INF 2005 Programmation orientée objet avec C++

Texte 2

Version du 12 janvier 2015

1. Mots clés du langage C++ et premiers pas

1.1 Les mots clés du langage C++

À l'instar des autres langages de programmation, la programmation C++ possède des mots clés propres à la structure du langage C++. À part les mots clés courant du langage C, C++ possède des mots clés supplémentaires. Le tableau 1 illustre ces mots clés.

Tableau 1 Les mots clés de C++ n'existant pas en C

bool	catch	class	const_cast
delete	dynamic_cast	explicit	export
false	friend	inline	mutable
namespace	new	operator	private
protected	public	reinterpret_cast	static_cast
template	this	true	throw
try	typeid	typename	using
virtual			

Le tableau 2 présente la liste complète des mots clés de C++. Certains existent déjà en C, mais certains sont spécifiquement propres à C++.

Tableau 2 Les mots clés de C++

asm	auto	bool	break	case
catch	char	class	const	const_cast
continue	default	delete	do	double
dynamic_cast	else	enum	explicit	export
extern	false	float	for	friend
goto	if	inline	int	long
mutable	namespace	new	operator	private
protected	public	register	reinterpret_cast	return
short	signed	sizeof	static	static_cast
struct	switch	template	this	throw true
true	try	typedef	typeid	typename
union	unsigned	using	virtual	void
volatile	wchar_t	while		

1.2 Familiarisation avec le langage

Nous nous familiariserons avec le langage C++ en examinant quelques exemples de programmation, soit quelques opérations élémentaires. Ces opérations vont de l'affichage d'un texte jusqu'aux processus d'affectation, d'incrémentation et de décrémentation.

Abordons le langage de programmation C++ en étudiant le programme qui permettra l'affichage à l'écran du texte suivant :

```
Bienvenue au cours Programmer en C++
```

Examinons chaque ligne de code de ce programme :

```
// Mon premier programme en C++
#include <iostream>
using namespace std;
main()
{
   cout << "Bienvenue au cours Programmer en C++"<< endl;
   return 0; // Indique que le programme se termine avec succès
}
```

Dans le but d'améliorer la lisibilité et la compréhension d'un programme en C++, des commentaires sont nécessaires et ils sont très simples à produire. Ainsi, la première ligne du programme commence par deux barres obliques `//`; ces deux barres indiquent au compilateur que ce qui va suivre est un commentaire et, par conséquent, ne génère pas de code machine. Bien que tout programme doive être commenté, il faut éviter de placer des commentaires partout, car le programme peut devenir illisible; il faut les limiter aux éléments importants.

D'autre part, la ligne :

```
#include <iostream> ou #include <iostream.h>
```

indique au préprocesseur d'inclure le contenu du fichier `iostream.h`, ainsi que tout programme dont les entrées proviennent du clavier et (ou) les sorties se font à l'écran.

Par ailleurs, la ligne :

```
#main()
```

indique le point d'entrée de tout programme en C++; `main()` est une fonction, la première à être exécutée.

Cette fonction est suivie d'une accolade ouvrante « { » qui indique le début de la fonction; une accolade fermante }, placée plus loin, signifie la fin de la fonction. Autrement dit, le corps d'une fonction commence par une accolade ouvrante et se termine par une accolade fermante. Nous pouvons également dire qu'à toute accolade ouvrante correspond une accolade fermante.

Quant à la ligne :

```
cout << "Bienvenue au cours programmer en C++\n";
```

elle contient la commande `cout <<` qui permet d'afficher à l'écran `Bienvenue au cours Programmer en C++`. Notons que toute chaîne de caractères à afficher à l'écran est toujours placée entre deux guillemets `" "`. Par ailleurs, `\n` indique le passage à la ligne suivante, tandis que `;` indique la fin de la ligne.

À la même ligne précédente :

```
cout << "Bienvenue au cours programmer en C++ " << endl;
```

on ajoute la commande `<< endl` (abréviation de *end of line*) qui force toutes les sorties à être affichées à l'écran au moment de l'exécution du programme.

La ligne :

```
return 0; // Indique que le programme se termine avec succès
```

contient deux éléments. Le premier est l'expression `return 0`, qui est normalement incluse dans chaque fonction `main()` d'un programme. Il s'agit de l'une des méthodes pour sortir d'une fonction; l'omission de cette ligne n'entraîne pas un message d'erreur, mais un avertissement. Le deuxième élément de la ligne est un commentaire car, comme nous l'avons vu, il y a présence d'une double barre oblique `//`.

2. Les entrées-sorties

Contrairement aux entrées-sorties de C (`printf` et `scanf`), les entrées-sorties de C++ reposent sur la notion de *flot* (classes particulières), ce qui permet notamment de leur donner un sens pour les types définis par l'utilisateur que sont les classes (grâce au mécanisme de surdéfinition d'opérateur).

Les entrées-sorties présentées ci-dessous se limitent à l'aspect conversationnel, c'est-à-dire :

- la lecture des données sur l'entrée standard;
- l'écriture des données sur la sortie standard.

