

COURS C++

Programmation structurée C++ de base

I.U.T. informatique – Semestre 1

Christine JULIEN

TABLE DES MATIERES

CHAPITRE 1 : INTRODUCTION.....	5
1.1. ORIGINE	5
1.2. DU SOURCE A L'EXECUTABLE	6
1.2.1. Environnement de programmation en C++.....	6
1.2.2. Le cycle de développement d'un programme.....	6
CHAPITRE 2 : STRUCTURE D'UN PROGRAMME C++	7
2.1. STRUCTURE D'UN PROGRAMME SIMPLE	7
2.2. STRUCTURE D'UN PROGRAMME PLUS COMPLEXE	8
CHAPITRE 3 : VARIABLES ET CONSTANTES	10
3.1. LES VARIABLES	10
3.1.1. Définition	10
3.1.2. Types prédéfinis	10
3.1.3. Déclaration de variables.....	10
3.2. LES CONSTANTES	11
3.2.1. Définition	11
3.2.2. Constantes littérales.....	11
3.2.3. Constantes symboliques	11
CHAPITRE 4 : OPERATEURS ET INSTRUCTIONS.....	12
4.1. LES OPERATEURS	12
4.1.1. Définition	12
4.1.2. L'opérateur d'affectation.....	12
4.1.3. Les opérateurs arithmétiques.....	12
4.1.4. Les opérateurs relationnels.....	13
4.1.5. Les opérateurs logiques	13
4.2. LES INSTRUCTIONS	14
4.2.1. Définition	14
4.2.2. La séquence.....	14
4.2.3. La sélection	14
4.2.4. La sélection à choix multiples ou aiguillage.....	15
4.2.5. La répétition.....	16
4.2.5.1. Boucle <i>tantque</i>	16
4.2.5.2. Boucle <i>répéter ... tantque</i>	17
4.2.5.3. Boucle <i>pour</i>	17
CHAPITRE 5 : TABLEAUX ET ENREGISTREMENTS	18
5.1. LES TABLEAUX.....	18
5.1.1. Définition	18
5.1.2. Définition d'un type tableau	18
5.1.3. Déclaration d'un tableau	18
5.1.4. Opération d'accès à un élément d'un tableau	18
5.1.5. Opérations sur les tableaux	19
5.1.6. Cas particulier : les chaînes de caractères.....	19
5.1.6.1. Définition	19
5.1.6.2. Opérations sur les chaînes.....	19

5.2. LES ENREGISTREMENTS.....	19
5.2.1. Définition	19
5.2.2. Définition d'un type enregistrement.....	20
5.2.3. Déclaration d'un enregistrement.....	20
5.2.4. Opération d'accès à un champ d'un enregistrement	20
5.2.5. Opérations sur les enregistrements.....	21
5.2.5.1. Affectation	21
5.2.5.2. Comparaison	21
CHAPITRE 6 : SOUS-PROGRAMMES.....	22
6.1. DEFINITION	22
6.2. LES SOUS-PROGRAMMES	22
6.2.1. Traduction des procédures.....	22
6.2.2. Traduction des fonctions.....	23
6.2.3. Résumé	23
6.3. LES MODES DE TRANSMISSION DES PARAMETRES	23
6.3.1. Définition	23
6.3.2. Paramètres de types prédéfinis ou enregistrements.....	24
6.3.3. Paramètres de type tableau.....	24
6.4. APPEL D'UNE FONCTION C++	25
6.5. DECOMPOSITION D'UN PROGRAMME EN SOUS-PROGRAMMES.....	26
CHAPITRE 7 : EXCEPTIONS.....	27
7.1. DEFINITION	27
7.2. LEVEE D'UNE EXCEPTION	27
7.3. INTERCEPTION D'UNE EXCEPTION	27
CHAPITRE 8 : FONCTION SURCHARGEE ET FONCTION GENERIQUE	29
8.1. FONCTION SURCHARGEE	29
8.2. FONCTION GENERIQUE	29
ANNEXES.....	31
ANNEXE 1 : LES MOTS RESERVES DU LANGAGE.....	31
ANNEXE 2 : LES TYPES PREDEFINIS	31
ANNEXE 3 : LES PRINCIPAUX CARACTERES SPECIAUX.....	32
ANNEXE 4 : LES PRINCIPAUX OPERATEURS	32
ANNEXE 5 : LES ENTREES/SORTIES	33
ANNEXE 6 : LES CHAINES DE CARACTERES.....	36
ANNEXE 7 : TABLE DES CODES ASCII	37
ANNEXE 8 : EXEMPLE DE PROGRAMME C++	38

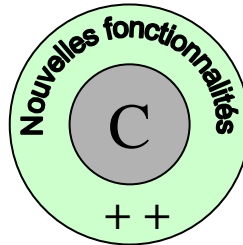
Chapitre 1 : INTRODUCTION

1.1. Origine

C++ a été conçu par *Bjarne Stroustrup* au début des années 80. C++ est une extension du langage C, initialement conçu par *Dennis Ritchie* durant les années 70.

A l'origine langage natif du système Unix, C s'est imposé par la suite comme un langage universel. Il présente la caractéristique d'être à la fois un langage de programmation relativement proche de l'architecture des ordinateurs (octets, mots, pointeurs, tableaux, ...) et un langage structuré de haut niveau (structures de contrôle, récursivité, typage, ...). C'est aussi un langage réputé pour son efficacité en temps d'exécution.

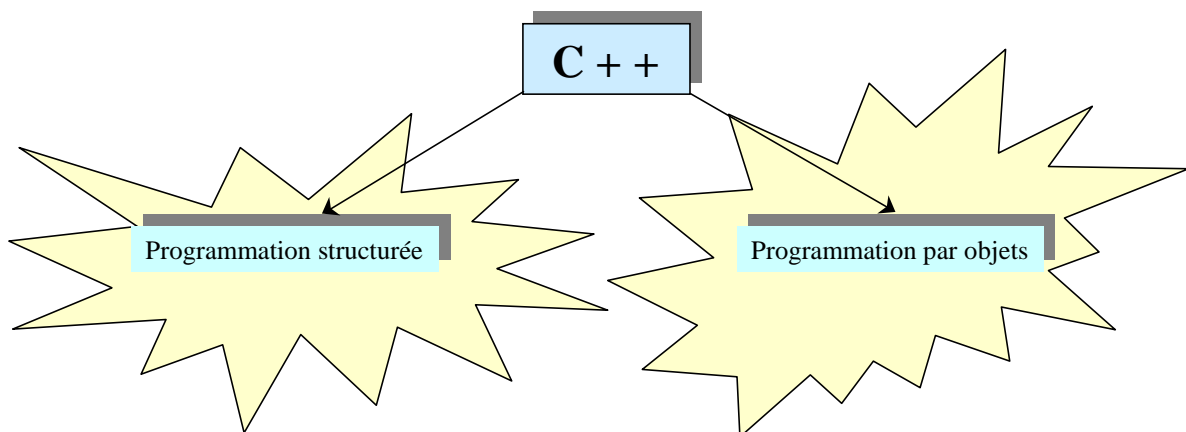
C++ est une sur-couche du C ; c'est-à-dire qu'il rajoute au C de nouvelles fonctionnalités. Pratiquement, mis à part quelques exceptions, tout programme écrit en langage C est compatible avec un programme écrit en C++ (compatibilité ascendante), alors que l'inverse n'est pas vrai.



Pourquoi C++ ?

C++ a été créé principalement pour 2 raisons :

- D'une part pour corriger les imperfections du C. Celles-ci sont notamment liées au fait que C est un langage très permissif en matière de typage, C ne gère pas de façon simple les entrées/sorties, le passage de paramètre, l'allocation dynamique, ...
- D'autre part pour pouvoir faire de la *programmation par objets*. La programmation par objets remonte aux années 80 et correspond à une nouvelle façon de concevoir des programmes. On utilisera ce type de programmation pour programmer des types abstraits de données (C++ 2ème partie).



Pourquoi l'apprentissage de C++ ?

- il n'a pas certains inconvénients du langage C,
- ce langage est très utilisé dans l'industrie,
- il est disponible sur tout type d'ordinateur,

1.2. Du source à l'exécutable

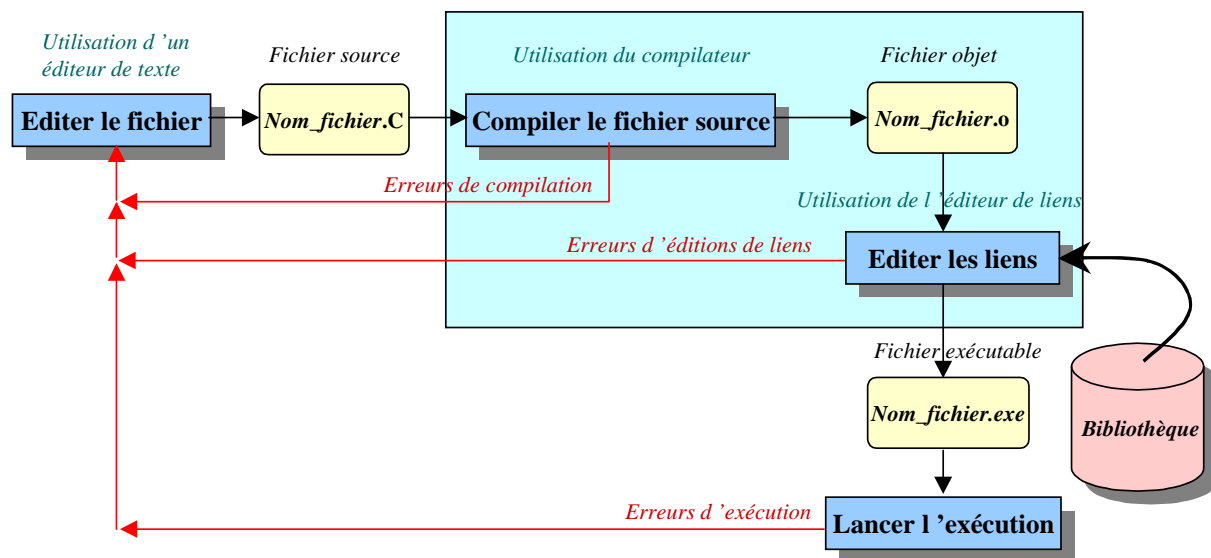
1.2.1. Environnement de programmation en C++

Pour écrire et exécuter un programme écrit en C++, il faut disposer :

- D'un *éditeur de texte* (pour écrire le programme source).
- D'un *compilateur - éditeur de liens* (pour transformer le code source en code binaire exécutable par la machine).
- D'une *bibliothèque standard* fournie avec le compilateur. Cette bibliothèque regroupe des fichiers qui vont permettre l'utilisation de fonctions externes comme par exemple les opérations d'entrées/sorties.

