Structure de contrôle

En programmation informatique, une **structure de contrôle** est une instruction particulière d'un langage de programmation impératif pouvant dévier le flot de contrôle du programme la contenant lorsqu'elle est exécutée. Si, au plus bas niveau, l'éventail se limite généralement aux branchements et aux appels de sous-programme, les langages structurées offrent des constructions plus élaborées comme les alternatives (*if*, *if –else*, *switch*, ...), les boucles (*while*, *do–while*, *for*, ...) ou encore les appels de fonction. Outre ces structures usuelles, la large palette des structures de contrôle s'étend des constructions de gestion d'exceptions (*try –catch*, ...) fréquemment trouvés dans les langages de haut niveau aux particularismes de certains langages comme les instructions différées (*defer*) de Go.

Sommaire

Structures de contrôle séquentielles

Un programme informatique impératif est une suite d'instructions. Un registre interne du processeur, le compteur ordinal (PC), est chargé de mémoriser l'adresse de la prochaine instruction à exécuter.

Séquence

La plupart des instructions d'un programme sont exécutées séquentiellement : après le traitement de l'instruction courante le compteur ordinal est incrémenté, et la prochaine instruction est chargée.

La séquence est donc la structure de contrôle implicite. Elle donne l'ordre d'exécution des instructions, souvent séparées par un point-virgule ou par des retours chariots.

Arrêt du programme

Un programme s'arrête généralement après l'exécution de la dernière instruction. La plupart des langages de programmation proposent également une ou plusieurs instructions pour stopper l'exécution du programme à une position arbitraire.

Selon l'environnement d'exécution sous-jacent (système d'exploitation ou microprocesseur), cet arrêt peut être définitif ou correspondre à une suspension de l'exécution du programme en attendant un événement externe : c'est par exemple le fonctionnement habituel de la plupart des instructions d'entrée sorties qui bloquent le flot d'exécution (mécanisme d'interruption avec stockage en mémoire tampon) jusqu'à ce que le périphérique concerné ait terminé de traiter les données.

Structures de contrôle itératives

Ces instructions permettent de réaliser une machine à états finis, cela signifie que leur seul effet de bord est de modifier un registre qui correspond à l'état courant du programme.

Dans un processeur, cet état correspond à la valeur du compteur ordinal.

Commandes à étiquettes

Une étiquette est un nombre ou un identificateur associé à une instruction du code source. Il est destiné à servir de cible à une structure de contrôle située ailleurs dans le programme. Hormis ce rôle de localisation, une étiquette n'a aucun effet sur le programme : elle ne modifie pas le comportement de l'instruction à laquelle elle est associée.

Les numéros de lignes sont un type d'étiquettes utilisées par des langages comme le Fortran ou le BASIC. Présentes devant les lignes du code source, elles doivent dans la plupart de ces langages augmenter au fur et à mesure du programme mais sans être forcément contigües. Par exemple en BASIC :

```
    10 X = 3
    20 PRINT X
```

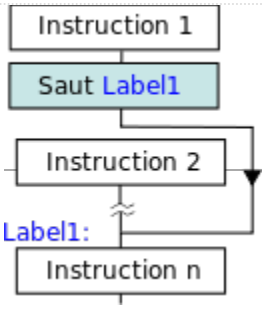
Dans d'autres langages, les étiquettes sont des identificateurs alphanumériques, qui se placent généralement en début de ligne et qui sont suivis par un double point. Par exemple en langage C :

```
success:
  printf ("The operation was successful.\n");
```

Il existe deux familles d'instructions qui permettent d'adresser ces étiquettes : les sauts inconditionnels et les sauts conditionnels.