De façon générale, l'entrée standard et la sortie standard correspondent respectivement au clavier et à l'écran. Retenez également que, comme les redirections sur Linux, la plupart des environnements de programmation permettent aussi d'effectuer une redirection de ces unités vers des fichiers, mais cet aspect reste alors totalement transparent au programme.

Les opérateurs `<<` et `>>` figurent dans le fichier d'en-tête `<isotream>`, à inclure comme en-tête à votre programme et les symboles correspondants sont définis dans l'espace de nom `std`.

2.1 L'affichage à l'écran

Voici une procédure d'affichage à l'écran :

```
cout << "Bonjour Monsieur X\n";
cout << "Je vais vous calculer " << NFOIS << " racines carrées
du nombre\n";
```

Dans l'exemple ci-dessus, `<<` représente un opérateur qui permet d'envoyer de l'information sur le flot `cout`, correspondant à l'écran. Plus précisément, on voit que cet opérateur dispose de deux opérandes :

- l'opérande de gauche correspond à un flot de sortie susceptible de recevoir des données;
- l'opérande de droite correspond à une expression.

Soit les instructions suivantes :

```
int n = 30 ;
cout << "valeur : " ;
cout << n ;
```

Le résultat affiché est :

```
Valeur : 25
```

Dans l'exemple ci-dessus, nous avons utilisé l'opérateur `<<` sur le flot `cout` à deux reprises et de manière différente :

- *Pour envoyer une information de type chaîne (constante*). Ici l'opérateur transmet les caractères de la chaîne.
- *Puis pour envoyer une information de type entier*. Ici l'opérateur procède à un formatage pour convertir la valeur binaire soumise en une suite de caractères.

Dans les deux cas, le rôle de l'opérateur `<<` est différent.

Une autre possibilité serait d'utiliser une forme condensée de ces instructions. Ainsi, notre exemple pourra être condensé comme suit :

```
cout << "valeur :" << n ;
```

2.2 La lecture au clavier

L'opérateur `>>` permet de lire l'information sur le flot `cin` correspondant au clavier. Il dispose, comme l'opérateur `<<`, de deux opérandes :

- l'opérande de gauche correspond à un flot d'entrée susceptible de fournir de l'information;
- l'opérande de droite correspond à une valeur.

L'opérateur >> peut jouer plusieurs fonctions :

- Il peut être utilisé pour différents types d'informations. Par exemple :

```
int n ;
char c ;
.....
cin << n ;  // lit une suite de caractères représentant
            // un entier, la convertit en variable de type
            // int et range le résultat dans n
cin << c ; // lit un caractère et le range dans c
```

- Il fournit un résultat.

- Il possède aussi une associativité de gauche à droite. Grâce à ces deux dernières propriétés, la notation `cin >> n >> p` est équivalent à `(cin >> n) >> p`.

2.3 Exemple

Voici un exemple d'utilisation des fonctions entrées-sorties :

```
#include <iostream> // pour Linux pour Windows ajoutez le .h
using namespace std ;
main()
{
  char nom [20], prenom [20], ville [25] ;
  cout << "Quelle est votre ville actuelle : " ;
  cin >> ville ;
  cout << "Donnez votre nom et votre prénom : " ;
  cin >> nom >> prenom ;
  cout << "Bonjour cher " << prenom << " "<< nom << " qui habite
   à " << ville ;
}
```

Voici le résultat de l'exécution :

```
Quelle est votre ville actuelle : Montréal
Donnez votre nom et votre prénom : Dupont Yves
Bonjour cher Yves Dupont qui habite à Montréal
```

3. Opérateurs et expressions

Le langage C++ est certainement l'un des langages les plus fournis en opérateurs. Cette richesse se manifeste tout d'abord pour ce qui est des opérateurs classiques (arithmétiques, relationnels, logiques) ou moins classiques (manipulations de bits). Mais, de surcroît, C++ dispose d'un important éventail d'opérateurs originaux d'affectation.

Ce dernier aspect nécessite une explication. En effet, dans la plupart des langages, on trouve, comme en langage C++ :

- des expressions formées (entre autres) à l'aide d'opérateurs;
- des instructions pouvant éventuellement faire intervenir des expressions, comme l'instruction d'affectation suivante :

```
y = a * x + b ;
```

ou encore comme l'instruction d'affichage suivante :

```
cout << "valeur = " << n + 2 * p ;
```

dans laquelle apparaît l'expression `n + 2 * p ;`

Mais, généralement, dans les langages autres que C++ ou C, l'expression possède une valeur mais ne réalise aucune action, en particulier aucune attribution d'une valeur à une variable.

En langage C++, il en va différemment. D'une part, les (nouveaux) opérateurs d'incrémentation pourront non seulement intervenir au sein d'une expression (laquelle, au bout du compte, possèdera une valeur), mais également agir sur le contenu de variables.

