LANGAGE C – FICHE 2
Structures de contrôle

Booléen

Une variable booléenne est une variable ne pouvant prendre que deux valeurs : true ou flase (vrai ou faux).

En C, le type booléen ne fait pas parti des types de base, ce qui conduit souvent à considérer les entiers 0 et 1 comme équivalant à false et true respectivement.

Pour pouvoir utiliser les variables booléennes en C, il faut importer la bibliothèque stdbool. Il sera donc interdit (cours, TD, TP, DS, colle et examen) d'utiliser les valeurs entières 0 et 1 à la place des variables booléennes.

Les programmes utilisant les variables booléennes devront donc être précédés de la commande de préprocesseur : # include <stdbool.h>

Opérateurs

Pour les comparaisons, les opérateurs utilisés sont les suivants :

Symbole	Signification
==	Egalité
<	Strictement inférieur
<=	Inférieur ou égal

Symbole	Signification
!=	Différent de
>	Strictement supérieur
>=	Supérieur ou égal

Les opérateurs logiques utilisés sont les suivant :

Symbole	Signification	Dénomination	Table de vérité
&&	ET logique	Conjonction	a / b / a && b false / false / false false / true / false true / false / false true / true / true
|| (alt gr+ touche 6)	OU logique	Disjonction	a / b / a || b false / false / false false / true / true true / false / true true / true / true
!	NON	Négation	a / !a false / true true / false

Evaluation paresseuse des expressions logiques

L'évaluation d'une expression logique correspondant à une combinaison d'opérateurs se fait en considérant les opérations successives : la première expression est évaluée ; La seconde

ne le sera que si le résultat e la première opération est concluant (opérateur ET appliqué tant que l'on n'obtient pas de résultat FAUX : opérateur OU appliqué tant que l'on obtient pas de résultat VRAI).

Exemples :

Expression	Comportement lors de l'évaluation
1 && 1 && 0 && 1 && 1	
1 && 1	Le premier opérateur ET est appliqué et l'on obtient le résultat VRAI. Test non concluant donc le logiciel poursuit l'évaluation
1 && 1 && 0	Le résultat est FAUX. L'évaluation s'arrête car, quels que soient les résultats des opérations suivantes, le résultat du test restera FALSE. La fin de l'expression ne sera donc pas évaluée.

0 || 1 || 0 || 0 || 1	
0 || 1	Dès la première évaluation, le résultat est VRAI. Il est inutile de poursuivre et de réaliser les tests suivants.

Remarque importante concernant l'évaluation paresseuse :

Quant on écrit une opération logique, les termes susceptibles de provoquer des erreurs doivent être placés après le test d'erreur. Il s'agit d'une des techniques de programmation défensive visant à fiabiliser les codes.

Exemple :

On souhaite tester si 10/x est supérieur à 1. Pour cela, il faut que x soit non nul. Les deux équations logiques suivantes peuvent être envisagées du point de vue mathématiques ; mais elle ne donneront pas le même résultat en informatique :

- 10/x > 1 && x != 0 : si x = 0, l'exécution peut provoquer une erreur fatale si x = 0 (division par 0)
- x != 0 && 10/x > 1 : avec l'évaluation paresseuse, la partie 10/x > 1 ne sera évaluée que si x est non nul.

Structures de test

<u>if / else if / else</u>

`if (conditions)`
`{`
`  Instructions ;`
`}`
`else if (conditions)`
`{`
`  Instructions ;`
`}`
`else`
`{`
`  Instructions ;`
`}`
	Condition associée au if :
	Equation logique
	Evaluée de gauche à droite
	Evaluation paresseuse à prendre en considération
	Pas de ; à la fin de la ligne commençant par if, else if ou else
	
	Else if et else sont facultatifs
	
	Les blocs d'instructions à réaliser sont délimités par des accolades. Elles peuvent néanmoins être omises si le bloc d'instructions ne contient qu'une seule instruction.

Switch

`switch (variable)`
`{`
`  case valeur1 :`
`  {`
`  Instructions ;`
`  break ;`
`  }`
`  case valeur2 :`
`  {`
`  Instructions ;`
`  break ;`
`  }`
`  ...`
`  default :`
`  {`
`  Instructions ;`
`  }`
`}`
	L'instruction switch permet de présenter de manière plus compacte les tests imbriqués.
	
	La variable spécifiée va successivement être comparée à différentes valeurs, chaque valeur étant associée à un comportement particulier du programme.
	
	Chaque bloc d'instructions se termine par l'instruction break (sortie de structure).
	
	L'instruction default précède le bloc d'instructions à exécuter si aucun des cas n'est vérifié.
	
	Default est facultatif.