Sauts inconditionnels

Un saut inconditionnel, souvent appelé goto, permet de renvoyer l'exécution vers une étiquette. Ce saut est systématique, il entraine une rupture du flot d'exécution, on dit aussi « rupture de séquence » ou branchement. L'instruction qui suit le saut dans le programme ne pourra donc elle-même être atteinte que par un autre saut.




```
Instruction 1
Saut Label1
Instruction 2
...
Label1:
Instruction n
```

Exemples de mise en œuvre :

- **BASIC** : GOTO *etiquette*
- **Asm 86** : JMP *etiquette*

Sauts conditionnels

Les sauts conditionnels permettent de réaliser un branchement si une condition est vérifiée. Si la condition n'est pas vérifiée, l'exécution se poursuit séquentiellement. La condition est parfois appelée « condition de rupture » puisqu'elle implique en général une rupture dans le flot d'exécution lorsque la condition est vérifiée.

Dans un langage de haut niveau, cette condition est le résultat de l'évaluation d'une expression. En langage assembleur, c'est un indicateur : le bit d'un registre spécifique qui peut être modifié par d'autres instructions du langage.

Exemples de mise en œuvre :

- **BASIC** : IF *condition* GOTO *etiquette*
- **Asm 86** : JZ *etiquette*

Autres commandes à étiquettes

D'autres structures de contrôle qui seront présentées plus loin utilisent également la notion d'étiquette.

C'est par exemple le cas des appels de sous-programmes ou des commandes de sortie de boucle.

Commandes de blocs

Formellement, le saut conditionnel est la seule instruction nécessaire pour réaliser une machine à états finis. Si la plupart des processeurs supportent également, pour des raisons de performances, le saut inconditionnel, ce sont souvent les seules utilisées à ce niveau.

Les créateurs de langages de programmation ont cependant rapidement estimé que les étiquettes étaient peu adaptées aux mécanismes cognitifs de l'être humain. Souvent qualifiés de programmation spaghetti, les programmes utilisant de nombreuses instructions de saut sont difficiles à comprendre et donc à maintenir par les programmeurs. Ceux-ci peinent en effet à en percevoir la sémantique et sont obligés de dérouler mentalement les différents états de ce programme en le parcourant d'étiquette en étiquette.

Sont donc apparus de nouveaux jeux d'instructions de contrôle qui ne travaillent plus avec des étiquettes, mais sur des blocs d'instructions. D'un langage à l'autre, un grand nombre de commandes de blocs ont été inventées, mais on peut globalement les classer en deux grandes familles :

- les alternatives : exécuter un bloc d'instructions si une certaine condition est réunie.
- les boucles : exécuter un bloc d'instructions à plusieurs reprises.

Une telle démarche relève de la programmation structurée.

Blocs d'instructions

Un bloc d'instruction regroupe plusieurs instructions contigües. Si l'on considère la façon de déclarer ces blocs, il existe deux grandes classes de langages :

Dans la première famille, les instructions de contrôle sont composées de deux mots-clef : un mot initial qui marque le début et un mot final qui indique la fin du bloc d'instructions ; la forme du mot final varie d'un langage à l'autre :

- Ada : le mot final est end + espace + mot initial ex : if ... end if, loop ... end loop
- Algol 68, bash : mot initial en miroir : ex : if ... fi, case ... esac
- Fortran 77 : end + mot initial ex : IF ... ENDIF, DO ... ENDDO
- Modula-2 : le même mot final end pour toutes les structures de contrôle.
- Turbo Basic : chaque structure de contrôle à son propre mot final : if ... end if; for ... next; while ... wend

Dans la seconde famille de langages, les instructions de contrôle opèrent par défaut sur une instruction atomique. Le langage définit un autre mécanisme pour déclarer des blocs qui peuvent être utilisés avec ces instructions :

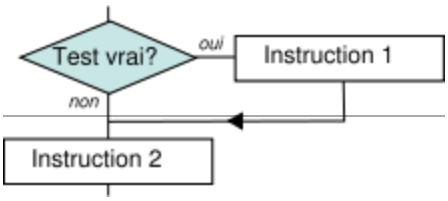
- des mots-clefs - Algol 60 et Pascal : begin ... end ; PL/1 : DO ... END
- la ponctuation - C, C++, Java, Perl, et PHP : accolades { ... } ; BCPL : $(... $)
- la mise en page - Haskell et Python : indentation comme syntaxe

Alternatives

Les alternatives sont des structures de programmation effectuant un test logique sur une condition et permettant un choix entre divers blocs d'instructions suivant le résultat de ce test. La condition est en général appelée « condition de continuation » car le flot d'exécution continue avec le bloc d'instruction qui suit immédiatement la condition lorsque celle-ci est vraie.