1.2.2. Le cycle de développement d'un programme

Le schéma suivant illustre le cycle de développement d'un programme écrit en C++.



Le compilateur est un programme qui permet de détecter des erreurs syntaxiques (exemple un *if* doit toujours être suivi d'une parenthèse ouvrante) et de traduire le programme en langage binaire.

L'éditeur de liens permet de rendre un programme exécutable.

Souvent, on peut regrouper dans une même commande les directives de compilation et d'édérations de liens.

Remarque : Ce cycle de développement est lié à n'importe quel programme écrit dans un langage compilé. Seules les commandes diffèrent.

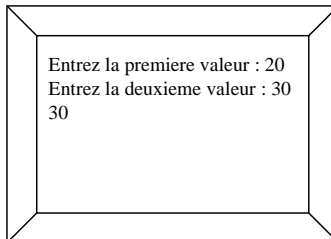
Chapitre 2 : STRUCTURE D'UN PROGRAMME C++

2.1. Structure d'un programme simple

Voici un exemple de programme simple permettant de saisir deux valeurs entières à partir du clavier et d'afficher la plus grande à l'écran.

Algorithme	Programme C++ : max2entiers.C
<pre>importer entréesSortie ; -- détermine le plus grand des 2 entiers a et b programme calculerLePlusGrand glossaire a <Entier> ; -- 1^{er} entier à comparer b <Entier> ; -- 2^{er} entier à comparer début -- lire les 2 entiers a et b écrire ("Entrez la première valeur : ") ; lire (a) ; écrire ("Entrez la deuxième valeur : ") ; lire (b) ; -- calculer et écrire le plus grand des 2 entiers si a > b alors écrire (a) ; sinon écrire (b) ; fin si ; fin.</pre>	<pre>#include "/usr/local/public/BIBLIOC++/entreeSortie.h" #include "/usr/local/public/BIBLIOC++/chaine.h" // détermine le plus grand des 2 entiers a et b int main() { int ai ; // premier entier a comparer int bi ; // deuxième entier a comparer // lire les 2 entiers a et b écrire(uneChaine ("Entrez la première valeur : ")); lire(ai) ; écrire(uneChaine ("Entrez la deuxième valeur : ")); lire(bi) ; // calculer et écrire le plus grand des 2 entiers if (ai > bi) écrire(ai); else écrire(bi); }</pre>

A l'exécution :



```
Entrez la première valeur : 20
Entrez la deuxième valeur : 30
30
```

Outre les commentaires introduits par les symboles `//`, le programme ci-dessus se décompose en 2 grandes parties :

- la partie «*directives d'inclusion de fichiers d'en-tête*»,
- la partie «*programme principal*».

a) Directives d'inclusion de fichiers d'en-tête

Le langage C++ ne possède pas d'instructions d'entrée/sortie spécifiques. Pour pouvoir faire des entrées/sorties, il est nécessaire de faire appel à des sous-programmes externes de la bibliothèque standard.

Les en-têtes des sous-programmes externes se trouvent dans des fichiers d'extension `.h`. Il est indispensable d'inclure dans le programme source les fichiers d'en-tête auxquels appartiennent les sous-programmes externes appelés. Cette inclusion se fait grâce à la directive «`#include ...`».

Pour simplifier les entrées/sorties standards, nous avons développé notre propre bibliothèque de sous-programmes (*cf. annexe 5 : les entrées/sorties*). Cette bibliothèque est accessible par tout programme C++ en introduisant la directive :

```
#include "/usr/local/public/BIBLIOC++/entreeSortie.h"
```

b) Le programme principal

Il s'introduit par le mot-clé *main* qui correspond au nom du sous-programme auquel le système donne la main lors de l'exécution d'un programme.

Les instructions se trouvant à l'intérieur du *main* sont délimitées par "{ }". Chaque instruction se termine par ";". A l'intérieur du bloc d'instructions, nous trouvons :

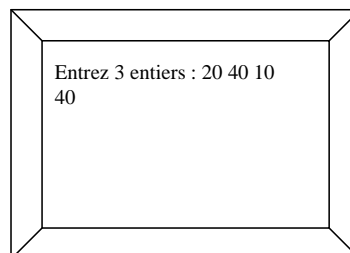
- **Les déclarations de variables** : C++ est un langage **typé**. Cette contrainte oblige l'utilisateur à définir pour toute variable son type. Un **type** indique le domaine des valeurs pouvant être prises par la variable ainsi que les opérations permises. Il existe plusieurs types prédéfinis par le langage (*cf. chapitre 3*). Par exemple *int* est le type correspondant aux entiers. Les déclarations de variables peuvent se faire à n'importe quel endroit du bloc.
- **Les instructions du programme** : *if* est un mot réservé qui introduit une sélection. Nous reviendrons sur les structures de contrôle dans le *chapitre 4*.

2.2. Structure d'un programme plus complexe

Voici un programme qui consiste à saisir 3 valeurs entières et à afficher la plus grande. Dans cet exemple, le programme principal fait appel à une fonction qui retourne le maximum de 2 valeurs entières.

Algorithme	Programme C++ : max3entiers.c
<pre>importer entréeSortie ; -- retourne le plus grand entier des 2 entiers x et y fonction max (entrée x <Entier>, entrée y <Entier>) retourne <Entier> début si x > y alors retourner (x) ; sinon retourner (y) ; fin si ; fin. -- détermine le plus grand entier des 3 -- entiers a, b et c programme calculerLePlusGrand glossaire a <Entier> ; -- 1^{er} entier à comparer b <Entier> ; -- 2^{es} entier à comparer c <Entier> ; -- 3^{es} entier à comparer début -- lire les 3 entiers a, b et c écrire ("Entrez 3 entiers : ") ; lire (a) ; lire (b) ; lire (c) ; -- calculer et écrire le plus grand -- des 3 entiers écrire (max (a, max (b, c))) ; fin.</pre>	<pre>#include "/usr/local/public/BIBLIO++/entreeSortie.h" #include "/usr/local/public/BIBLIO++/chaine.h" // retourne le plus grand entier des 2 entiers x et y int max(const int x, const int y) { if (x > y) return (x); else return (y); } // détermine le plus grand entier des 3 // entiers a, b et c int main() { int a, b, c; // entiers a comparer // lire les 3 entiers a, b et c écrire (uneChaine("Entrez 3 entiers : ")); lire(a); lire(b); lire(c); // calculer et écrire le plus grand des // 3 entiers écrire(max(a, max(b, c))); }</pre>

A l'exécution :



```
Entrez 3 entiers : 20 40 10
40
```

Plusieurs remarques sur l'écriture du programme peuvent être faites :

- Tout identificateur de fonction C++ doit être précédé du type de la valeur qu'elle retourne. Dans l'exemple précédent, la fonction *max()* retourne un entier. De même

que les variables sont typées, les paramètres le sont aussi et chacun d'eux dans l'en-tête de la fonction doit être précédé de son type. Nous reviendrons sur les modes de transmission des paramètres dans le *chapitre 6*.

- Plusieurs instructions peuvent être placées sur une même ligne.
- On peut déclarer plusieurs variables d'un même type en une seule écriture.

Remarque : Appel de sous-programme en C++

Un sous-programme f peut appeler un autre sous-programme g si la **définition** (c'est-à-dire le **corps**) de g précède la définition de f (par exemple, la fonction *main* de l'exemple précédent peut faire un appel à la fonction *max*). Si tel n'est pas le cas, il faut déclarer le sous-programme g sous forme de prototype.

Un **prototype** de sous-programme appelé encore **signature** correspond à l'en-tête du sous-programme suivi d'un `;`.

Ainsi en prototypant la fonction *max* en début de programme, nous pouvons fournir la définition de la fonction *max* après la définition de la fonction *main* de la manière suivante :

```
#include "/usr/local/public/BIBLIO++/entreeSortie.h"

// prototype de la fonction max
int max(const int x, const int y);

// corps de la fonction main
int main()
{
    int a, b, c; // entiers a comparer
    // lire les 3 entiers a, b et c
    ecrire(uneChaine ("Entrez 3 entiers : "));
    lire(a); lire(b); lire(c) ;
    // calculer et ecrire le plus grand des
    // 3 entiers
    ecrire(max (a, max(b, c)));
}

// corps de la fonction max
int max(const int x, const int y)
{
    if (x > y)
        return (x);
    else
        return (y);
}
```

Chapitre 3 : VARIABLES ET CONSTANTES

3.1. Les variables

3.1.1. Définition

Une *variable* est un emplacement mémoire destiné à recevoir une valeur.

Une variable peut occuper 1 ou n octets consécutifs en mémoire. La taille d'une variable dépend de son *type* (entier, caractère, réel, ...).

Un *identificateur* de variable est un nom symbolique qui désigne explicitement son emplacement mémoire. Il ne doit pas correspondre à un mot réservé du langage (*cf. annexe 1 : les mots réservés*).

La *valeur* d'une variable représente le contenu de son emplacement mémoire.

Exemple :

A

short A = 125; 125 Codage sur 2 octets

3.1.2. Types prédéfinis

En C++, il existe plusieurs types prédéfinis pour représenter les caractères (*char*), les booléens (*bool*), les entiers (*short*, *int*, *long*) ou encore les nombres réels (*float*, *double*).

À chaque type est associé un domaine de valeurs possibles (*cf. annexe 2 : les types prédéfinis*), ainsi qu'un ensemble d'opérations.

3.1.3. Déclaration de variables

Pour pouvoir utiliser une variable, il faut qu'elle ait fait l'objet d'une déclaration. Celle-ci consiste à fournir conjointement un identificateur de variable et le type de la variable (*type suivi de l'identificateur*).