Ainsi, l'expression `++i` (car, comme nous le verrons, il s'agit bien d'une expression en C++) réalisera une action, à savoir augmenter la valeur de `i` de `1`; en même temps, elle aura une valeur, à savoir celle de `i` après l'incrémentation.

D'autre part, une affectation apparemment classique telle que `i = 5` pourra, à son tour, être considérée comme une expression (ici, de valeur 5).

D'ailleurs, en C++, l'affectation `=` est un opérateur. Par exemple, la notation suivante `k = i = 5` représente une expression en C++. Elle sera interprétée comme `k = (i = 5)`. Autrement dit, elle affectera à `i` la valeur `5`, puis elle affectera à `k` la valeur de l'expression `i = 5`, c'est-à-dire `5`.

En langage C++, les notions d'expression et d'instruction sont étroitement liées puisque la principale instruction de ce langage est une expression terminée par un point-virgule. On la nomme souvent « instruction expression ».

Voici des exemples de telles instructions qui reprennent les expressions :

```
++i ;
i = 5 ;
k = i = 5 ;
```

Les deux premières ont l'allure d'une affectation telle qu'on la rencontre classiquement dans la plupart des autres langages. Notez que, dans ces deux cas, il y a l'évaluation d'une expression (`++i` ou `i = 5`) dont la valeur est finalement inutilisée. Dans le dernier cas, la valeur de l'expression `i = 5`, c'est-à-dire `5`, est à son tour affectée à `k`; par contre, la valeur finale de l'expression complète est, là encore, inutilisée.

C++ offre un jeu très étendu d'opérateurs, ce qui permet d'écrire une grande variété d'expressions. Un principe général est que toute expression retourne une valeur. On peut donc utiliser le résultat de l'évaluation d'une expression comme une partie d'une autre expression. De plus, la parenthèse permet de forcer l'ordre d'évaluation.

Voici les opérateurs qui seront utilisés dans ce texte et dans les suivants.

OPÉRATEURS ARITHMÉTIQUES	
+	addition
-	soustraction
*****	multiplication
/	division (entière ou réelle)
%	modulo (sur les entiers)
OPÉRATEURS RELATIONNELS	
> **> =** **< =** **<**	comparaisons
= = **! =**	égalité et inégalité
!	négation (opérateur unaire)
& &	ET relationnel
| |	OU relationnel

Enfin, C++ vous permet d'écrire ce que l'on nomme des « expressions mixtes » dans lesquelles interviennent des opérandes de types différents. Voici un exemple d'expression autorisée, dans laquelle `n` et `p` sont de type `int`, tandis que *x* est de type `float` :

```
n * x + p
```

Dans ce cas, le compilateur sait, compte tenu des règles de priorité, qu'il doit d'abord effectuer le produit `n*x`. Pour que cela soit possible, il va mettre en place des instructions de conversion de la valeur de `n` dans le type `float` (car on considère que ce type `float` permet de représenter à peu près convenablement une valeur entière, l'inverse étant naturellement faux). Au bout du compte, la multiplication portera sur deux opérandes de type `float` et elle fournira un résultat de type `float`.

4. Opérateurs logiques

En programmation C++, les opérateurs logiques permettent de former des conditions complexes en combinant plusieurs opérateurs simples qui sont reliés soit par le « ET » logique, le « OU » logique ou par le « NON » logique.

4.1 Le ET logique

L'opérateur logique ET, représenté par `&&` en programmation C++, permet de relier plusieurs conditions et de s'assurer que ces conditions sont vraies avant d'effectuer une opération donnée.

Si, en Amérique du Nord, pour jouer au football il faut mesurer au moins 170 cm et peser au moins 75 kg, ces conditions sont traduites en programmation C++ par :

```
if (taille> = 170 && poids> = 75)
cout << "Vous pouvez jouer au football américain." << endl;
```

Le `if` contient deux conditions relatives à la taille et au poids. D'abord, la condition `taille> = 170` est évaluée pour déterminer si un individu mesure au moins 170 cm. Ensuite, la condition `poids> = 75` détermine si ce même individu pèse au moins 75 kg. Si les deux conditions sont vraies, la déclaration `if` considère la combinaison des conditions aussi vraie. Le code situé à l'intérieur de la structure `if` s'exécute alors.

4.2 Le OU logique

Le OU logique permet de vérifier si une seule condition est vraie parmi un ensemble de conditions avant d'effectuer une opération donnée. Le OU logique en programmation C++ est représenté par l'opérateur `||`. Voyons un exemple.

Pour jouer au football américain, il faut mesurer au moins 170 cm ou peser au moins 75 kg. Ces conditions sont traduites en programmation C++ de la façon suivante :

```
if (taille > = 170 || poids >= 75)
cout << "Vous pouvez jouer au football américain" << endl;
```

L'exemple comporte deux conditions. L'une des deux (ou les deux) doit être vraie pour que la combinaison des deux soit vraie et que le message apparaisse à l'écran.

