Nom _____

Prénom _____

Groupe 20 (A. Blondin Massé) ou 40 (G. Francoeur)

Code permanent _____

Solution de l'examen

Date : 26 octobre 2019

Titre du cours : Construction et maintenance de logiciels

Sigle : INF3135

Enseignants : Alexandre Blondin Massé (groupe 20) et Guy Francoeur (groupe 40)

Instructions

1) Vous avez trois heures pour répondre à l'examen ;
2) Vous avez droit à toute votre documentation papier ;
3) Vous ne devez pas dégrafer le questionnaire ;
4) Il est interdit d'utiliser un ordinateur, peu importe sa taille et sa forme (téléphone portable, agenda électronique, etc.) ;
5) Il est interdit de parler et de prêter de la documentation à un autre étudiant ;
6) Indiquez clairement vos réponses finales ;
7) Signez le registre de présence avant de quitter.

Question	1	2	3	4	5	Total
Sur	20	15	20	15	30	100
Note						

Question 1. . . . . (20 points)
Pour cette question, aucune justification n'est requise. Dans le cas d'une mauvaise réponse, vous pourriez obtenir quelques points si vous avec laissé des traces de votre démarche.

(1) (4 points) Quelle commande Unix parmi les suivantes permet de récupérer les lignes 6 à 10 inclusivement (la numérotation commence à 1) du fichier `README.md` et de les placer dans un fichier appelé `extrait.md`?

(a) `tail -n 10 README.md | head -n 5 > extrait.md`

(b) `head -n 10 README.md | tail -n 5 > extrait.md`

(c) `head -n 10 README.md | tail -n 5 | cat extrait.md`

(d) `tail -n 10 README.md | head -n 5 | cat extrait.md`

(2) (4 points) Quelle sous-commande de Git parmi les suivantes permet de consulter l'historique d'un projet ?

(a) `git pull` (c) `git history` (e) `git push` **(g) `git log`**

(b) `git fetch` (d) `git add` (f) `git commit` (h) `git rebase`

(3) (4 points) Considérez le travail pratique 1 et les deux commandes suivantes qui donnent de l'information sur le contenu du dépôt et, en particulier, du fichier `Makefile` :

```
$ tree --charset ascii   # Affiche l'arborescence du répertoire
.
|-- .gitignore
|-- check.bats
|-- examples
|   |-- soccer-default.out
|   |-- soccer.in
|   |-- soccer-long.in
|   |-- soccer-long-table-id.out
|   |-- soccer-long-table-names.out
|   |-- soccer-short.in
|   |-- tennis2.in
|   |-- tennis-default.out
|   |-- tennis.in
|   |-- tennis-name.out
|   |-- tennis-name-table.out
|   `-- tennis-table.out
|-- Makefile
|-- README.md
`-- tournament.c
$ cat Makefile
tournament: tournament.c
    gcc -o tournament tournament.c

.PHONY: test

test:
    bats check.bats
```

Parmi les expressions suivantes, laquelle devrait absolument se retrouver dans le fichier *.gitignore* ?

(a) *check.bats* (c) **.c* (e) **.in* (g) *Makefile*
(b) *examples/* (d) *test* (f) *README.md* **(h) *tournament***

(4) (4 points) Considérez les déclarations suivantes :

```
struct Repertoire {
    char *nom;
    unsigned int num_fichiers;
    struct Fichier *fichiers[30];
    unsigned int num_repertoires;
    struct Repertoire *repertoires[30];
};

struct Fichier {
    char *nom;
    unsigned int taille;
};

struct Repertoire rep;
```

Quelle instructions C permet d'afficher sur la sortie standard le nom du premier fichier contenu dans le répertoire référencé par la variable *rep* ?

(a) *printf("%s\n", rep.fichiers[0]->nom)*

(b) *printf("%s\n", rep.fichiers[0].nom)*

(c) *printf("%s\n", rep->fichiers[0].nom)*

(d) *printf("%s\n", rep->fichiers[0]->nom)*

(5) (4 points) Parmi les fonctions suivantes de la bibliothèque *string.h*, lesquelles sont vulnérables à l'**écriture** sur des zones de mémoire non réservées ? Encerclez toutes les réponses qui s'appliquent.