Un identificateur de variable doit commencer par une lettre suivie éventuellement de n'importe quelle combinaison de lettres, de chiffres et du symbole `_` (souligné).

C++ fait la différence entre les majuscules et les minuscules. Ainsi *var* et *VAR* représentent deux variables distinctes.

Exemples :

```
int var;  
long VAR;  
char lettre1, lettre2, lettre3;  
bool TRAIT1_FINI, finBoucle;
```

Dans une déclaration, le symbole `=` permet d'affecter **initialement** une valeur à une variable. En l'absence du symbole `=`, le contenu de la variable est indéfini.

Exemples :

```
int var = 1;  
char lettre1 = 'A', lettre2 = 'B', lettre3;  
bool TRAIT1_FINI, finBoucle = true;
```

3.2. Les constantes

3.2.1. Définition

Une *constante* est une zone de stockage d'information non modifiable.

En C++, on distingue deux types de constantes :

- les **constantes littérales**,
- les **constantes symboliques**.

3.2.2. Constantes littérales

Une *constante littérale* est une valeur définie par le programmeur qui peut être affectée à une variable ou bien utilisée dans l'évaluation d'une expression.

Exemples :

34.456	<i>constante réelle</i>
12.346 e2	<i>constante réelle notation scientifique</i>
123	<i>constante entière</i>
'A'	<i>constante caractère</i>
"cours d'info."	<i>constante chaîne de caractères</i>

Remarque :

Il existe une syntaxe particulière pour certains caractères spéciaux. Cette syntaxe utilise un caractère précédé du caractère d'échappement « \ ». Ces caractères peuvent aussi être utilisés dans des chaînes de caractères. Par exemple, le caractère '\n' représente une nouvelle ligne (*cf. annexe 3 : principaux caractères spéciaux*).

3.2.3. Constantes symboliques

Une *constante symbolique* consiste à fournir un identificateur à une constante littérale. Elle est introduite par le mot réservé *const* et est suivie d'une déclaration avec affectation de valeur. Son contenu ne peut pas être modifié.

Exemples :

const float PI = 3.14;	<i>// constante décimale</i>
const char ESPACE = ' ';	<i>// constante caractère</i>
float x = 5 * PI;	<i>// utilisation d'une constante symbolique</i>

Chapitre 4 : OPERATEURS ET INSTRUCTIONS

4.1. Les opérateurs

4.1.1. Définition

Un *opérateur* est un symbole qui permet d'effectuer une action. Les opérateurs agissent sur des opérandes.

Un opérateur possède une **pluralité** et une **priorité**.

La *pluralité* (ou *arité*) d'un opérateur indique le nombre d'opérandes dont a besoin l'opérateur pour effectuer l'action. On parle d'opérateurs unaires (1 opérande), binaires (2 opérandes) ou encore ternaires (3 opérandes).

Dans une expression contenant plusieurs opérateurs, la *priorité* des opérateurs détermine l'ordre dans lequel seront évaluées les opérations. Les opérateurs qui ont la plus grande priorité sont évalués d'abord. On peut toutefois modifier la priorité des opérateurs en utilisant le parenthésage (cf. *annexe 4 : principaux opérateurs*).

4.1.2. L'opérateur d'affectation

L'*opérateur d'affectation* ($=$) est binaire. Il oblige l'opérande gauche à recevoir la valeur de l'opérande droit. Il traduit l'action élémentaire du langage algorithmique symbolisée par \leftarrow .

Exemples :

```
int somme, a, b, numCompte = 1000;
bool test;
somme = 0;
somme = numCompte;
test = (a == b);
somme = somme + 1;
```

4.1.3. Les opérateurs arithmétiques

Les *opérateurs arithmétiques* sont des opérateurs binaires. Ils permettent d'effectuer des opérations sur les entiers et les réels.

Il existe cinq opérateurs arithmétiques :

+	Addition
-	Soustraction
*	Multiplication
/	Division
%	Modulo (reste de division entière)

Exemples :

```
int somme = 0, resultat;
const int INCREMENT = 5;
somme = somme + 10; // somme = 10
somme = somme + 8 * 3 + INCREMENT; // somme = 39
resultat = somme % 5; // resultat = 4
resultat = (resultat + 32) / 5; // resultat = 7

float op, pi = 3.14;
op = 567.89 + 78.8 - pi; // op = 643,55

char c = 'A'; // c = 65
c = c + 1; // c = 66 = 'B'
c = 'D' + 2; // c = 70 = 'F'
```

Remarque : Conversions explicites et implicites

C++ autorise une conversion implicite d'un type vers un autre. Mais il faut faire attention aux possibles pertes d'informations.

Exemples :

```
int e;
float r;
char c;
...
e = r * 5.7;           // attention
r = e;                 // oui
c = e;                 // attention
```

Il est possible de convertir explicitement des valeurs dans un autre type en faisant un forçage de type de la forme : *nom_de_type (expression)*.

Exemples :

```
int n = 100 ;
double racine ;
racine = sqrt(double (n * 10));    // sqrt attend un type double

int nbVal = 4 , somme = 50;
float moyenne;
moyenne = somme / float (nbVal);    // nbVal est converti en flottant
```

Remarque : Les opérateurs d'incrément et de décrémentation

Ces opérateurs unaires consistent à ajouter (++) ou à soustraire (- -) la valeur 1 à une variable.

Exemples :

```
int i = 10;
i++;           // ⇔ i = i + 1
i--;           // ⇔ i = i - 1
```

4.1.4. Les opérateurs relationnels

Les *opérateurs relationnels* permettent de comparer deux opérandes et sont utiles pour exprimer les conditions. Le résultat obtenu est une valeur booléenne vrai (*true*) ou faux (*false*).

Il existe 6 opérateurs relationnels :

==	Égal à
!=	Différent de
>	Supérieur à
>=	Supérieur ou égal à
<	Inférieur à
<=	Inférieur ou égal à

Exemples :

```
int i = 10;
ecrire (i == 10);    // 1 (vrai)
ecrire (i != 10);    // 0 (faux)
ecrire (i < 10);     // 0 (faux)
```

4.1.5. Les opérateurs logiques

Les *opérateurs logiques* sont utiles pour évaluer plusieurs conditions. Le résultat obtenu est une valeur booléenne *vrai* (1) ou *faux* (0).

Il existe 3 opérateurs logiques :

&&	et logique (opérateur binaire)
 	ou logique (opérateur binaire)
!	non logique (opérateur unaire)

Exemples :

```
int i = 10, j = 5;
bool trouve = true;
ecrire ((i == 10) && (j == 5)); // 1 (vrai)
ecrire ((i == 5) || (j == 5)); // 1 (vrai)
ecrire (!(i < 5)); // 1 (vrai)
ecrire (!trouve); // 0 (faux)
```

4.2. Les instructions

4.2.1. Définition

Une *instruction* indique une action à réaliser.

Toutes les instructions se terminent par `;`. Toutefois, il est possible de mettre plusieurs instructions sur une même ligne.

Les *blancs* (espace, tabulation, retour à la ligne) sont ignorés.

Les structures de contrôle conventionnelles de l'algorithmique existent en C++. Elles permettent de définir des schémas d'ordonnancement des instructions.

4.2.2. La séquence

La séquence exprime un enchaînement inconditionnel d'actions.

En C++, un *bloc* permet de regrouper un ensemble d'instructions et peut être assimilé à une séquence. Il est délimité par `{` et `}`. Les blocs peuvent être imbriqués les uns dans les autres.

Les variables déclarées dans un bloc ne sont accessibles que dans ce bloc ou dans les sous-blocs de ce bloc.

4.2.3. La sélection

La *sélection* exprime le choix entre deux actions. Elle est introduite par le mot réservé *if*. Sa syntaxe est la suivante :

```
if (condition)
    action1;
else
    action2;
```

Toute condition doit obligatoirement être encadrée par des parenthèses.

Une condition amène toujours un résultat booléen.

Exemples :

Algorithme	Traduction C++
<pre>si a > b alors a <- b ; sinon b <- a ; fin si ;</pre>	<pre>if (a > b) a = b; else b = a;</pre>
<pre>si a > b alors a <- b ; fin si ;</pre>	<pre>if (a > b) a = b;</pre>

<pre> si a > b alors a <- b ; b <- c ; f <- e ; sinon b <- a ; fin si ; </pre>	<pre> if (a > b) { a = b; b = c; f = e; } else b = a; </pre>
--	---

Tout comme en algorithmique, il est possible d'imbriquer des sélections dans d'autres sélections. Un *else* se rapporte toujours au *if* le plus proche.

Exemples :

Algorithme	Traduction C++
<pre> si n = 0 alors si a > b alors z <- a ; sinon z <- b ; fin si ; fin si ; </pre>	<pre> if (n == 0) if (a > b) z = a; else z = b; </pre>
<pre> si n = 0 alors si a > b alors z <- a ; fin si ; sinon z <- b ; fin si ; </pre>	<pre> if (n == 0) { if (a > b) z = a; } else z = b; </pre>

4.2.4. La sélection à choix multiples ou aiguillage

La *sélection à choix multiples* est une généralisation de la sélection. Elle permet d'effectuer une action en fonction d'une valeur prise par une expression entière (*char* ou *int*). Elle est introduite par un *switch*. Sa syntaxe est la suivante :

<pre> switch (expression) { case constante1 : action1; case constante2 : action2; ... default : actionN; } </pre>

- *expression* doit être une expression entière,
- *constante1*, *constante2*, ... sont des constantes entières désignant des étiquettes.

Cette structure de contrôle provoque un branchement vers l'étiquette correspondant à la valeur de l'expression. Une fois le branchement effectué, les actions sont exécutées en séquence jusqu'à la fin du *switch* ou jusqu'à la rencontre d'une instruction *break*. L'étiquette *default* est facultative. Elle introduit le traitement à effectuer lorsque la valeur de l'expression ne correspond à aucune des étiquettes spécifiées.

