sont plus nécessaires, et sont potentiellement nuisibles à la compréhension et à l'analyse d'un programme. L'une d'entre elles est l'instruction aller à (traduite dans de nombreux langages par goto) qui permet de passer directement et sans conditions d'une instruction d'un programme à une autre. Le problème avait par ailleurs déjà été soulevé en … 1968 !

Pourquoi dans ce cas en parler ? tout simplement, parce que vous risquez un jour de rencontrer cette instruction en lisant un ouvrage de référence ou en consultant un site internet relatif à l'algorithmique ou en langage C. Cette instruction n'offre qu'un confort relatif. Plutôt que de l'employer en désespoir de cause, réfléchissez plutôt à la structure de votre programme : si l'emploi de aller à reste la seule solution, c'est simplement parce que la structure que vous employez n'est pas la bonne. Il est possible de réaliser tous les algorithmes sans employer cette instruction (c'est mathématiquement prouvé).

C'est donc la première et la dernière fois que nous rencontrerons cette instruction.