Université de Caen Basse-Normandie
U.F.R. des Sciences
Département d'informatique

Bâtiment Sciences 3 - Campus Côte de Nacre
F-14032 Caen Cédex, FRANCE

Contrôle continu 2026-2027

Niveau	**M1**
Parcours	**Informatique - cybersécurité**
Unité d'enseignement	**MCYBFI1B - Programmation C/C++ sécurisée**
Responsable	**Emmanuel Cagniot** **emmanuel.cagniot@ensicaen.fr**

1 Exercice

Ce devoir est à effectuer de préférence en binôme. La réponse, une archive `devoir.tgz` *contenant le code source, devra être postée par l'un des étudiants (un seul post) sur e-campus La deadline est fixée au vendredi* 20 *novembre* 2026, 23*h*59 *(le dépôt sera clôturé à l'issue et plus aucun devoir ne sera accepté, même par e-mail). Vous apporterez un soin particulier aux conventions de nommage des différentes fonctions, aux commentaires ainsi qu'à la façon de déclarer vos couples pointeurs-pointés.*

Un compilateur n'est rien d'autre qu'un outil de ré-écriture *source to source* : il prend un texte rédigé dans une grammaire particulière en entrée et le retranscrit en sortie dans une autre grammaire. Bien souvent, le langage cible est un assembleur pour un processeur ou une famille de processeurs.

Comme les architectures ciblées peuvent être nombreuses (voire encore inconnues), un compilateur ne génère pas directement du code assembleur mais du *byte code* c'est à dire un langage assembleur intermédiaire pour un processeur imaginaire.

Ce *byte code* peut ensuite être transformé en un langage assembleur pour un architecture donnée (en exploitant toutes ses caractéristiques). Il peut également être interprété : c'est par exemple le cas du langage JAVA où le *byte code* issu de la compilation est exécuté par un interpréteur appelé `JVM` (*Java Virtual Machine*).

Une équipe de votre société est en train d'écrire un compilateur permettant de compiler des expressions entières préfixées telles que (+ (∗ 2 5) (+ 3 (/ 5 4))) où le signe des opérations binaires est placé devant les opérandes. Ces expressions obéissent à la grammaire algébrique de la figure 1.1.

Ces expressions sont traduites dans un assembleur pour un processeur imaginaire seulement pourvu d'une pile. Ses mnémoniques sont :

Mne	Signification
`PUSH e`	Empile l'entier `e`
`PRINT`	Dépile l'entier en sommet de pile et affiche sa valeur sur la sortie standard.
`ADD`	Dépile les entiers `e2` puis `e1` et empile la somme `e1+e2`
`SUB`	Dépile les entiers `e2` puis `e1` et empile la différence `e1-e2`
`MUL`	Dépile les entiers `e2` puis `e1` et empile `e1*e2`
`DIV`	Dépile les entiers `e2` puis `e1` et empile `e1/e2`

```
Expression       ::=  Entier | '(' Opérateur Expression Expression ')'
Entier           ::=  '0' | Chiffre_Positif { Chiffre }
Chiffre          ::=  '0' | Chiffre_Positif
Chiffre_Positif  ::=  '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
Opérateur        ::=  '+' | '-' | '*' | '/'
```

FIGURE 1.1 – Grammaire algébrique décrivant des expressions entières préfixées.

La figure 1.2 présente le résultat de la compilation de l'expression (+ (∗ 2 5) (+ 3 (/ 5 4))) dans cet assembleur.

```
1  PUSH    2
2  PUSH    5
3  MUL
4  PUSH    3
5  PUSH    5
6  PUSH    4
7  DIV
8  ADD
9  ADD
10 PRINT
```

FIGURE 1.2 – Expression (+ (∗ 2 5) (+ 3 (/ 5 4))) compilée.

Votre équipe est chargée de développer en langage C un interpréteur pour cet assembleur. Comme de nouvelles mnémoniques peuvent rapidement apparaître ou disparaître, vous décidez d'implémenter cet interpréteur dans le paradigme de la programmation par objets, ce qui vous donne accès aux *design patterns* [1] dont deux vont se révéler très utiles : `Interpreter` et `Visitor`.