Exemples :

Algorithme	Traduction C++
<pre> si choix = 1 alors b <- a ; sinon si choix = 2 alors a <- b ; sinon c <- d ; fin si ; fin si ; </pre>	<pre> int choix ; ... switch (choix) { case 1 : b = a; break; case 2 : a = b; break; default : c = d; } </pre>

<pre> si c = 'A' alors x <- 0 ; y <- 0 ; sinon si c = 'B' ou c = 'C' alors x <- 1 ; y <- 1 ; sinon si c = 'D' ou c = 'E' alors x <- 2 ; y <- 2 ; sinon si c = 'F' alors x <- 3 ; y <- 3 ; fin si ; fin si ; fin si ; fin si ; </pre>	<pre> char c ; ... switch (c) { case 'A' : x = 0; y = 0; break; case 'B' : case 'C' : x = 1; y = 1; break; case 'D' : case 'E' : x = 2; y = 2; break; case 'F' : x = 3; y = 3; } </pre>
<pre> si n = 4 alors écrire ("multiple de 4\n") ; écrire ("multiple de 2\n") ; sinon si n = 2 alors écrire ("multiple de 2\n") ; sinon si n = 9 alors écrire ("multiple de 9\n") ; écrire ("multiple de 3\n") ; sinon si n = 3 alors écrire ("multiple de 3\n") ; fin si ; fin si ; fin si ; fin si ; </pre>	<pre> int n ; ... switch (n) { case 4 : écrire(uneChaine ("multiple de 4\n")); case 2 : écrire(uneChaine ("multiple de 2\n")); break; case 9 : écrire(uneChaine("multiple de 9\n")); case 3 : écrire(uneChaine("multiple de 3\n")); } </pre>

4.2.5. La répétition

La répétition exprime qu'une action est répétée tant qu'une condition reste vérifiée. En C++, il existe plusieurs formes pour exprimer une répétition : le *while*, le *do while* et le *for*.

4.2.5.1. Boucle *tantque*

Elle traduit exactement le *tantque* algorithmique. Sa syntaxe est la suivante :

```

while (condition)
  action;

```

Exemples :

Algorithmme	Traduction C++
<pre> s <- 0 ; i <- 1 ; tantque i <= 10 faire s <- s + i ; i <- i + 1 ; fin tantque ; </pre>	<pre> s = 0; i = 1; while (i <= 10) { s = s + i; i++; } </pre>
<pre> s <- 0 ; lire (val) ; tantque val /= 0 faire s <- s + val ; lire (val) ; fin tantque ; </pre>	<pre> s = 0; lire (val); while (val != 0) { s = s + val; lire(val); } </pre>

4.2.5.2. Boucle *répéter ... tantque*

Cette structure introduite par *do ... while* traduit une boucle qui s'exécute au moins une fois. En effet, la condition de boucle est placée après l'action à réaliser. Sa syntaxe est la suivante :

```
do
    action ;
while (condition) ;
```

Exemple :

Algorithme	Traduction C++
<pre>écrire ("Entrez une valeur ") ; lire (val) ; tantque val <= 0 faire écrire ("Entrez une valeur ") ; lire (val) ; fin tantque ;</pre>	<pre>do { écrire(uneChaine("Entrez une valeur ")); lire(val); } while (val <= 0);</pre>

4.2.5.3. Boucle *pour*

Elle s'introduit par *for* et traduit une boucle qui porte sur une ou plusieurs variables de contrôle devant subir une variation à chaque itération. On parle de traitements répétés « réguliers ». Sa syntaxe est la suivante :

```
for (initialisation ; condition ; variation)
    action ;
```

- *initialisation* représente la ou les actions qui permettent d'initialiser la ou les variables de contrôle. Chaque action doit être séparée de la suivante par une virgule.
- *condition* représente la condition de boucle.
- *variation* représente la ou les actions qui modifient la ou les variables de contrôle avant de passer à l'itération suivante. Chaque instruction doit être séparée de la suivante par une virgule.

Exemples :

Algorithme	Traduction C++
<pre>s <- 0 ; i <- 1 ; n <- 0 ; tantque i <= 10 faire n <- s ; s <- s + i ; i <- i + 1 ; fin tantque ;</pre>	<pre>s = 0; n = 0; for (i = 1 ; i <= 10 ; i++) { n = s; s = s + i; }</pre>
<pre>i <- 1 ; j <- 10 ; tantque i <= 10 et j > 5 faire écrire (i) ; i <- i + 3 ; j <- j - 1 ; fin tantque ;</pre>	<pre>for (i=1, j=10 ; i<=10 && j>5 ; i=i+3, j--) écrire(i) ;</pre>

Chapitre 5 : TABLEAUX ET ENREGISTREMENTS

5.1. Les tableaux

5.1.1. Définition

Un *tableau* permet de stocker consécutivement en mémoire des valeurs de même type. Chaque élément est repéré par un indice fournissant le rang qu'il occupe dans le tableau.

5.1.2. Définition d'un type tableau

L'utilisation d'un tableau dans un programme nécessite auparavant la définition de son type.

Une définition d'un type tableau, introduite par le mot réservé *typedef*, consiste à fournir :

- le *type des éléments* du tableau,
- l'*identificateur du type tableau* (nom symbolique fourni par l'utilisateur),
- la *taille* du tableau (expression entière) entre crochets.

La syntaxe est la suivante :

```
type_des_éléments nom_du_type_tableau[taille_tableau] ;
```

Exemples :

```
const int NBMAX = 100;

typedef int Tab1[6];           // définition d'1 tableau 1D de 6 entiers
typedef float Tab2[NBMAX];     // définition d'1 tableau 1D de 100 réels
typedef int Tab3[3][5];       // définition d'1 tableau 2D de 15 entiers
typedef char Tab4[2][2][3];    // définition d'1 tableau 3D de 12 caractères
```

5.1.3. Déclaration d'un tableau

Une déclaration de tableau consiste, de manière analogue à une déclaration de variable de type prédéfini, à fournir un identificateur précédé du type du tableau. Ce dernier doit avoir fait auparavant l'objet d'une définition de type (cf. § 5.1.2).

Exemples :

```
Tab1 a;                       // déclaration d'1 tableau 1D de 6 entiers
Tab2 b, c;                     // déclaration de 2 tableaux 1D de 100 réels
Tab3 d;                         // déclaration d'1 tableau 2D de 15 entiers
```

Un tableau peut être initialisé lors de sa déclaration en spécifiant une liste de valeurs entre accolades.

Exemples :

```
Tab1 x = {10, 20, 30, 40, 50, 60};
Tab3 y = {
    { 1, 2, 3, 4, 5},
    {10, 20, 30, 40, 50},
    {100, 200, 300, 400, 500}
};
```

5.1.4. Opération d'accès à un élément d'un tableau

L'accès à un élément d'un tableau se fait par l'intermédiaire d'une expression entière mise entre crochets précisant l'indice de l'élément dans le tableau.

L'indice du premier élément commence toujours à 0.

Un élément de tableau est considéré comme une variable et est manipulé en tant que telle.

Exemples :

```
x[1] = x[1] + 30;           // x[1] = 20 + 30
y[2][3] = y[2][3] + y[0][0]; // y[2][3] = 400 + 1
int i = 6;
x[2] = x[i-3];             // x[2] = x[3] = 40
y[2][2] = x[2];           // y[2][2] = 40
```

5.1.5. Opérations sur les tableaux

Il n'y a pas d'affectation globale possible entre tableaux ni de comparaison possible entre 2 tableaux.

Exemple :

```
tab7 = tab1;           // illégal
if (tab1 == tab2) ...   // incorrect
if (tab1 != tab2) ...   // incorrect
```

5.1.6. Cas particulier : les chaînes de caractères

5.1.6.1. Définition

Le type prédéfini *chaîne de caractères* n'existe pas en C++. Pour manipuler des chaînes de caractères, il faut travailler sur des tableaux de caractères, ce qui peut s'avérer très contraignant.

5.1.6.2. Opérations sur les chaînes

De part la représentation interne d'une chaîne de caractères en C++ sous forme de tableau de caractères, il est impossible de faire des affectations ou des comparaisons entre chaînes. Toutefois, afin de manipuler globalement des chaînes et simplifier la tâche du programmeur, nous avons développé notre propre bibliothèque qui, au travers du type abstrait de données *Chaîne*, offre toute une panoplie de fonctionnalités (cf. *annexe 6 : les chaînes de caractères*). Cette bibliothèque est accessible par tout programme C++ en introduisant la directive :

```
#include "/usr/local/public/BIBLIOC++/chaîne.h"
```

Exemple :

```
Chaîne c1, c2;           // déclaration de 2 chaînes de caractères
c1 = uneChaîne("toto");   // c1 = "toto"
c2 = c1;                 // c2 = "toto"
c1 = concatener(c1, uneChaîne(" ")); // c1 = "toto "
c1 = concatener(c1, c2);  // c1 = "toto toto"
if (diff(c1, c2))
    ecrire(uneChaîne("chaines differentes\n")); // chaines differentes
for (int i = longueur(c2) - 1 ; i >= 0 ; i--)
    ecrire(ieme(c2, i));  // otot
```

5.2. Les enregistrements

5.2.1. Définition

Un *enregistrement* sert à regrouper un ensemble de variables pour le manipuler comme s'il s'agissait d'une seule variable.

5.2.2. Définition d'un type enregistrement

L'utilisation d'un enregistrement nécessite auparavant la définition de son type.

Une définition d'un type enregistrement, introduite par le mot réservé *struct*, consiste à fournir :

- l'identificateur du type enregistrement,
- la liste des variables (ou champs) composant l'enregistrement encadrée par des accolades.

Le type d'un champ correspond à un type prédéfini, à un type tableau ou à un autre type enregistrement.

La syntaxe est la suivante :

```
struct nom_du_type_enregistrement
{
    type1 variable1;
    type2 variable2;
    ...
    typen variableN;
} ;
```

Exemples :

```
struct Date // définition d'un enregistrement Date
{
    int jour;
    Chaine mois;
    int annee;
};

struct Individu // définition d'un enregistrement Individu
{
    Chaine nom;
    Chaine prenom;
    Date dateNaiss;
};
```

5.2.3. Déclaration d'un enregistrement

Une déclaration d'enregistrement consiste, de manière analogue à une déclaration de variable de type prédéfini, à fournir un identificateur précédé du type de l'enregistrement. Ce dernier doit avoir fait l'objet d'une définition de type (cf. § 5.2.2).