4.3 Le NON logique

C++ fournit l'opérateur NON logique, représenté par le symbole point d'exclamation `!`, pour permettre à un programmeur d'inverser le sens d'une condition. Contrairement aux symboles `&&` et `||` qui exécutent des opérations binaires, le NON logique est un opérateur unaire, c'est-à-dire qu'il ne dispose qu'un seul opérande.

Le NON Logique est utilisé lorsque la condition doit être fausse pour exécuter le reste du programme. Voyons un exemple d'une utilisation simple du NON logique.

```
if (!(note == -1))
cout << "La prochaine note est" << note << endl;
```

Dans cet exemple, la paire de parenthèses utilisées dans `(note == -1)` est nécessaire. Sinon, le NON logique s'exécute avant l'égalité, ce qui ne correspond pas à ce que l'on veut exprimer comme condition.

Le tableau 3 illustre l'ordre de priorité dans lequel s'exécutent les principales opérations arithmétiques.

Tableau 3 Ordre de priorité des opérateurs arithmétiques

Opérateur	Associativité	Genre d'opération
`( )`	De gauche à droite	Parenthèses
`++ -- + - !`	De gauche à droite	Primaire
`* / %`	De gauche à droite	Multiplicatif
`+ -`	De gauche à droite	Additif
`<< >>`	De gauche à droite	Insertion/extraction
`< <= > >=`	De gauche à droite	Relationnel
`== !=`	De gauche à droite	Égalité
`&&`	De gauche à droite	Logique ET
`||`	De gauche à droite	Logique OU
`?:`	De droite à gauche	Conditionnel
`= += -= *= /= %=`	De droite à gauche	Affectation
`,`	De gauche à droite	Virgule

Pour éviter toute confusion dans l'écriture du code, il peut être avantageux d'éviter le NON logique; nous pouvons tout simplement fournir une équivalence en récrivant le code ainsi :

```
if( note != -1)
cout << "La prochaine note est" << note << endl;
```

5. Structures de contrôle

Les structures de contrôle sont des éléments indispensables à la programmation. Comme leur nom l'indique, ces structures permettent de contrôler le déroulement de l'exécution d'un programme en fonction des contraintes que le programme doit respecter. Le langage de programmation C++ dispose d'opérateurs de comparaison, de structures conditionnelles, de structures de répétition, de commandes de rupture de séquence, ainsi que des opérateurs logiques. Étudions ces différentes structures.

5.1 Les opérateurs de comparaison

Les opérateurs de comparaison sont des opérateurs de prise de décision, qui permettent de vérifier si une condition est vraie ou fausse. Les opérateurs de comparaison utilisés en C++ sont les opérateurs d'égalité = ou ≠ et les opérateurs relationnels <, ≤ , > et ≥. Le tableau 4 présente ces opérateurs, ainsi que des exemples d'application en C++.

Tableau 4 Opérateurs de prise de décision

Opérateur	Correspondance en C++	Exemple en C++	Signification
=	==	*x == y*	*x est égale à y*
≠	!=	*x! = y*	*x est différente de y*
≤	<=	*x< = y*	*x est inférieure ou égale à y*
<	<	*x<y*	*x est inférieure à y*
≥	>=	*x> = y*	*x est supérieure ou égale à y*
>	>	*x>y*	*x est supérieure à y*

5.2 Les structures conditionnelles

Les structures conditionnelles sont des structures de sélection qui permettent de choisir une action à effectuer parmi plusieurs. En programmation C++, il y a trois structures

conditionnelles : le `if`, le `if/else` et le `switch`. Voyons chacune de ces structures en étudiant des exemples.

5.2.1 La structure de sélection `if`

La structure conditionnelle `if` consiste à faire un test, puis à exécuter ou non une instruction selon le résultat obtenu après le test.

La syntaxe est la suivante :

```
if (test) opération;
```

où « *test* » est une expression dont la valeur est booléenne ou entière. Toute valeur non nulle est considérée comme vraie. Si le test est vrai, l'opération est exécutée; cette opération peut être une instruction ou un bloc d'instructions.

Voyons un exemple. Pour réussir le cours *Programmer avec C++*, il faut obtenir une note égale ou plus grande que 50.

La structure conditionnelle se présente de la manière suivante :

```
if (note > = 50)
  cout << "Cours réussi";
```

Le diagramme suivant illustre bien l'exemple précédent.

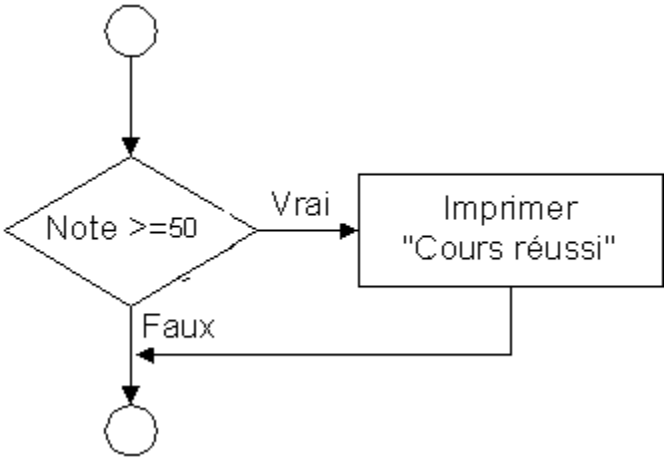

Figure 1 Diagramme de contrôle de la structure `if`.