Pour cette première version de l'interpréteur vous décidez de ne pas gérer les problèmes de manque de mémoire (les allocations dynamiques sont supposées ne jamais échouer).

Le *pattern* `Interpreter` nécessitera une pile d'entiers relatifs, pile que vous modélisez par le diagramme de classes de la figure 1.3.

La classe non dérivable `Stack` propose un constructeur par défaut permettant de créer une pile vide. La méthode `size` retourne le nombre d'éléments de la pile tandis que la méthode `empty` indique si elle est vide. L'ajout d'un élément en sommet de pile s'effectue via la méthode `push` (la pile est un conteneur dynamique c'est à dire que sa capacité d'accueil n'est pas fixe et peut évoluer dans le temps). La méthode `top` retourne la valeur de l'élément en sommet de pile tandis que la méthode `pop` le supprime. Enfin, la méthode `clear` vide la pile de tous ses éléments.

1.1 Question

Écrivez la classe `Stack` en vous servant de la fonction `realloc` de la bibliothèque standard pour gérer le comportement dynamique du conteneur.

Le diagramme de la figure 1.4 présente les interfaces et classes relatives aux mnémoniques que vous allez écrire dans ce devoir.

L'interface `Mne`, qui modélise une mnémonique, implémente les *design patterns* `Interpreter` (méthode `interpret`) et `Visitor` (méthode `accept`).

La méthode `interpret` permet à une mnémonique de s'interpréter dans le contexte représenté par son paramètre `stack`.

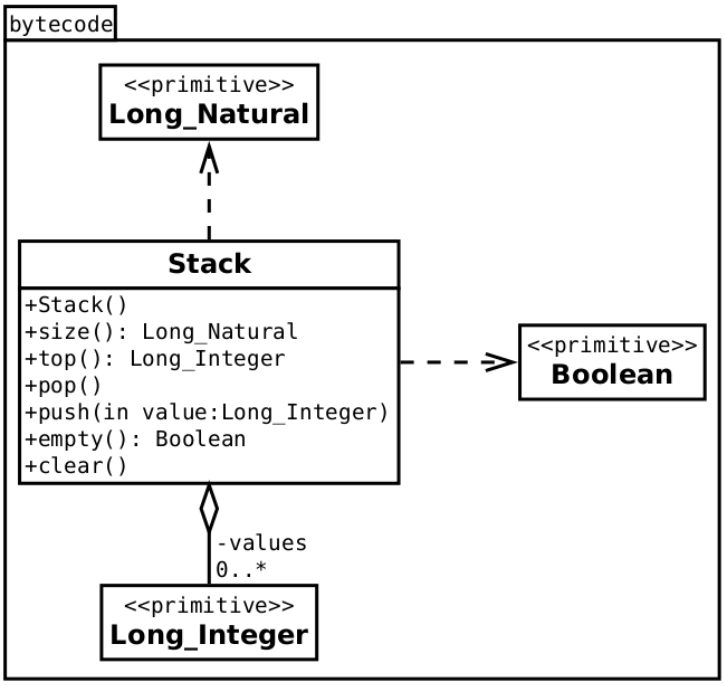


FIGURE 1.3 – Classe `Stack` modélisant une pile d'entiers relatifs.

Le *pattern* `Interpreter` oblige à prévoir à l'avance toutes les opérations applicables à une structure de données, ce qui n'est pas toujours possible. Le *pattern* `Visitor` représente une porte laissée ouverte sur cette structure afin de pouvoir implémenter à posteriori des opérations non prévues initialement. Cette porte est représentée par la méthode `accept` et son paramètre `visitor` à qui la mnémonique demande d'effectuer une action sur elle en appelant la méthode qui la concerne.

1.2 Question

Écrivez l'interface `Mne`.

La classe dérivable `Visitor` propose une méthode par mnémonique pouvant être rencontrée et un comportement par défaut (ne rien faire) pour chaque. Ce comportement permet aux futures classes dérivées de ne redéfinir que la ou les méthodes relatives aux mnémoniques ciblées et pas les autres.