Exemples :

```
Date d1, d2; // déclaration de 2 enregistrements Date
Individu x; // déclaration d'un enregistrement Individu

typedef Date TabDate[5]; // définition d'1 tableau de 5 enreg. Date
TabDate y; // déclaration d'1 tableau de 5 enreg. Date
```

Un enregistrement peut être initialisé lors de sa déclaration en spécifiant une liste de valeurs entre accolades.

Exemples :

```
Date d3 = {3, uneChaine("Mars"), 1990};
Individu z = {uneChaine("DURAND"), uneChaine("Arthur"),
              {1, uneChaine("Fevrier"), 1980}};
```

5.2.4. Opération d'accès à un champ d'un enregistrement

L'opérateur « . » permet d'accéder à un champ d'un enregistrement.

Exemples :

```
d1.jour = 26;
d1.mois = uneChaine("Janvier");
d1.annee = 1997;

x.nom = uneChaine("DUPOND");
x.prenom = uneChaine("toto");
x.dateNaiss.jour = 12;
x.dateNaiss.mois = uneChaine("Octobre");
x.dateNaiss.annee = 1978;

y[0].jour = 20;
y[0].mois = uneChaine("Fevrier");
y[0].annee = 1956;
```

5.2.5. Opérations sur les enregistrements

5.2.5.1. Affectation

Il est possible d'affecter globalement le contenu d'un enregistrement dans un autre, à condition qu'ils soient de même type. L'affectation opère champ à champ.

Exemples :

```
d1.jour = 26;
d1.mois = uneChaine("Janvier");
d1.annee = 1950;

x.nom = uneChaine("DUPOND");
x.prenom = uneChaine("toto");
x.dateNaiss = d1;           // affectation d'une date dans une autre

y[0] = d1;                  // affectation d'une date dans une autre
y[1] = x.dateNaiss;         // affectation d'une date dans une autre
y[2] = y[1];                // affectation d'une date dans une autre
```

5.2.5.2. Comparaison

Il est impossible de comparer globalement le contenu de deux enregistrements en utilisant les opérateurs `==` et `!=`.

Exemples :

```
if (d1 == x.dateNaiss) ...           // illégal
if (y[0] != y[1]) ...                 // illégal
```

Chapitre 6 : SOUS-PROGRAMMES

6.1. Définition

C++ ne fait pas la distinction entre les procédures, les fonctions, ou encore le programme principal : tout est considéré comme fonction. Effectivement, le langage considère qu'une procédure est une fonction qui ne retourne pas de valeur et que le programme principal est une fonction particulière qui s'appelle *main*.

Tout programme C++ doit contenir une fonction *main* pour pouvoir être exécuté. Elle a la syntaxe suivante :

```
int main()
{ ... }
```

6.2. Les sous-programmes

6.2.1. Traduction des procédures

Une procédure algorithmique se traduit en C++ par une fonction qui retourne une valeur de type *void* et de syntaxe :

```
void nom(paramètres_formels)
{
    corps
}
```

- *nom* correspond au nom symbolique que l'utilisateur veut donner à la procédure. Le nom d'une procédure doit respecter les mêmes contraintes qu'un identificateur de variable.
- *paramètres_formels* correspond à la liste des paramètres formels de la procédure, chacun d'eux séparé du suivant par une virgule. Chaque paramètre doit obligatoirement être précédé de son type. Même si la procédure n'admet aucun paramètre, les parenthèses sont obligatoires.
- *corps* est encadré par un bloc { }. Il contient la déclaration des variables locales à la procédure et ses instructions.

Exemple :

Algorithme	Traduction C++
<pre>-- calcule et affiche la moyenne -- à partir de s et n procédure moyenne (entrée s <Réal>, entrée n <Entier>) glossaire m <Réal> ; -- moyenne des valeurs début si n /= 0 alors m <- s / n ; écrire ("moyenne : ") ; écrire (m) ; sinon écrire ("calcul impossible"); fin si ; fin</pre>	<pre>// calcule et affiche la moyenne // a partir de s et n void moyenne(const float s, const int n) { float m; // moyenne des valeurs if (n != 0) { m = s / float (n); écrire(uneChaine("moyenne : ")); écrire(m); } else écrire(uneChaine("calcul impossible")); }</pre>

6.2.2. Traduction des fonctions

Une fonction algorithmique se traduit en C++ par une fonction de syntaxe :

```
type nom(paramètres_formels)
{
    corps
    return (expression);
}
```

- *type* correspond au type de la valeur retournée par la fonction. Ce type peut être un type prédéfini C++ ou bien un type enregistrement.
- *nom* correspond au nom symbolique que l'utilisateur veut donner à la fonction. Le nom d'une fonction doit respecter les mêmes contraintes qu'un identificateur de variable.
- *paramètres_formels* correspond à la liste des paramètres formels de la fonction, chacun d'eux séparé par une virgule. Chaque paramètre doit obligatoirement être précédé de son type. Même si la fonction n'admet aucun paramètre, les parenthèses sont obligatoires.
- *corps* est encadré par un bloc { }. Il contient la déclaration des variables locales à la fonction et ses instructions.
- *expression* correspond à la valeur retournée par la fonction.

Une fonction algorithmique se traduit en C++ par une fonction de syntaxe :

Exemple :

Algorithme	Traduction C++
<pre>-- calcule et retourne s/n fonction moyenne (entrée s <Réal>, entrée n <Entier>) retourne <Réal> glossaire m <Réal> ; -- moyenne des valeurs début m <- s / n ; retourner(m) ; fin</pre>	<pre>// calcule et retourne n/s float moyenne(const float s, const int n) { float m ; // moyenne des valeurs m = s / float (n); return (m); }</pre>

6.2.3. Résumé

En résumé, procédures et fonctions se distinguent par rapport au type du résultat retournée par une fonction C++ :

Sous-programme algorithmique	Type du résultat de la fonction C++
procédure	void
fonction	≠ void

Comme en algorithmique, un résultat de fonction est renvoyé à l'appelant par l'instruction *return*.

6.3. Les modes de transmission des paramètres

6.3.1. Définition

En algorithmique, il existe trois utilisations possibles d'un paramètre par un sous-programme : le mode *entrée* (lecture seulement), le mode *sortie* (écriture seulement) et le mode *mise à jour* (lecture et écriture).

C++ fait également une distinction au niveau de l'utilisation des paramètres lors de leur transmission. Elle se fait dans l'en-tête de la fonction. Cependant, la syntaxe C++ diffère de celle utilisée en algorithmique et dépend du type des paramètres.

6.3.2. Paramètres de types prédéfinis ou enregistrements

Pour les paramètres de types prédéfinis ou de type enregistrement, il existe en C++ deux modes de transmission :

- la transmission *par valeur*,
- la transmission *par référence*.

S'il s'agit d'une *transmission par valeur*, le nom du paramètre formel est précédé de son type, s'il s'agit d'une transmission par référence, le nom du paramètre formel est précédé de son type suivi du symbole &.

Les règles de mode de transmission d'un paramètre de type prédéfini ou enregistrement sont fonction de son utilisation :

<i>Mode de transmission en algorithmique</i>	<i>Mode de transmission en C++</i>
entrée	par valeur précédé de <i>const</i> const <i>typeParamètre</i> <i>nomParamètre</i>
sortie	par référence <i>typeParamètre</i> &<i>nomParamètre</i>
mise à jour	par référence <i>typeParamètre</i> &<i>nomParamètre</i>

Exemple :

<i>Algorithme</i>	<i>Traduction C++</i>
<pre>-- échange le contenu des deux variables -- entières x et y procédure échanger (mise à jour x <Entier>, mise à jour y <Entier>) glossaire z <Entier> ;-- variable servant à l'échange début z <- x ; x <- y ; y <- z ; fin</pre>	<pre>// échange le contenu des deux variables // entieres x et y void echanger(int &x, int &y) { int z; // variable servant a l'echange z = x; x = y; y = z; }</pre>

6.3.3. Paramètres de type tableau

Par convention, un tableau C++ correspond à l'adresse de son premier élément. Pour cette raison, il n'existe qu'un seul mode de transmission pour un tableau : *la transmission par adresse*. Le paramètre formel est alors précédé du type du tableau.

Les règles de mode de transmission d'un paramètre tableau dépendent de son utilisation :

<i>Mode de transmission en algorithmique</i>	<i>Mode de transmission en C++</i>
entrée	par adresse précédé de <i>const</i> const <i>typeTableau</i> <i>nomTableau</i>
sortie	par adresse <i>typeTableau</i> <i>nomTableau</i>
mise à jour	par adresse <i>typeTableau</i> <i>nomTableau</i>

Exemple 1 :

<i>Algorithme</i>	<i>Traduction C++</i>
-------------------	-----------------------

<pre> type Tab100 : tableau [1 à 100] de <Entier> ; -- incrémente de 1 les éléments d'un tableau t procédure incrémenterDeUn (mise à jour t <Tab100>, entrée n <Entier>) glossaire i <Entier> ; -- indice de parcours du tableau début i <- 1 ; tantque i <= n faire t [i] <- t [i] + 1 ; i <- i + 1 ; fin tantque ; fin </pre>	<pre> typedef int Tab100[100]; // incrémente de 1 les éléments d'un tableau t void incrémenterDeUn(Tab100 t, const int n) { for (int i = 0 ; i < n ; i++) t[i] = t [i]++; } </pre>
---	--

Exemple 2 :

Algorithme	Traduction C++
<pre> type Tab100 : tableau [1 à 100] de <Entier> ; -- recherche dans le tableau d'entiers t de -- n éléments la première occurrence de -- la valeur x -- renvoie dans trouvé le résultat de la -- recherche et renvoie dans r le rang de la -- première occurrence si trouve = vrai procédure rechercher (entrée t <Tab100>, entrée n <Entier>, entrée x <Entier>, sortie r <Entier>, sortie trouvé <Booléen>) glossaire i <Entier> ; -- indice de parcours du tableau début i <- 1 ; trouvé <- faux ; tantque non trouve et i <= n faire si t[i] = x alors trouvé <- vrai ; r <- i ; sinon i <- i + 1 ; fin si ; fin tantque ; fin </pre>	<pre> typedef int Tab100[100]; // recherche dans le tableau d'entiers t de // n elements la première occurrence de // la valeur x // renvoie dans trouve le resultat de la // recherche et renvoie dans r le rang de la // première occurrence si trouve = vrai void rechercher(const Tab100 t, const int n, const int x, int &r, bool &trouve) { int i; // indice de parcours du tableau i = 0; trouve = false ; while (!trouve && i < n) if (t[i] == x) { trouve = true ; r = i ; } else i++ ; } </pre>

Remarque : le mot réservé *const* précédant le paramètre de type tableau permet au compilateur de vérifier que les éléments du tableau ne sont pas modifiés par la fonction C++. On retrouve ainsi la sémantique du mode *entrée* du langage algorithmique.