Structures de répétition

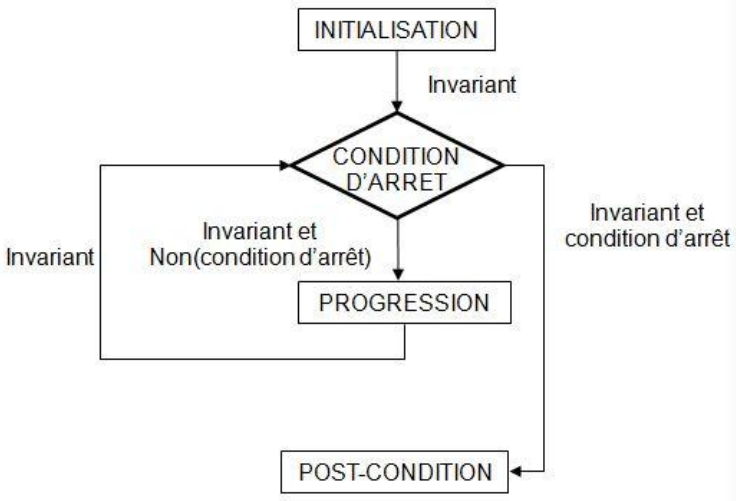
Lorsque des instructions sont amenées à être exécutées un certains nombre de fois, on les place au sein de boucles.

Les boucles sont constituées des 4 éléments suivants :

- le bloc d'instructions qui sera exécuté à plusieurs reprises ;
- une condition, impliquant une variable de boucle, qui permet de contrôler le nombre d'itérations à effectuer. Il peut y avoir plusieurs variables de boucles ;
- une initialisation, qui concerne la variable (ou les variables de boucles). Elle peut être réalisée par le programmeur ou par l'instruction de répétition ;
- une modification de la variable de boucle, qui peut être réalisée par le programmeur ou par l'instruction de répétition.

Si un de ces éléments est absent, la boucle est incorrecte.
Le principe de la boucle est présenté ci-contre.

Il fera l'objet d'une étude détaillée dans un chapitre ultérieur.



Boucle for

Contexte d'utilisation : répétition d'un bloc d'instructions lorsque l'on connait a priori le nombre d'itérations.

Particularité : initialisation et modification de la variable de boucle gérées par l'instruction.

Syntaxe :

`for (initialisation ; fin ; modification)`
`{`
`  Instructions ;`
`}`
	Pas de ; à la fin de la ligne du for
	
	Gestion de la variable de boucle :
	• entre parenthèses
	• initialisation de la variable : possibilité de déclarer in situ
	• fin : utilisation des opérateurs de comparaison
	• modification : incrémentation ou décrémentation. Possibilité d'utiliser les opérateurs particuliers :
	  ○ i++ équivalent à i = i+1 (ou i += 1)
	  ○ i— équivalent à i = i-1 (ou i -= 1)
	
	Délimitation du bloc d'instruction par des accolades.
	
	Possibilité de plusieurs variables de boucles

Exemple

```
for (int i=0 ; i<10 ; i++)
{
      printf ("%d", i);
}
```

```
// Cas de deux variables de boucle
// les deux variables évoluent simultanément

for (int i=0, int j=10 ; i<j ; i+=2, j--)
{
      printf ("%d, %d", i, j);

}
```

Boucle while

Contexte d'utilisation : répétition d'un bloc d'instructions lorsque l'on ne connait pas a priori le nombre d'itérations et que l'on ignore si il doit être exécuté au moins une fois.

Particularité : initialisation et modification de la variable de boucle gérées par le programmeur.

Syntaxe :

`Initialisation`
`while (condition)`
`{`
`  Instructions ;`
`}`
	Pas de ; en fin de la ligne du while
	
	Condition entre parenthèses
	
	Bloc d'instructions délimité par les accolades

	Initialisation de la variable de boucle avant le while Condition du while doit porter sur la variable de boucle Risque de boucle infinie su la variable de boucle n'évolue pas dans le bloc d'instructions Si la condition est fausse dès le premier test, le bloc d'instruction n'est pas exécuté : une condition fausse entraîne la sortie de la boucle

Boucle do

Contexte d'utilisation : répétition d'un bloc d'instructions lorsque l'on ne connait pas a priori le nombre d'itérations et que l'on sait qu'il doit être exécuté au moins une fois.

Particularité : initialisation et modification de la variable de boucle gérées par le programmeur.

Syntaxe :

Initialisation do { *Instructions* ; } while (*condition*) ;	Pas de ; sur la ligne du do Condition entre parenthèses Bloc d'instructions délimité par des accolades. Le bloc d'instructions est exécuté au moins une fois, même si la condition est fausse, puisque la vérification de la condition se fait après son exécution