1.3 Question

Écrivez la classe `Visitor`.

1.4 Question

Écrivez les classes non dérivables `Print`, `Add` et `Push` qui représentent des mnémoniques.

Le diagramme de la figure 1.5 présente la classe dérivable `Program` qui modélise un code rédigé dans notre assembleur. Son constructeur par défaut produit un code vide de mnémonique tandis que sa méthode `clear` vide un code de toutes ses mnémoniques. Les méthodes `new_print`, `new_add` et `new_push` ajoutent respectivement une mnémonique `Print`, `Add` et `Push` en fin de code. Enfin, nous considérons (c'est un choix) qu'un code assembleur est lui-même une super mnémonique en implémentant l'interface `Mne`.

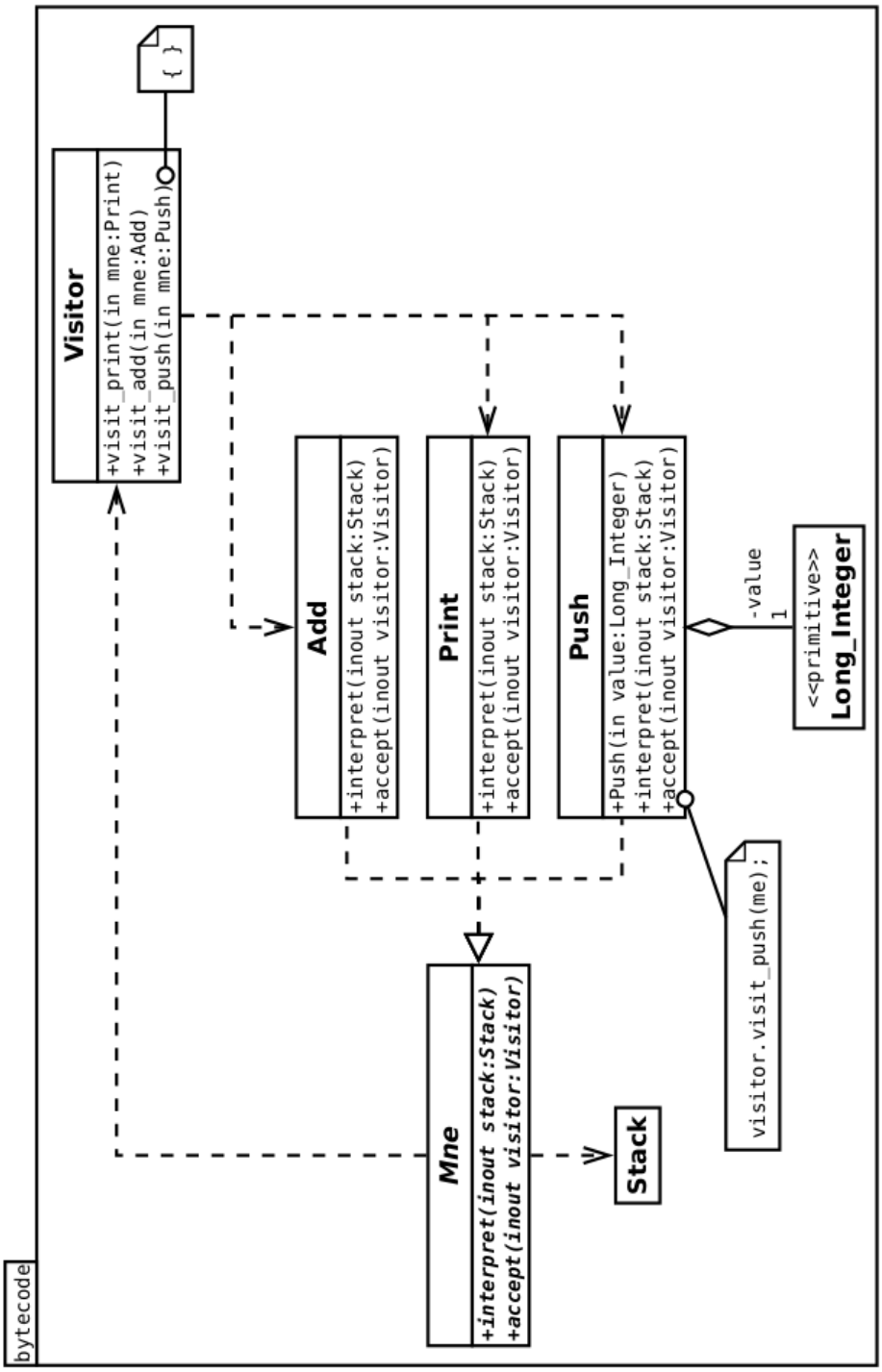


FIGURE 1.4 – Diagramme de classes relatif aux mnémoniques.