Test si

Le test *si* est la forme d'alternative la plus simple : si *Test* est vérifié, on exécute *Instruction 1* puis *Instruction 2* ; si *Test* n'est pas vérifié, on exécute directement *Instruction 2*



Pseudo code :

```
SI Test
     Instruction 1
FIN SI
Instruction 2
```

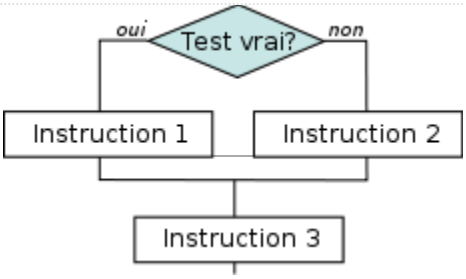
Exemples de mise en œuvre :

- **BASIC** et **Pascal** : IF *condition* THEN *instruction*
- **C** : if (*condition*) *instruction*
- **Python** : if *condition* : *instruction*
- **Forth** : *condition* IF *instruction* THEN

Le mot-clef *if* correspond à *si* en anglais ; le mot-clef *then* correspond à *alors*.

Test si sinon

```
SI Test
   Instruction 1
SINON
   Instruction 2
FIN SI
Instruction 3
```



Si *Test* est vérifié, on exécute *Instruction 1* puis *Instruction 3* ; sinon on exécute *Instruction 2* puis *Instruction 3*.

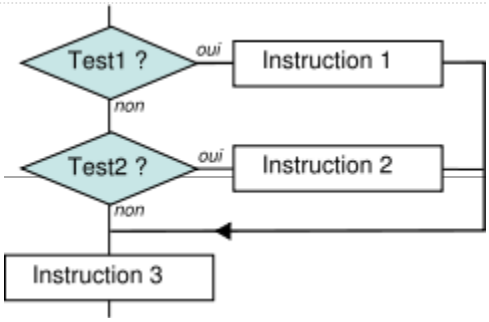
Exemples de mise en œuvre :

- **Pascal** : if *condition* then *instruction* else *instruction*
- **C** : if (*condition*) *instruction* else *instruction*

Le mot-clef *else* correspond à *sinon* en anglais. Historiquement, on rencontre un premier *if then else* dans Algol 60.

Test sinon si

```
SI Test 1
   Instruction 1
SINONSI Test 2
   Instruction 2
FIN SI
Instruction 3
```



Si *Test 1* est vérifié, on exécute *Instruction 1* puis *Instruction 3* ; sinon, si *Test 2* est vérifié on exécute *Instruction 2* puis *Instruction 3* ; sinon on exécute directement *Instruction 3*.

On peut enchaîner autant d'instructions *sinon si* que désiré : seule la première dont la condition sera vérifiée sera exécutée. On peut généralement associer une clause *sinon* qui ne sera exécutée que si aucune clause *sinon si* n'a été vérifiée.

Dans les langages où la définition des blocs d'instruction est indépendante des structures de contrôle, cette instruction est redondante puisqu'elle revient à enchaîner un *si* après un *sinon*. Par exemple en langage C on écrira :

```
if ( test1 )
        Instruction1
else if ( test2 )
        Instruction2
Instruction3
```

Certains de ces langages définissent tout de même une instruction *sinon si* (par exemple *elseif* en PHP), pour des raisons de performance par exemple.

Test selon

Le test *selon* est une spécialisation de l'instruction *sinon si*, qui permet de sélectionner le bloc à exécuter en fonction de la valeur d'une variable. Il est utilisé lorsqu'un aiguillage offre plusieurs sorties, et que l'on doit tester une condition plusieurs fois, en utilisant toujours la même variable.

```
SELON Variable 1
CAS Valeur 1:
        Instruction 1
CAS Valeur 2:
        Instruction 2
FIN SELON
Instruction 3
```

Structurellement, c'est équivalent à une succession de *sinon si*, mais le fait de savoir que la valeur de la variable testée ne changera pas lors de l'évaluation des conditions permet au compilateur de faire quelques optimisations.