Les différents modes de transmission de paramètres seront étudiés en détail dans la troisième partie du cours C++ (*Pointeurs et allocation dynamique*).

6.4. Appel d'une fonction C++

Un appel de fonction C++ se fait en précisant simplement le nom de la fonction suivie éventuellement de la liste des paramètres effectifs entre parenthèses.

Exemple : Séquence d'appel de la fonction C++ *rechercher* de l'exemple précédent

```

...
typedef int Tab100[100];
...
bool trouve;
Tab100 notes;
int nbNotes, rang;
...
rechercher(notes, nbNotes, 20, rang, trouve);
...

```

Remarque :

Un appel de fonction C++ n'admettant aucun paramètre se fait toujours en précisant le nom de la fonction suivi de `()`.

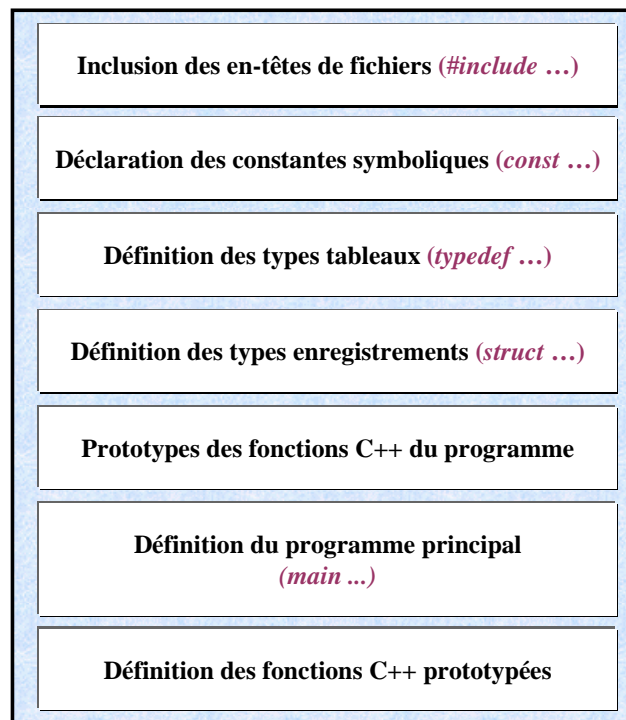
6.5. Décomposition d'un programme en sous-programmes

Une application C++ sera organisée en plusieurs fonctions. Ces fonctions représentent des blocs. Il doit exister obligatoirement un bloc correspondant au programme principal et introduit par le mot clé *main*.

Les variables déclarées au début du programme sont des variables globales et sont donc accessibles dans toutes les fonctions du programme. On limitera la déclaration de variables globales à celles de constantes symboliques introduites par le mot-clé *const* (sauf cas particuliers).

Par contre, en début de programme devront figurer les directives d'inclusion des fichiers d'en-tête (*#include ...*), les déclarations de type tableau (*typedef ...*), et les déclarations de type enregistrement (*struct ...*).

En règle générale, un programme C++ sera organisé de la manière suivante :



Remarque :

Une convention d'écriture des programmes en C++ a été adoptée à l'IUT (*cf. fascicule*).

Un exemple de programme C++ est fourni en *annexe 8*.

Chapitre 7 : EXCEPTIONS

7.1. Définition

En algorithmique, une *exception* correspond à une situation exceptionnelle à laquelle le programme ou le sous-programme ne peut répondre. Lorsqu'une exception est déclenchée, elle provoque un déroutement conduisant à l'arrêt du programme.

Le langage C++ possède son propre mécanisme pour signaler et traiter les exceptions. Il permet de traiter toute condition inhabituelle, mais prévisible, qui peut se produire lors de l'exécution d'un programme. Il utilise les mots-clés *throw*, *catch* et *try*.

7.2. Levée d'une exception

La syntaxe d'un déclenchement d'exception est la suivante :

```
throw expression;
```

L'expression peut être une chaîne de caractères indiquant la nature de l'anomalie qui s'est produite.

Toute fonction C++ déclenchant une ou plusieurs exceptions doit explicitement le signaler dans son entête en utilisant la syntaxe suivante :

```
type_fonction nom_fonction(paramètres_formels) throw (type_exception)
```

type_exception correspond au type de l'expression associée à l'exception déclenchée.

Exemple :

Algorithme	Traduction C++
<pre>-- calcule la moyenne des nbNotes notes d'un -- tableau fonction moyenneDesNotes (entrée notes <Tab100>, entrée nbNotes <Entier>) retourne <Réal> déclenche aucuneNote glossaire somme <Réal> ; -- somme des notes i <Entier> ; -- indice de parcours du tableau début somme <- 0 ; i <- 1 ; -- calculer la somme des notes tantque i <= nbNotes faire somme <- somme + notes[i] ; i <- i + 1 ; fin tantque ; -- lever l'exception si nbNotes = 0 alors déclencher(aucuneNote) ; fin si ; -- calculer la moyenne des notes retourner (somme / nbNotes) ; fin</pre>	<pre>typedef float Tab100[100]; // calcule la moyenne des nbNotes notes d'un // tableau float moyenneDesNotes(const Tab100 notes, const int nbNotes) throw (Chaine) { float somme = 0; // somme des notes // calculer la somme des notes for (int i=0 ; i<nbNotes ; i++) somme = somme + notes[i]; // lever l'exception if (nbNotes == 0) throw uneChaine ("AUCUNE NOTE"); // calculer la moyenne des notes return (somme / nbNotes); }</pre>

7.3. Interception d'une exception

Si, une exception est déclenchée, un traitement par défaut appelé *terminate()* s'exécute. Celui-ci provoque l'arrêt de l'exécution du programme.

Toutefois, un sous-programme a la possibilité d'intercepter l'exception et de la traiter en effectuant un traitement particulier utilisant les mots-clés *try* et *catch*.

Le traitement des exceptions sera étudié plus tard.

Dans les programmes à réaliser, nous nous contenterons de gérer les cas d'anomalie par des déclenchements d'exception sans associer de traitement particulier.

Chapitre 8 : FONCTION SURCHARGÉE ET FONCTION GÉNÉRIQUE

8.1. Fonction surchargée

Dans une même application, il est possible en C++ que plusieurs fonctions possèdent le même nom symbolique. On dit alors que ces fonctions sont *surchargées*. Le choix de la fonction utilisée par le système à l'exécution dépend alors du type et du nombre des arguments à l'appel.

Il est intéressant de remarquer que la surcharge a été utilisée pour les fonctions *lire()* et *ecrire()* utilisées dans *entréeSortie.h* (cf. annexe 5).

Exemple :

```
// calcule la somme de 2 réels
float somme(const float x, const float y);

// calcule la somme de 2 entiers
int somme(const int x, const int y);

int main()
{
    int a = 3, b = 4;
    float c = 2.5, d = 1.0;
    écrireNL(somme(a, b));           // 7
    écrireNL(somme(c, d), 2);       // 3.50
}

float somme(const float x, const float y)
{
    return (x + y);
}

int somme(const int x, const int y)
{
    return (x + y);
}
```

8.2. Fonction générique

Une fonction générique est une fonction qui ne dépend pas du type des paramètres passés à la fonction. Un mécanisme très puissant est prévu en C++ pour pouvoir créer des fonctions génériques tout en conservant le contrôle de type : c'est le concept de "*patron de fonction*".

Pour créer une fonction générique, l'en-tête de la fonction doit être précédée de :

```
template <class identificateur_de_type_générique>
```

Le choix de l'identificateur de type générique est laissé au programmeur. Il représente un type de donnée formel qui devra être substitué au moment de l'appel par un type de donnée réel. En règle générale, l'identificateur de type est nommé T.

Exemple :

La permutation de deux valeurs entières et deux valeurs réelles nécessite l'écriture de deux fonctions distinctes dont les instructions sont identiques excepté le type des variables. L'écriture d'une fonction générique *permuter()* permet d'éviter cette redondance de code :

```

// permet de permuter 2 valeurs de même type
template <class T>
void permuter(T &a, T &b);

int main()
{
    int x = 5, y = 8;
    float z = 2.5, t = 8.9;
    permuter(x, y);
    permuter(z, t);
}

template <class T>
void permuter(T &a, T &b)
{
    T c;
    c = a;
    a = b;
    b = c;
}

```

Remarques : le compilateur détecte une erreur si à l'appel de la fonction les deux paramètres sont de types différents.

Pour utiliser des types génériques différents dans une fonction générique, il suffit de fournir des identificateurs de type générique différents.