5.2.2 La structure conditionnelle `if/else`

Cette structure permet au programmeur de choisir l'exécution d'une action parmi plusieurs, selon une certaine condition. Autrement dit, lorsqu'une condition est vérifiée,

une action *X* est déclenchée. Dans le cas contraire, une action *Y* est déclenchée. La structure conditionnelle `if/else` est en somme une variante permettant de spécifier l'action à exécuter quand le test de la structure conditionnelle `if` échoue.

La syntaxe prend la forme suivante :

```
if (test) opération1;
else opération2;
```

L'exemple précédent pourrait être repris en affichant `Cours réussi` si un étudiant obtient une note égale ou supérieure à `50`, et `Cours échoué` dans le cas contraire.

Voyons d'abord le programme de l'implémentation de la structure de contrôle, puis le diagramme de la figure 2 qui illustre comment le programme serait alors contrôlé.

```
if (note > = 50)
   cout << "Cours réussi";
else
   cout << "Cours échoué";
```

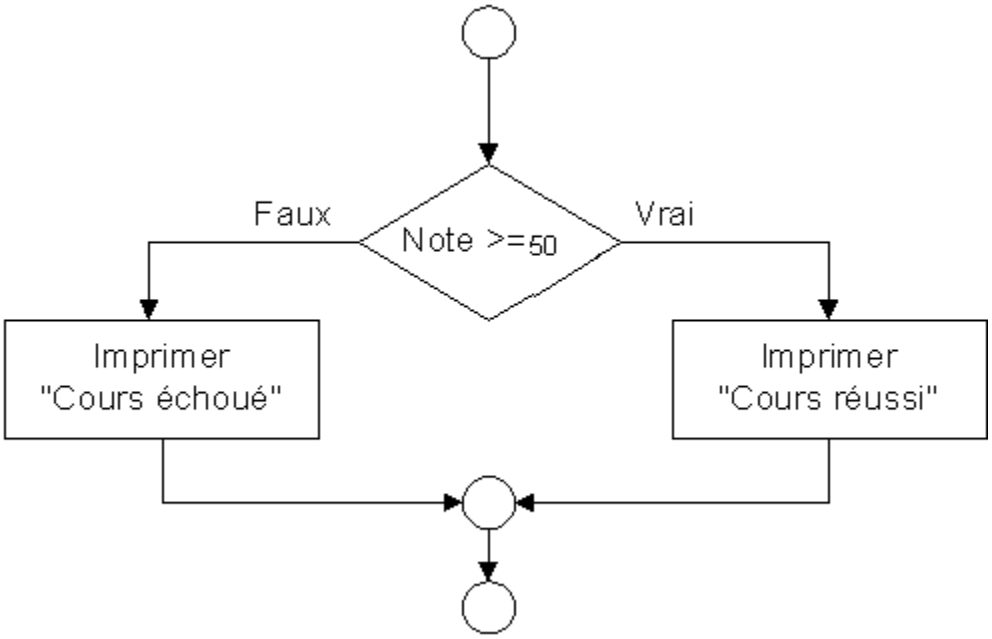

Figure 2 Diagramme de contrôle de la structure `if/else`.

Par ailleurs, la structure `if/else` peut servir à choisir une action spécifique parmi plusieurs. Par exemple, imaginons que nous voulons être plus précis et attribuer une cote à l'étudiant selon la valeur numérique de la note qu'il obtient. Nous pouvons dire que, si l'étudiant obtient une note supérieure à `75`, il obtient un `A`; si la note est

comprise entre `68` et `75`, il obtient un `B`; si elle est comprise entre `60` et `68`, il obtient un `C`; si elle est comprise entre `50` et `60`, il obtient un `D`; finalement, si sa note est inférieure à `50`, l'étudiant échoue son cours et il obtient un `E`.

Exemple :

```
if (note > = 75)
  cout << "A" << endl;
else if (note > = 78)
  cout << "B" << endl;
else if (note > = 60)
  cout << "C" << endl;
else if (note > = 50)
  cout << "D" << endl;
else
  cout << "E" << endl;
```

Dans le même ordre d'idées, lorsque l'on veut introduire plusieurs instructions à l'intérieur d'une structure de contrôle, on insère ces instructions à l'intérieur d'une paire d'accolades.

Exemple :

```
if (note > = 50)
  cout << "Cours réussi" << endl;
else {
  cout << "Cours échoué" << endl;
  cout << "Vous devez absolument le reprendre." << endl;
}
```

Notez que, si l'on omet la paire d'accolades, la commande :

```
cout << "Vous devez absolument le reprendre." << endl;
```

sera à l'extérieur du corps de la partie `else` et sera toujours exécutée, peu importe la valeur de la variable *note*.