Suivant les langages, l'expression qui suit le mot-clef *cas* peut être une simple valeur pour un test d'égalité, une collection, un intervalle, ou une expression rationnelle par exemple.

Le *switch case* du langage C s'écarte sensiblement du modèle ci-dessus. Son comportement *fall through* (passage à travers) des case non terminé par le mot-clef *break*, le rapproche plus d'une structure à étiquette.

Le switch case apparaît dans Algol 68.

Boucles

Une boucle est une structure de contrôle destinée à exécuter une portion de code plusieurs fois de suite, la structure de contrôle branchant le pointeur ordinal au début du code tant qu'une condition de continuation est remplie ou, selon les boucles, qu'une condition de sortie n'est pas remplie.

Normalement, une boucle s'exécute selon le cas, soit un nombre de fois connu à l'avance, soit jusqu'à ce qu'une condition permette de sortir de la boucle. Il arrive toutefois qu'une erreur de programmation fasse que la condition ne devienne jamais vraie. Le programme s'exécute alors indéfiniment à l'intérieur de cette boucle infinie.

Fortran II a introduit les boucles en 1958.

Les langages proposent en général plusieurs types de boucles :

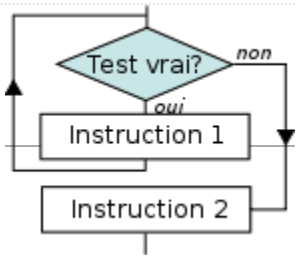
- boucle à pré-condition : la condition est vérifiée avant la première boucle
- boucle à post-condition : la condition est vérifiée après la première boucle
- boucle à condition d'arrêt : la condition est vérifiée au milieu de la boucle
- boucle itérative : un compteur est utilisé pour compter le nombre d'itérations
- boucle de parcours : la boucle est exécutée sur chacun des éléments d'une liste

Les langages peuvent proposer plusieurs types de boucles à pré et post-condition :

- boucle « tant que », qui est exécutée tant que la condition est vraie (en précondition : « while test ; do quelquechose; done ». En postcondition : « do quelquechose; while test »)
- boucle « jusqu'à (ce que) », qui est exécutée jusqu'à ce que la condition soit vraie (pré condition « until test ; do quelquechose; done ». En postcondition, « do quelquechose ; until test »)

Boucle « tant que » à précondition

```
TANTQUE Test
     Instruction 1
FIN TANTQUE
Instruction 2
```



Si *Test* est vérifié, on exécute *Instruction 1*, puis arrivé à la fin du bloc on évalue à nouveau *Test* et on recommence. Quand *Test* renvoie un résultat faux, on quitte la boucle en sautant à *instruction 2*. Le *Test* est en général appelé « condition de continuation » puisque le flot d'exécution continue avec *Instruction 1* lorsque le *Test* est vrai.

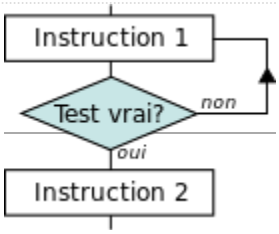
Exemples de mise en œuvre :

- **C** : while (*condition_de_continuation*) *instruction*
- **Pascal** : while *condition_de_continuation* do *instruction*

Le mot-clef *while* en anglais correspond à *tant que* en français.

Boucle « jusqu'à ce que » à postcondition

```
REPETE
     Instruction 1
JUSQUACEQUE Condition 1
Instruction 2
```



Cette boucle permet de réitérer une instruction ou un ensemble d'instructions jusqu'à ce qu'une condition de sortie soit vérifiée. La série d'instructions est exécutée au moins une fois, quelle que soit la condition.

Exemples de mise en œuvre :

- **Pascal** : repeat *instruction* until *condition_d_arret*
- **Perl** : do { *instruction* } until (*condition_d_arret*);

Le mot-clef *until* correspond au français *jusqu'à ce que*, *repeat* à *répète*, *do* à *fait*.