Exemple :

```

template <class T1, class T2>
void fonction (const T1 a, const T2 b)
{ ... }

```

ANNEXES

ANNEXE 1 : Les mots réservés du langage

and	and_eq	asm	auto	bitand
bitor	bool	break	case	catch
char	class	compl	const	const_cast
continue	default	delete	do	double
dynamic_cast	else	enum	explicit	extern
false	float	for	friend	goto
if	inline	int	long	mutable
namespace	new	not	not_eq	operator
or	or_eq	private	protected	public
register	reinterpret_cast	return	short	signed
sizeof	static	static_cast	struct	switch
template	this	throw	true	try
typedef	typeid	typename	union	unsigned
using	virtual	void	volatile	wchar_t
while	xor	xor_eq		

ANNEXE 2 : Les types prédéfinis

Type C++	Signification	Taille	Valeurs
bool	Booléen	1 octet	true (= 1) ou false (= 0)
char	Caractère	1 octet	256 caractères code ASCII
short	Entier court	2 octets	De -32 768 à 32 767
unsigned short	Entier court non signé	2 octets	De 0 à 65 535
int	Entier court ou long	2 octets ou 4 octets	
unsigned int	Entier court ou long non signé	2 octets ou 4 octets	
long	Entier long	4 octets	De -2 147 483 648 à 2 147 483 647
unsigned long	Entier long non signé	4 octets	De 0 à 4 294 967 295
float	Réel	4 octets	De 1,2 e-38 à 3,4 e38
double	Réel double précision	8 octets	De 2,2 e-308 à 1,8 e308

ANNEXE 3 : Les principaux caractères spéciaux

<i>Caractère</i>	<i>Signification</i>
<code>\n</code>	Nouvelle ligne (retour chariot et saut de ligne)
<code>\b</code>	Retour arrière
<code>\t</code>	Tabulation horizontale
<code>\f</code>	Saut de page
<code>\a</code>	Signal sonore
<code>\"</code>	Guillemets
<code>\'</code>	Quote
<code>\0</code>	Caractère nul
<code>\\</code>	antislash

ANNEXE 4 : Les principaux opérateurs

<i>Priorité</i>	<i>Opérateurs</i>
8	<code>++ -- -(unaire) +(unaire)</code>
7	<code>* / %</code>
6	<code>+(binaire) -(binaire)</code>
5	<code>< > <= >=</code>
4	<code>== !=</code>
3	<code>&&</code>
2	<code> </code>
1	<code>=</code>

ANNEXE 5 : Les entrées/sorties

Afin de simplifier l'utilisation des entrées/sorties pour les types prédéfinis et le type *Chaine*, nous avons développé une bibliothèque de fonctions C++ spécifiques. Son utilisation nécessite l'inclusion du fichier d'en-tête :

"usr/local/public/BIBLIOC++/entreeSortie.h"

```
//-----  
// SPECIFICATION DES SOUS-PROGRAMMES DE LECTURE SUR L'ENTREE STANDARD  
// CHAQUE LECTURE DOIT ETRE VALIDEE PAR L'APPUI SUR LA TOUCHE <ENTREE>  
//-----  
  
// attend l'appui sur la touche <ENTREE>  
void lire();  
  
// lit un booléen v  
// si la valeur lue est 1, le booléen v prend la valeur true  
// si la valeur lue est 0, le booléen v prend la valeur false  
// leve l'exception "LECTURE BOOLEEN INCORRECTE" si la saisie est incorrecte  
void lire(bool &v) throw (Chaine);  
  
// lit un caractere v  
// leve l'exception "LECTURE CARACTERE INCORRECTE" si la saisie est incorrecte  
void lire(char &v) throw (Chaine);  
  
// lit un entier court v  
// leve l'exception "LECTURE ENTIER COURT INCORRECTE" si la saisie est  
// incorrecte  
void lire(short &v) throw (Chaine);  
  
// lit un entier court non signe v  
// leve l'exception "LECTURE ENTIER COURT NON SIGNE INCORRECTE" si la saisie  
// est incorrecte  
void lire(unsigned short &v) throw (Chaine);  
  
// lit un entier v  
// leve l'exception "LECTURE ENTIER INCORRECTE" si la saisie est incorrecte  
void lire(int &v) throw (Chaine);  
  
// lit un entier non signe  
// leve l'exception "LECTURE ENTIER NON SIGNE INCORRECTE" si la saisie est  
// incorrecte  
void lire(unsigned int &v) throw (Chaine);  
  
// lit un entier long v  
// leve l'exception "LECTURE ENTIER LONG INCORRECTE" si la saisie est incorrecte  
void lire(long &v) throw (Chaine);  
  
// lit un entier long non signe v  
// leve l'exception "LECTURE ENTIER LONG NON SIGNE INCORRECTE" si la saisie est  
// incorrecte  
void lire(unsigned long &v) throw (Chaine);  
  
// lit un reel v  
// leve l'exception "LECTURE REEL INCORRECTE" si la saisie est incorrecte  
void lire(float &v) throw (Chaine);  
  
// lit un reel double precision v  
// leve l'exception "LECTURE REEL DOUBLE PRECISION INCORRECTE" si la saisie est  
// incorrecte  
void lire(double &v) throw (Chaine);  
  
// lit une chaine ch sur l'entree standard  
// leve l'exception "LECTURE CHAINE INCORRECTE" si la saisie est incorrecte  
void lire(Chaine &ch) throw (Chaine);
```

```

//-----
// SPECIFICATION DES SOUS-PROGRAMMES D'ECRITURE SUR LA SORTIE STANDARD
// LES IDENTIFICATEURS DE SOUS-PROGRAMMES SE TERMINANT PAR NL POSITIONNENT
// LE CURSEUR AU DEBUT DE LA LIGNE SUIVANTE APRES AVOIR EFFECTUE L'ECRITURE
//-----

// positionne le curseur en debut de ligne suivante
void ecrireNL();

// ecrit un booléen v
// si v a pour valeur true, la valeur affichée est égale à 1
// si v a pour valeur false, la valeur affichée est égale à 0
void ecrire(const bool v);
void ecrireNL(const bool v);

// ecrit un caractère v
void ecrire(const char v);
void ecrireNL(const char v);

// ecrit un entier court v
void ecrire(const short v);
void ecrireNL(const short v);

// ecrit un entier court non signé v
void ecrire(const unsigned short v);
void ecrireNL(const unsigned short v) ;

// ecrit un entier v
void ecrire(const int v);
void ecrireNL(const int v) ;

// ecrit un entier non signé v
void ecrire(const unsigned int v);
void ecrireNL(const unsigned int v);

// ecrit un entier long v
void ecrire(const long v);
void ecrireNL(const long v);

// ecrit un entier long non signé v
void ecrire(const unsigned long v);
void ecrireNL(const unsigned long v);

// ecrit un réel v avec nb chiffres après le point decimal
void ecrire(const float v, const int nb);
void ecrireNL(const float v, const int nb);

// ecrit un réel double précision v avec nb chiffres après le point decimal
void ecrire(const double v, const int nb);
void ecrireNL(const double v, const int nb);

// ecrit une chaîne ch sur la sortie standard
void ecrire(const Chaine ch);
void ecrireNL(const Chaine ch);

```

```

//-----
// SPECIFICATION DES SOUS-PROGRAMMES DE GESTION DE L'AFFICHAGE ECRAN
//-----

// remet a blanc l'ecran et positionne le curseur en haut a gauche de l'ecran
void effacer();

// positionne l'affichage en mode video inverse
void inverser();

// positionne l'affichage en mode normal
void normal();

// positionne l'affichage en gras
void gras();

// emet un beep sonore
void beep();

// positionne le curseur en ligne l et colonne c
void allerLC(const int l, const int c);

```

ANNEXE 6 : Les chaînes de caractères

Pour simplifier l'utilisation des chaînes de caractères, nous avons développé une bibliothèque de fonctions C++ spécifiques. Son utilisation nécessite l'inclusion du fichier d'en-tête :

"usr/local/public/BIBLIOC++/chaine.h"

```
//-----  
// SPECIFICATION DES SOUS-PROGRAMMES DE MANIPULATION DU TYPE ABSTRAIT DE  
// DONNEE Chaîne  
//-----  
  
// constante designant une chaîne vide  
Chaîne CHAÎNE_VIDE();  
  
// construit une chaîne a partir d'un tableau de caracteres terminé par '\n'  
// leve l'exception "LONGUEUR INVALIDE" si la chaîne resultat depasse  
// 255 caracteres  
Chaîne uneChaîne(const TabCar t) throw (Chaîne);  
  
// renvoie le nombre de caracteres effectif d'une chaîne ch  
int longueur(const Chaîne ch);  
  
// renvoie une chaîne resultant de la concatenation de ch2 a ch1  
// leve l'exception "LONGUEUR INVALIDE" si la chaîne resultat  
// depasse 255 caracteres  
Chaîne concatener (const Chaîne ch1, const Chaîne ch2) throw (Chaîne);  
  
// renvoie le resultat de l'expression ch1 == ch2  
bool egal(const Chaîne ch1, const Chaîne ch2);  
  
// renvoie le resultat de l'expression ch1 < ch2 (ordre lexicographique)  
bool inf(const Chaîne ch1, const Chaîne ch2);  
  
// renvoie le resultat de l'expression ch1 > ch2 (ordre lexicographique)  
bool sup(const Chaîne ch1, const Chaîne ch2);  
  
// renvoie le resultat de l'expression ch1 != ch2  
bool diff(const Chaîne ch1, const Chaîne ch2);  
  
// renvoie le resultat de l'expression ch1 <= ch2 (ordre lexicographique)  
bool infEgal(const Chaîne ch1, const Chaîne ch2);  
  
// renvoie le resultat de l'expression ch1 >= ch2 (ordre lexicographique)  
bool supEgal(const Chaîne ch1, const Chaîne ch2);  
  
// renvoie le caractere de rang i d'une chaîne ch  
// l'indice du premier caractere a pour valeur 0  
// leve l'exception "RANG INVALIDE" si i < 0 ou si i >= longueur (ch)  
char ieme(const Chaîne ch, const int i) throw (Chaîne);  
  
// remplace le caractere de rang i d'une chaîne ch  
// par le caractère c  
// l'indice du premier caractere a pour valeur 0  
// leve l'exception "RANG INVALIDE" si i < 0 ou si i >= longueur (ch)  
char changerIeme(Caîne &ch, const char c, const int i) throw (Chaîne);
```

ANNEXE 7 : table des codes ASCII

Le système de codage ASCII (American Standard Code for Information Interchange) associe à chaque caractère un entier entre 0 et 127.

Les 32 premiers caractères sont des caractères non imprimables.