Par ailleurs, C++ possède un opérateur conditionnel, représenté par la combinaison des caractères *point d'interrogation* `?` et *deux-points* `:` . Très lié à la structure `if/else`, cet opérateur manipule trois paramètres pour former une expression conditionnelle :

- le premier paramètre constitue la condition à vérifier;
- le deuxième est le résultat de l'expression conditionnelle lorsque la condition est vérifiée;
- alors que le troisième devient le résultat lorsque la condition est fausse.

En utilisant un opérateur conditionnel, le code :

```
if (note > = 50)
   cout << "Cours réussi";
else
   cout << "Cours échoué";
```

Peut être réécrit de la manière suivante :

```
cout << (note > = 50 ? "Cours réussi" : "Cours échoué") << endl;
```

5.2.3 Le branchement conditionnel `switch`

L'écriture de `if` peut être relativement lourde dans certaines situations, particulièrement dans des cas où plusieurs instructions différentes doivent être exécutées selon la valeur d'une variable de type *intégral*. Le langage C++ fournit donc la structure de contrôle `switch` qui permet de faire un branchement conditionnel. La syntaxe utilisée est alors la suivante :

```
switch (valeur)
{
case cas1:
      [Instruction;
[break;]
]
case cas2:
      [Instruction;
[break;]
]
case casN:
      [Instruction;
[break;]
]
[default:
      <[Instruction;
[break;]
]
]
}
```

La valeur est évaluée en premier et son type doit être *entier*. Selon le résultat de l'évaluation, l'exécution du programme se poursuit. Si aucun résultat ne correspond et que `default` est présent, alors l'exécution se poursuit en tenant compte de ce qui suit le `default`. Si par contre il n'y a pas de `default`, alors le programme sort de la structure de contrôle `switch`.

Notons que les instructions qui suivent le *cas approprié* ou le `default` sont d'abord exécutées, puis ce sont les instructions du cas suivant qui le sont. Pour sortir de la structure `switch`, on doit obligatoirement utiliser le mot clé `break`.

Le diagramme de contrôle de la structure `switch` est illustré à la figure 3.

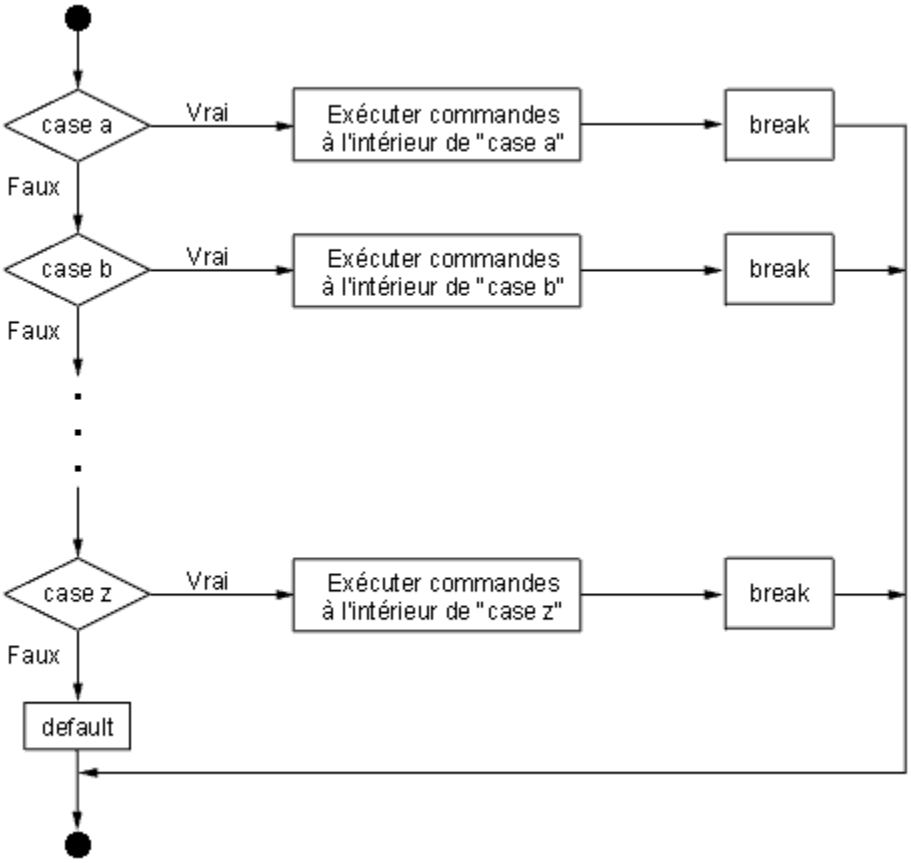

Figure 3 Diagramme de contrôle de la structure `switch`.