Commandes de sortie de boucle

Ces commandes s'utilisent au milieu d'une boucle pour réaliser une sortie anticipée de la boucle, généralement quand une certaine condition est vérifiée.

Ce sont des instructions comme le **`break`** du langage C.

En Ada la commande **`exit`**, qui est équivalente au `break`, accepte un identificateur, ce qui permet de l'utiliser dans des boucles imbriquées. Ce langage définit aussi une instruction **exit when** qui permet d'associer une condition.

Le langage C définit aussi une instruction **`continue`** qui force le pointeur de commande à sauter à la fin de la boucle. Si la condition d'arrêt de la boucle n'est pas encore vérifiée, la boucle entreprend une nouvelle itération, sinon le programme sort de la boucle et passe à la ligne qui la suit.

Les implémentations actuelles du langage Pascal comportent les procédures `Break` et `Continue`[1].

Toutes ces commandes sont une forme de saut inconditionnel : si elles n'utilisent pas d'étiquettes formelles, elles permettent en effet de modifier la valeur du pointeur ordinal pour adresser un endroit spécifique du programme.

Le langage Python ou Ruby permet d'associer une clause `else` à une boucle, pour effectuer un traitement personnalisé en cas de sortie normale d'une boucle.

Certains langages permettent de casser plusieurs boucles en une seule instruction, ou de nommer les boucles pour pouvoir choisir jusqu'à quelle boucle le programmeur veulent sortir des boucles imbriqués. L'absence d'une telle possibilité est souvent contournée par l'utilisation d'une procédure et d'un "return" au milieu des boucles imbriquées (Ruby et Python ne savent pas sortir de plusieurs boucles en une seule instruction, sauf au moyen de "return").

Extensions de la notion de boucles

Il existe d'autres variantes de boucle, leur comportement est identique à celle d'une boucle *tant que*, mais elles nécessitent des variables additionnelles au registre PC pour mémoriser leur état.

Compteurs

Un compteur permet de réaliser une boucle associée à une variable entière ou un pointeur qui sera incrémentée à chaque itération. Il est souvent utilisé pour exploiter les données d'une collection indexée.

Itérateurs

```
POURCHAQUE valeur DANS collection
   Instruction 1
FIN POURCHAQUE
Instruction 2
```

Un itérateur (ou curseur ou énumérateur) est un objet qui permet de réaliser une boucle parcourant tous les éléments contenus dans une structure de données.

Des langages comme le C++ ou l'Objective C (dans la version initiale du langage) n'ont pas de mots-clefs spécifiques pour les itérateurs, ils utilisent des méthodes de bibliothèque associées à des structures de contrôles plus générales.

exemple en Objective C :

```
NSEnumerator* enumerator = [collection objectEnumerator];
NSObject* obj;
while(obj = [enumerator nextObject]){
        NSLog(@"%@", obj);
}
```

Dans d'autres langages, ce concept bénéficie de structures de contrôle dédiées. Exemples de mise en œuvre :

- **Python** : for *valeur* in *collection* : *instruction*
- **Perl** : foreach *$valeur* (*collection*) *instruction* ;

Sous-programmes

Dans les années 1950, la mémoire des ordinateurs était onéreuse et les sous-programmes étaient principalement utilisés pour réduire la taille des programmes : un ensemble d'instructions était écrit une fois et pouvait être appelé depuis plusieurs endroits du programme.

Le support des sous-programmes a également permis aux ordinateurs l'usage des algorithmes récursifs.

Exemple de sous-programme en basic

```
10  X = 2
19  FARG = X
20  GOSUB 50
22  Y = FRET
30  PRINT Y
40  END
48  '-----------------------------------
49  ' Double la valeur passée en argument
50  FRET = FARG*2
60  RETURN
```

Dans cet exemple, à la ligne 20 la valeur courante du compteur ordinal est archivée puis le programme effectue un saut à la ligne 50. Arrivé à la ligne 60, l'instruction RETURN permet de réaffecter la valeur archivée dans le compteur ordinal et donc de revenir à la position d'appel du sous-programme.