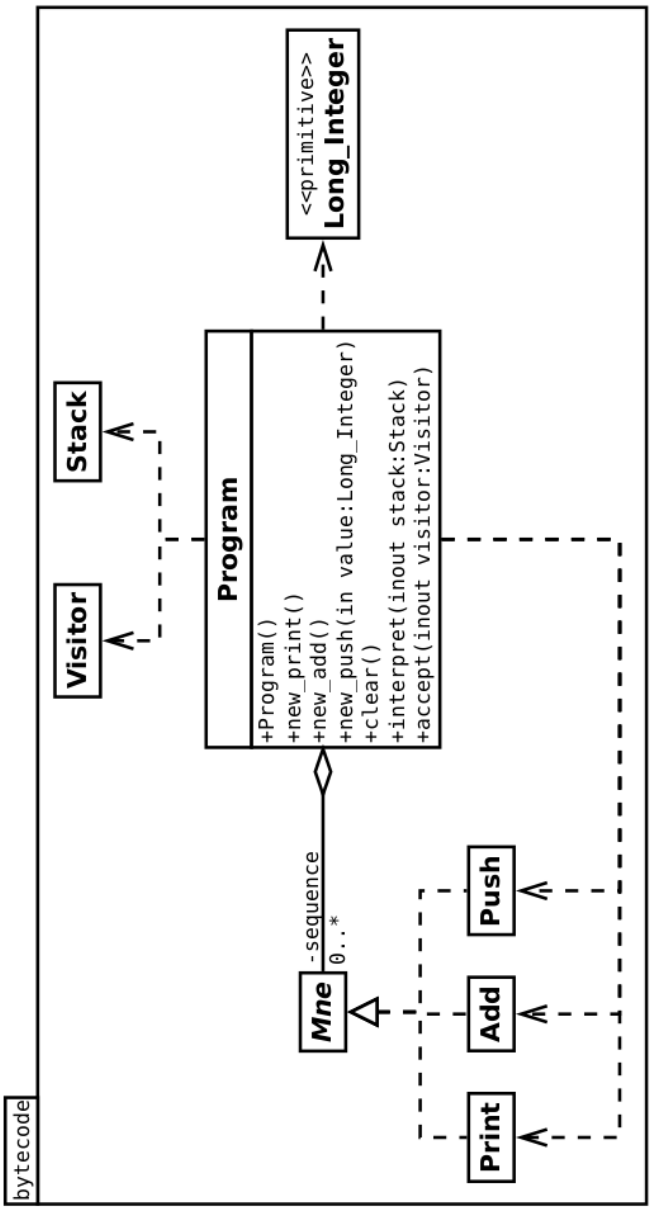


FIGURE 1.5 – Classe `Program` modélisant un code assembleur.

1.5 Question

Écrivez la classe `Program` qui représente un conteneur dynamique de mnémoniques en vous servant de la fonction `realloc`.

1.6 Question

Écrivez une classe non dérivable `Push_Counter` qui représente un visiteur permettant de compter le nombre de mnémoniques `Push` dans un code assembleur.

1.7 Question

Écrivez un programme de démonstration de votre application permettant de construire le code assembleur de la figure 1.2, de l'exécuter et d'afficher sur la sortie standard le nombre de ses mnémoniques `Push`.

Références

[1] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns : Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.