Examinons le programme qui illustre comment la structure de sélection `switch` peut être utilisée pour déterminer le nombre d'étudiants d'une classe qui ont obtenu la même note lors d'un examen.

```
X = 2
switch (X)
{
case 1:

case 2:      // Si x = 1 ou 2, la ligne suivante sera exécutée.
      X = 2-X;
break;
```

```
case casN:
      X = 0;/       // Cette ligne ne sera jamais exécutée
default:
      break;

}
```

Notons qu'il est interdit d'effectuer une déclaration de variable dans l'une des instructions de la structure de sélection `switch`.

5.3 Les structures de répétition

Les structures de répétition permettent d'exécuter une opération un certain nombre de fois ou jusqu'à ce que la condition d'arrêt soit atteinte. Pour cela, les structures de répétition comportent une variable de contrôle qui permet de vérifier si la condition d'arrêt est atteinte et cette variable doit être initialisée. En programmation C++, les plus importantes structures de répétition sont le `while`, le `do/while` et le `for`.

5.3.1 La structure de répétition `while`

La structure de répétition `while` permet d'exécuter une instruction tant que la condition d'exécution est vraie.

La syntaxe d'exécution est la suivante :

```
While (test) opération;
```

où `opération` est exécutée tant que le `test` est vérifié. L'ordre d'exécution est la suivante :

```
Test

Opération
```

Voyons un exemple où la structure de répétition `while` est utilisée. Calculons la somme des 10 premiers entiers naturels différents de 0, soit 1, 2, 3, 4, 5, 6, 7, 8, 9 et 10. Cette opération va être effectuée 10 fois, donc notre test prendra la valeur 10.

```
// initialisation des variables.
somme = 0;
test = 1;
While(test <= 10)
{
      somme = somme + test;
      test++;
}
```

Plus concrètement :

```
// Programme en C++
include <iostream.h>
include <iomanip.h>
// initialisation des variables
int main()
{
 int somme = 0;
 int test = 1;
 while(test< = 10)
 {
   somme + = (somme + test);
   test++;
 }
 cout << "La somme est de : "<< endl;
 cout << somme << endl;
 return 0;
}
```

5.3.2 La structure de répétition *do/while*

La structure de répétition `do/while` est similaire à `while`, à la seule différence que `do/while` vérifie la condition de continuation de la boucle après l'exécution du corps de cette boucle. De ce fait, la boucle s'exécute au moins une fois. La forme générale de cette structure est la suivante :

```
do {
 instructions
} while (test);
```

Illustration :

```
// Utilisation de la structure de répétition do/while
// #include <iostream.h> pour windows
#include <iostream>
using namespace std;
main()
{
  int produit = 1 ;
  int test = 1;
  do {
    produit += (produit * test) ;
    cout << " Somme des produits : " << produit << endl;
    test ++;
      } while (test<=10);
      return 0;
}
```

5.3.3 La structure de répétition `for`

La structure de répétition *for* est sans doute l'une des structures les plus importantes. Elle permet de réaliser toutes sortes de boucles, en particulier les boucles d'itération sur une variable de contrôle. Sa syntaxe est la suivante :

```
for (initialisation; test; itération)
      opérations
```

où `initialisation` est une instruction (ou un bloc d'instructions) exécutée avant la première itération, `test` une expression dont la valeur déterminera la fin de la boucle, `itération` l'opération à effectuer en fin de boucle et `opérations` constitue le traitement de la boucle.

Reprenons l'exemple de la section 5.3.2 en utilisant la structure `for` :

```
// Structure de répétition for
// #include <iostream.h>
#include <iostream>
using namespace std;
main()
{
      // Initialisation, condition de répétition et
      // incrémentation sont incluses dans l'en-tête de la
      // structure for
  int produit;
  for (int test = 1; test < = 10; test++)
    produit += (produit * test);
    cout << "Somme des produits : " << produit << endl;
return 0;
}
```

Lorsque l'exécution de la boucle commence, la variable de contrôle `test` est initialisée à `1`, ce qui vérifie la condition `test< = 10`. Les commandes à l'intérieur de la boucle sont alors exécutées et la valeur de `test` est incrémentée. Ceci se répète jusqu'à ce que `test` atteigne la valeur `11`, ce qui fait échouer la condition et arrête la boucle.

5.4 Les commandes de rupture de séquences : `break`, `continue` et `return`

5.4.1 Les commandes break *et* continue

Les commandes `break` et `continue` modifient le flot de contrôle d'un programme. Lorsque la déclaration `break` est utilisée à l'intérieur d'une structure de répétition ou à l'intérieur de la structure `switch`, elle force la sortie immédiate de cette structure. Dans

ce cas, l'exécution du programme se poursuit, mais seulement à partir de la première commande qui suit la structure utilisant le `break`.