(a) *strcmp* (b) *strlen* **(c) *strcat*** (d) *strdup* **(e) *strcpy***

Question 2. . . . . (15 points)
Considérez le fichier *examen.c* ci-bas :

```c
// Fichier examen.c
#include <stdio.h>

#define MSG "CASINO"

enum jour {LUNDI, MARDI, MERCREDI, JEUDI, VENDREDI};

int main(int argc, char *argv[]) {
    enum jour j = JEUDI;
    printf("%s", MSG);
    printf("\t%s", argv[0]);
    printf("\n+%d", argc);
    for (int i = 1; i < argc - 1; ++i) {
        printf("/%c/",argv[i][1]);
    }
    printf("\n%d, %lu , %lu\n", j, sizeof j, sizeof(unsigned int));
    return 0;
}
```

(1) (5 points) Quelle est la commande qui permet de compiler le programme *examen.c* ci-haut et produire uniquement un exécutable nommé *examen* avec le standard C99 ?

Solution:

```
$ gcc -std=c99 -o examen examen.c
```

(2) (10 points) En supposant que l'exécutable *examen* a été produit correctement, et qu'un entier occupe 4 octets, qu'affiche la commande suivante sur la sortie standard ?

```
$ ./examen "un petit tour" de BMW X3 M
```

Solution:

```
CASINO    ./examen
+6/n//e//M//3/
3, 4 , 4
```

Question 3. . . . . (20 points)
Identifiez et corrigez les erreurs (s'il y a lieu) dans le programme suivant. Il doit compiler avec succès et sans avertissement avec `-Wall`. Annotez directement le code source avec les corrections (il a été aéré pour vous laisser de la place pour écrire).

Solution: Les corrections sont les suivantes :

1. Il faut ajouter `#include <stdio.h>` pour pouvoir utiliser `printf` ;
2. Le mot `typedefine` doit être remplacé par `typedef` ;
3. Il manque un point-virgule dans `struct Point_s` pour le champ `y` ;
4. Il manque un point-virgule dans la déclaration du tableau `a` ;
5. Il faut réserver de l'espace mémoire avec `malloc` pour la variable `var1` ;
6. Il faut ajouter `struct` devant `Point_s` ;
7. Il faut mettre `UINT` en majuscules ;
8. Il faut ajouter `*` devant `var1` dans l'affectation à la valeur `1` ;
9. Pour initialiser les champs `x` et `y` de `pt`, il faut utiliser l'opérateur `.` plutôt que `->` ;
10. Dans le premier affichage, le format devrait `%d` plutôt que `%s` ;
11. Toujours dans le premier affichage, il faut ajouter `*` devant `var1` ;
12. Encore dans le premier affichage, il faut utiliser l'opérateur `.` plutôt que `->` pour accéder aux champs de `pt` ;
13. Dans le deuxième affichage, il faut utiliser l'opérateur d'adressage `&` pour récupérer l'adresse de `b` ;
14. Finalement, il faut terminer la fonction `main` avec `return 0`.

Voici le code après correction :

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  typedef unsigned int UINT;
5
6  struct Point_s {
7      int x;
8      int y;
9  };
10
11 int main(int argc, char *argv[]) {
12     int a[3] = {1, -1, 2};
13     int *var1 = malloc(sizeof(int));
14     struct Point_s pt;
15     UINT b = 0;
16     *var1 = 1;
17     pt.x = a[2]; pt.y = a[1];
18     printf("var1 + ptx + pty = %d\n", *var1 + (pt.x + pt.y));
19     printf("L'adresse de la variable b est %p\n", &b);
20     return 0;
21 }
```

Question 4. . . . . (15 points)
Écrire un Makefile le plus court possible qui compile un programme nommé *projet.c* et produit un exécutable nommé *pro*. Vous devez exprimer toutes les dépendances et vous assurer que le fichier source est disponible. On s'attend à ce que la commande *make* produise le fichier exécutable *pro*.

De plus, proposez une cible permettant de supprimer les fichiers qui ont été produits par la commande *make*. Plus précisément, on s'attend à ce que la commande *make mrproper* nettoie le répertoire de tous les fichiers générés.

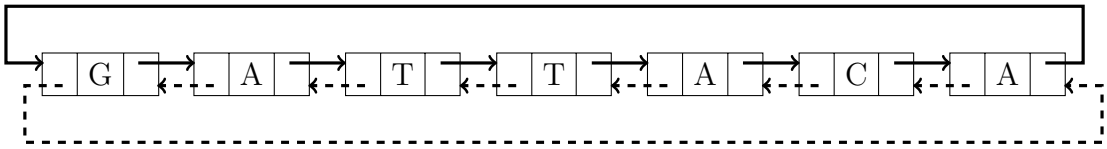
Solution:

```
pro: projet.c
    gcc -o $@ $^

.PHONY: mrproper

mrproper:
    rm -f pro
```

Question 5. ..... (30 points)
En informatique, une structure de données utile est la **liste circulaire**. Il s'agit essentiellement d'une liste doublement chaînée ayant la propriété additionnelle que le premier et le dernier noeud sont reliés comme s'ils étaient consécutifs. Voici une représentation schématique d'une liste circulaire contenant dans l'ordre les caractères `'G'`, `'A'`, `'T'`, `'T'`, `'A'`, `'C'` et `'A'` :

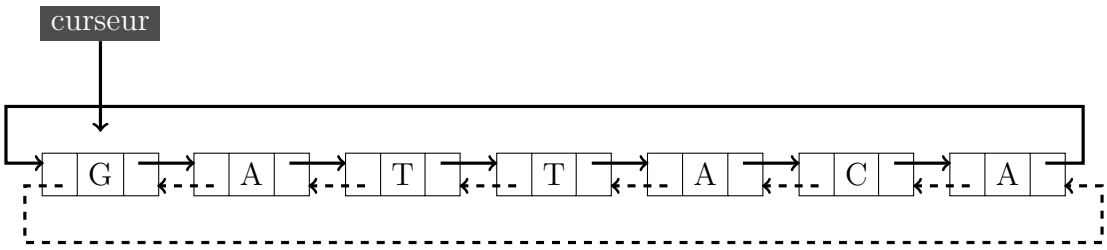


En C, un noeud d'une telle liste est représenté grâce à la déclaration suivante :