Fonctionnement

Ces structures de contrôle nécessitent donc un automate à pile pour enregistrer l'adresse de retour du sous-programme.

Repose sur la notion de pile, avec deux instructions :

- CALL [X] : empile la valeur Y du registre PC et effectue un saut à l'adresse X.
- RETURN : dépile la valeur Y et effectue un saut à cette adresse.

Procédures

Aujourd'hui les sous-programmes sont utilisés pour améliorer la structuration d'un programme. Repérés par une étiquette de branchement, ils peuvent donc être considérés comme une extension du saut inconditionnel et, du point de vue de la programmation structurée, en partagent la plupart des défauts.

Beaucoup de langages modernes ne supportent donc pas directement la notion de sous-programme au profit de constructions de haut niveau qui peuvent être appelées, d'un langage à l'autre *procédure*, *fonction*, *méthode*, ou *routine* par exemple. Ces constructions ajoutent la notion de passage de paramètres et surtout le cloisonnement des espaces de nom pour éviter que le sous-programme ait un effet de bord sur la routine appelante.

Extensions

Il existe diverses extensions à la notion de procédure comme les coroutines, signaux et slots, fonctions de

rappel (callback), méthodes virtuelles...

Elles permettent de modifier dynamiquement, c'est-à-dire à l'exécution, la structure du flot d'exécution du programme.

Exceptions

Tout programme en exécution peut être sujet à des erreurs pour lesquelles des stratégies de détection et de réparation sont possibles. Ces erreurs ne sont pas des bugs mais des conditions particulières (ou conditions exceptionnelles, ou exceptions) dans le déroulement normal d'une partie d'un programme.

Par exemple, l'absence d'un fichier utile n'est pas un bug du programme ; par contre, ne pas gérer son absence en serait un.

Quand on n'a pas d'outil pour séparer l'exécution normale et l'exécution exceptionnelle du programme, un algorithme, dont l'exécution normale s'exprime de façon simple et élégante, peut devenir difficile à maintenir une fois « enrobé » par une logique de traitement de ces situations exceptionnelles ; disposer au niveau du langage de programmation de structures pour différencier l'exécution normale de l'exécution dans un contexte exceptionnel peut être utile. Ces structures de contrôles forment un système de gestion d'exceptions.

Exemples de mise en œuvre :

- **Python** : try : *Instruction* except *Exception* : *Handler*

Programmation multitâche

Dans un système multitâche, plusieurs flots d'exécutions, appelés processus légers, s'exécutent simultanément.

Il est alors nécessaire d'assurer la synchronisation de ces flots d'exécution. Dans la plupart des langages, c'est fait par des bibliothèques externes ; certains d'entre eux intègrent néanmoins des structures de contrôle permettent d'agir sur des taches concourantes.

Exemples de mise en œuvre :

- **Ada** : accept *Rendezvous* do *Instruction* end *Rendezvous* ;

Programmation événementielle

La programmation événementielle est une autre façon de contrôler le flot d'exécutions d'un programme. Il s'agit de créer des gestionnaires qui viendront s'abonner à une boucle mère, chargée d'aiguiller les évènements qui affectent le logiciel.

Cette méthodologie a été popularisée par les bibliothèques de programmation des interfaces utilisateurs. Elle est en effet particulièrement adaptée pour gérer les mouvements de souris par exemple. On retrouve le même principe dans des outils non graphiques comme awk ou XSLT par exemple. À un niveau plus proche du matériel, on retrouve une philosophie similaire avec les interruptions.

Notes et références

1. Guide de référence de Free Pascal : procédures Break (http://www.freepascal.org/docs-html/rtl/system/break.html) et Continue (http://www.freepascal.org/docs-html/rtl/system/continue.html).

Annexes

Articles connexes

- Algorithmique
- Structure de données
- Boucle infinie
- Instruction nulle
- Instructions de la base impérative
- Style de programmation

Sur les autres projets Wikimedia :

Structure de contrôle, sur Wikibooks