D'ailleurs, ce dernier est souvent utilisé pour sortir d'une boucle avant même la condition d'arrêt, ou pour ne pas exécuter le reste d'une structure `switch`. Dans l'exemple qui suit, nous allons afficher à l'écran une partie de mot `teluq`.

```
// Utilisation de break dans une structure for
#include <iostream>
using namespace std;
main()
{
 int x;
 char tableau [6]={'t', 'e', 'l', 'u', 'q'};
 for (int x = 0; x <=5; x++) {
      if (x == 3)
            break;  // Sortir de la boucle si seulement x == 3
                    // 0 = t
                    // 1 = e
                    // 2 = l
      cout << tableau[x] << " ";
}
cout << endl << "Sortir de la boucle si x == 3" << endl;
return 0;
}
```

Le résultat :

```
t e l
Sortir de la boucle si x == 3
```

Par ailleurs, la commande `Continue` saute directement à la dernière ligne de l'instruction `while`, `do` ou `for` la plus imbriquée. Cette ligne est l'accolade fermante. C'est à ce niveau que les tests de continuation sont faits pour `for` et `do`, ou que le saut au début du `while` est effectué (suivi immédiatement du test). On reste donc dans la structure dans laquelle on se trouvait au moment de l'exécution de `continue`, contrairement à ce qui se passe avec la commande `break`.

```
// Utilisation de continue dans une structure for
#include <iostream>
using namespace std;
main()
{
  char tableau [6] ={'t', 'e', 'l', 'u', 'q'};
  for (int x = 0; x <= 5; x++) {
    if (x == 3)
```

```
      continue;   // Sauter le code restant de la boucle si
                  // seulement x == 3
                  // 0 = t
                  // 1 = e
                 // 2 = l
                 // 3 = q

   cout << tableau[x] << " ";
  }
 cout << endl << "Structure continue utilisée pour sauter la
valeur 3" << endl;
 return 0;
}
```

Le résultat :

```
T e l q
Structure continue utilisée pour sauter la valeur 3
```

Dans ce programme, lorsque la valeur de `x` devient `3`, la structure `continue` est exécutée, ceci fait sauter le reste du code du corps de la structure `for` et amène immédiatement le programme à la prochaine itération.

Ainsi, la commande :

```
cout << tableau[x] << " ";
```

n'est pas exécutée et, de ce fait, la valeur `4` n'est pas affichée à l'écran lors de l'exécution du programme.

5.4.2 La commande `return`

La commande `return` permet de quitter immédiatement la fonction en cours; cette commande peut prendre comme paramètre la valeur de retour de la fonction.

6. Affectation

L'affectation permet d'assigner à une variable une valeur donnée. Soit une variable `x` qui est un entier. Si nous désirons lui ajouter une valeur égale à `8`, nous procéderons de la manière suivante :

```
x = x + 8
```

Cette ligne peut être récrite de la façon suivante, en utilisant l'opérateur `+=` :

```
x += 8
```

Dans ce cas, l'opérateur `+=` ajoute la valeur de l'expression de droite à celle de la variable de gauche. Nous pouvons généraliser ainsi :

Toute expression de la forme :

```
variable = variable opérateur expression;
//Le point-virgule est nécessaire.
```

où l'*opérateur* peut être représenté par l'un des opérateurs binaires suivants :

`+` : addition
`?` : soustraction
`*` : multiplication
`/` : division
`%` : reste de la division

peut être récrite ainsi :

```
variable opérateur = expression;
//Le point-virgule est nécessaire.
```

7. Incrémentation et décrémentation

7.1 L'incrémentation

L'incrémentation consiste à ajouter la valeur entière `1` à une variable quelconque jusqu'à l'obtention d'une valeur désirée. L'incrémentation peut se faire de deux manières :

- la pré-incrémentation : `++variable`
- la post-incrémentation : `variable++`

Soit la variable `x` un entier; si cette variable au départ est égale à `5` et si nous appliquons à cette variable les deux types d'incrémentation avant d'afficher cette variable, nous aurons `x = 6` puis l'affichage de `x` avec la pré-incrémentation et `x = 5`, l'affichage de `x` puis le passage de `x` à la valeur `6` avec la post-incrémentation.

7.2 La décrémentation

Toutes les opérations que nous venons de voir peuvent être réutilisées en remplaçant l'opérateur `++` par l'opérateur `--`, l'opérateur de décrémentation.

Par exemple, `x--` signifie qu'il faut utiliser la valeur courante de la variable `x` dans l'expression où elle apparaît et la décrémenter ensuite, alors que `--x` consiste à décrémenter `x`, puis à utiliser sa nouvelle valeur dans une expression donnée. On parle alors de pré-décrémentation et de post-décrémentation.

Voici une illustration :

```
// Post-décrémentation et pré-décrémentation
#include <iostream>
using namespace std;
main()
{
int x;
x = 10;
cout << x << endl;
cout << x-- << endl; // Post-décrémentation : afficher x, puis
                     //décrémentation
cout << x << endl << endl;
x = 10;
cout << x << endl;
cout << --x << endl; // Pré-décrémentation : décrémenter x, puis
                      // l'afficher
return 0;
```

qui donne comme résultats :

```
10
10
9

10
9
```