<i>Caractère</i>	<i>Valeur Décimale</i>	<i>Caractère</i>	<i>Valeur Décimale</i>	<i>Caractère</i>	<i>Valeur Décimale</i>	<i>Caractère</i>	<i>Valeur Décimale</i>
Ctrl @	0		32	@	64	,	96
Ctrl A	1	!	33	A	65	a	97
Ctrl B	2	"	34	B	66	b	98
Ctrl C	3	#	35	C	67	c	99
Ctrl D	4	\$	36	D	68	d	100
Ctrl E	5	%	37	E	69	e	101
Ctrl F	6	&	38	F	70	f	102
\a	7	'	39	G	71	g	103
\b	8	(40	H	72	h	104
\t	9)	41	I	73	i	105
\n	10	*	42	J	74	j	106
\v	11	+	43	K	75	k	107
\f	12	,	44	L	76	l	108
\r	13	-	45	M	77	m	109
Ctrl N	14	.	46	N	78	n	110
Ctrl O	15	/	47	O	79	o	111
Ctrl P	16	0	48	P	80	p	112
Ctrl Q	17	1	49	Q	81	q	113
Ctrl R	18	2	50	R	82	r	114
Ctrl S	19	3	51	S	83	s	115
Ctrl T	20	4	52	T	84	t	116
Ctrl U	21	5	53	U	85	u	117
Ctrl V	22	6	54	V	86	v	118
Ctrl W	23	7	55	W	87	w	119
Ctrl X	24	8	56	X	88	x	120
Ctrl Y	25	9	57	Y	89	y	121
Ctrl Z	26	:	58	Z	90	z	122
Ctrl [27	;	59	[91	{	123
Ctrl /	28	<	60	\	92		124
Ctrl]	29	=	61]	93	}	125
Ctrl ^	30	>	62	^	94	~	126
Ctrl _	31	?	63	_	95	Delete	127

ANNEXE 8 : Exemple de programme C++

```

//*****
// ROLE DU PROGRAMME      | rembourse plusieurs dettes a partir de billets //
//                         | disponibles en caisse //
//-----|-----
// FICHIER SOURCE          | /usr/public/mesProgrammes/dette.C //
//-----|-----
// LANGAGE                 | C++ //
//-----|-----
// AUTEUR(S)              | Christine JULIEN //
//-----|-----
// DATE CREATION           | 10/10/1998 //
// DATE MISE A JOUR        | 24/10/2008 //
//*****

//-----
// IMPORTATION DES BIBLIOTHEQUES UTILISEES
//-----
#include "/usr/local/public/BIBLIOC++/entreeSortie.h"
#include "/usr/local/public/BIBLIOC++/chaine.h"

//-----
// DEFINITION DES TYPES ENREGISTREMENT
//-----
// definition du type Caisse
struct Caisse
{
    int n50;      // nombre de billets de 50 euros disponibles
    int n20;      // nombre de billets de 20 euros disponibles
    int n10;      // nombre de billets de 10 euros disponibles
};

//-----
// SPECIFICATION DES SOUS-PROGRAMMES
//-----
// lit 3 entiers representant respectivement le nombre de billets de
// 50, 20 et 10 euros et les range dans caisse
void lireCaisse(Caisse &caisse);

// calcule et retourne la somme disponible dans caisse
int sommeCaisse(const Caisse caisse);

// compteur prend pour valeur initiale valeur
void initialiserCompteur(int &compteur, const int valeur);

// compteur devient egal a compteur + 1
void augmenterDeUn(int &compteur);

// retourne vrai si les valeurs compteur1 et compteur2 sont egales et faux sinon
bool egauxCompteurs(const int compteur1, const int compteur2);

// calcule la nouvelle dette d restant a payer en fonction du nombre n de
// billets disponibles en caisse d'une valeur donnee ;
// determine le nombre nb de billets delivres de cette valeur
// en remboursement de tout ou partie de d
void calculerDetteRestante(int &d, const int n, const int valeur, int &nb);

```

```

// rembourse une dette d en delivrant nb50, nb20, nb10 billets
// de valeurs respectives 50, 20 et 10 euros
// a partir de caisse ayant n50, n20 et n10 billets
// de valeurs respectives 50, 20 et 10 euros
void rembourserUneDette
    (int &d, const Caisse caisse, int &nb50, int &nb20, int &nb10);

// met a jour le montant caisse du debiteur en fonction du nombre de
// billets de 50 (nb50), 20 (nb20) et 10 (nb10) euros delivres
// pour rembourser une dette
void mettreAJourCaisse
    (Caisse &caisse, const int nb50, const int nb20, const int nb10);

//-----
// PROGRAMME PRINCIPAL
//-----
int main()
{
    Caisse maCaisse;// enregistrement correspondant a la caisse
    int uneDette;    // montant d'une dette
    int x50;         // nombre de billets de 50 euros delivres pour une dette
    int x20;         // nombre de billets de 20 euros delivres pour une dette
    int x10;         // nombre de billets de 10 euros delivres pour une dette
    int nbDettes;    // nombre de dettes a rembourser
    int cptDettes;   // nombre de dettes examinees

    effacer();
    ecrireNL(uneChaine("    **** REMBOURSEMENT DE PLUSIEURS DETTES ****"));
    ecrireNL();
    ecrireNL();
    // lire le nombre de billets disponible en caisse
    lireCaisse(maCaisse);
    // calculer et ecrire la somme disponible en caisse
    ecrire(uneChaine("Montant de la somme disponible en caisse : "));
    ecrireNL(sommeCaisse (maCaisse));
    ecrireNL();
    // lire le nombre de dettes a rembourser
    ecrire(uneChaine("Entrer le nombre de dettes a rembourser : "));
    lire(nbDettes);
    // rembourser toutes les dettes
    for
        ( initialiserCompteur (cptDettes, 0) ;
          !egauxCompteurs (cptDettes, nbDettes) ;
          augmenterDeUn (cptDettes))
    {
        ecrireNL();
        // lire le montant de la dette a rembourser
        ecrire(uneChaine("Entrer le montant de la dette a rembourser : "));
        lire(uneDette);
        // rembourser cette dette
        rembourserUneDette(uneDette, maCaisse, x50, x20, x10);
        // ecrire le nombre de billets de 50, 20, et 10 euros delivres
        ecrire(uneChaine("nombre de billets de 50 euros delivres : "));
        ecrireNL(x50);
        ecrire(uneChaine("nombre de billets de 20 euros delivres : "));
        ecrireNL(x20);
        ecrire(uneChaine("nombre de billets de 10 euros delivres : "));
        ecrireNL(x10);
        // ecrire le montant de la dette restant a rembourser
        ecrire(uneChaine("Montant de la dette restant a rembourser : "));
        ecrireNL(uneDette);
        // mettre a jour le montant de la caisse
        mettreAJourCaisse(maCaisse, x50, x20, x10);
        // calculer et ecrire la somme disponible en caisse
        ecrire(uneChaine("Montant de la somme disponible en caisse : "));
        ecrireNL(sommeCaisse(uneCaisse));
    }
    ecrireNL();
    ecrireNL(uneChaine("Fin de remboursement des dettes"));
}

```

```

//-----
// DEFINITION DES SOUS-PROGRAMMES
//-----

void lireCaisse(Caisse &caisse)
{
    ecrire(uneChaine("Entrer le nombre de billets de 50 euros disponibles : "));
    lire(caisse.n50);
    ecrire(uneChaine("Entrer le nombre de billets de 20 euros disponibles : "));
    lire(caisse.n20);
    ecrire(uneChaine("Entrer le nombre de billets de 10 euros disponibles : "));
    lire(caisse.n10);
}

int sommeCaisse(const Caisse caisse)
{
    return (caisse.n500 * 50 + caisse.n20 * 20 + caisse.n10 * 10 ;
}

void initialiserCompteur(int &compteur, const int valeur)
{
    compteur = valeur;
}

void augmenterDeUn(int &compteur)
{
    compteur++;
}

bool egauxCompteurs(const int compteur1, const int compteur2)
{
    return (compteur1 == compteur2);
}

void calculerDetteRestante(int &d, const int n, const int valeur, int &nb)
{
    // calculer le nombre de billets de valeur "valeur" a rembourser
    // s'il y a suffisamment de billets en caisse
    nb = d / valeur;
    // reajuster le nombre de billets de valeur "valeur" a rembourser
    // s'il ne reste pas assez de billets en caisse
    if (nb > n)
    {
        nb = n;
    }
    // mettre a jour le montant de la dette a rembourser
    d = d - (nb * valeur);
}

void rembourserUneDette
(int &d, const Caisse caisse, int &nb50, int &nb20, int &nb10)
{
    // calculer la dette apres remboursement avec des billets de 50 euros
    calculerDetteRestante(d, caisse.n50, 50, nb50);
    // calculer la dette apres remboursement avec des billets de 20 euros
    calculerDetteRestante(d, caisse.n20, 20, nb20);
    // calculer la dette apres remboursement avec des billets de 10 euros
    calculerDetteRestante(d, caisse.n10, 10, nb10);
}

void mettreAJourCaisse
(Caisse &caisse, const int nb50, const int nb20, const int nb10)
{
    caisse.n50 = caisse.n50 - nb50;
    caisse.n20 = caisse.n20 - nb20;
    caisse.n10 = caisse.n10 - nb10;
}

```


Exemple d'exécution :

```
**** REMBOURSEMENT DE PLUSIEURS DETTES ****

Entrer le nombre de billets de 50 euros disponibles : 12
Entrer le nombre de billets de 20 euros disponibles : 10
Entrer le nombre de billets de 10 euros disponibles : 3
Montant de la somme disponible en caisse : 830

Entrer le nombre de dettes a rembourser : 3

Entrer le montant de la dette a rembourser : 120
nombre de billets de 50 euros delivres : 2
nombre de billets de 20 euros delivres : 1
nombre de billets de 10 euros delivres : 0
Montant de la dette restant a rembourser : 0
Montant de la somme disponible en caisse : 710

Entrer le montant de la dette a rembourser : 200
nombre de billets de 50 euros delivres : 4
nombre de billets de 20 euros delivres : 0
nombre de billets de 10 euros delivres : 0
Montant de la dette restant a rembourser : 0
Montant de la somme disponible en caisse : 510

Entrer le montant de la dette a rembourser : 30
nombre de billets de 50 euros delivres : 0
nombre de billets de 20 euros delivres : 1
nombre de billets de 10 euros delivres : 1
Montant de la dette restant a rembourser : 0
Montant de la somme disponible en caisse : 480

Fin de remboursement des dettes
```