```
struct Node {           // A node in a circular list
    char value;         // The value
    struct Node *prev; // The previous node
    struct Node *next; // The next node
};
```

Dans la figure, les flèches pleines correspondent aux pointeurs `next` (noeud suivant), alors que les flèches pointillées correspondent aux pointeurs `prev` (noeud précédent).

Un **curseur** d'une liste circulaire est simplement un pointeur vers un des noeuds. Dans l'exemple ci-haut, on pourrait avoir par exemple un curseur qui pointe vers le noeud le plus à gauche de la liste circulaire :



Dans cette question, vous devez compléter l'implémentation de fonctions permettant de manipuler une liste circulaire contenant des caractères.

Plus spécifiquement, considérez le fichier `circular.c` suivant, dans lequel certaines fonctions sont implémentées alors que d'autres sont incomplètes. Prenez le temps de bien lire toutes les déclarations, incluant la documentation (*docstring*) expliquant le comportement attendu de chaque fonction :

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

// ----- //
// Types //
// ----- //
```

```c
struct Node {          // A node in a circular list
    char value;        // The value
    struct Node *prev; // The previous node
    struct Node *next; // The next node
};

/**
 * A circular list is exactly like a doubly linked list, with the exception
 * that the first and last nodes are also linked, making a loop.
 */
struct CircularList {          // A circular list
    struct Node *cursor;       // The current node
    unsigned int num_elements; // The number of elements
};

// --------- //
// Functions //
// --------- //

/**
 * Creates a circular list
 *
 * Note: Do not forget to use `circular_delete` when you are finished.
 *
 * @return  The circular list
 */
struct CircularList circular_create() {
    struct CircularList cl;
    cl.cursor = NULL;
    cl.num_elements = 0;
    return cl;
}

/**
 * Returns true if and only if a circular list is empty
 *
 * @param cl  The circular list
 * @return    True if the list is empty, false otherwise
 */
bool circular_is_empty(const struct CircularList *cl) {
    // TODO 1 //
}

/**
 * Prints the `n` first elements of a circular list to `stdout`,
 * starting from the current node (cursor)
 *
 * Since the list is circular, `n` can be larger than the number of elements in
 * the list, and the elements are simply repeated as many times as needed.
 *
 * If the list is empty, then it prints "( )", whatever the value of `n`.
 *
 * @param cl       A pointer to a circular list
 * @param n        The number of elements to print
 * @param forward  If true, the list is printed forward
 *                 otherwise, it is printed backward
 */
void circular_print(const struct CircularList *cl,
                    unsigned int n,
                    bool forward) {
    // TODO 2 //
}

/**
 * Inserts a value into a circular list
 *
```

```
 * @param cl     The circular list
 * @param value  The value to be inserted
 */
void circular_insert(struct CircularList *cl, char value) {
    struct Node *node = malloc(sizeof(struct Node));
    node->value = value;
    if (circular_is_empty(cl)) {
        node->next = node;
        node->prev = node;
    } else {
        node->next = cl->cursor;
        node->prev = cl->cursor->prev;
        cl->cursor->prev = node;
        if (cl->cursor->next == cl->cursor) {
            cl->cursor->next = node;
        }
        node->prev->next = node;
    }
    cl->cursor = node;
    ++cl->num_elements;
}

/**
 * Pops the current value (at the cursor) from a circular list
 *
 * In other words, it removes the node pointed by the cursor from the circular
 * list and returns the character contained in it.
 *
 * Note: If the list is empty, the character '?' is returned and an error is
 * printed on stderr.
 *
 * @param cl  The circular list
 * @return    The character stored at the cursor
 */
char circular_pop(struct CircularList *cl) {
    // TODO 3 //
}

/**
 * Shifts the cursor of a circular list from `i` elements.
 *
 * If the list is empty, nothing happens.
 * If `i > 0`, then the cursor is shifted by `i` elements forward (next
 * direction).
 * If `i < 0`, then the cursor is shifted by `-i` elements backward (prev
 * direction).
 *
 * @param cl  The circular list
 * @param i   The shift value
 */
void circular_shift(struct CircularList *cl, int i) {
    // TODO 4 //
}

/**
 * Deletes a circular list
 *
 * @param cl  The circular list
 */
void circular_delete(struct CircularList *cl) {
    while (!circular_is_empty(cl)) {
        circular_pop(cl);
    }
}

// ---- //
```

```
// Main //
// ---- //

int main() {
    struct CircularList cl = circular_create();
    printf("Initially, the circular list is empty: ");
    circular_print(&cl, cl.num_elements, true);
    printf("\nInserting the characters A, C, A, T, T, A, G: ");
    char *gattaca = "ACATTAG";
    for (unsigned int i = 0; i < 7; ++i) {
        circular_insert(&cl, gattaca[i]);
    }
    circular_print(&cl, cl.num_elements, true);
    printf("\nPrinted backward: ");
    circular_print(&cl, cl.num_elements, false);
    printf("\nPrinting 18 elements: ");
    circular_print(&cl, 18, true);
    printf("\nPrinting 18 elements backward: ");
    circular_print(&cl, 18, false);
    printf("\nShifted by 2: ");
    circular_shift(&cl, 2);
    circular_print(&cl, cl.num_elements, true);
    printf("\nShifted by -3: ");
    circular_shift(&cl, -3);
    circular_print(&cl, cl.num_elements, true);
    printf("\nRemoving current node: ");
    circular_pop(&cl);
    circular_print(&cl, cl.num_elements, true);
    printf(" forward and ");
    circular_print(&cl, cl.num_elements, false);
    printf(" backward\n");
    printf("Removing 4 nodes: ");
    for (unsigned int i = 0; i < 4; ++i) {
        circular_pop(&cl);
    }
    circular_print(&cl, cl.num_elements, true);
    printf("\nInserting C: ");
    circular_insert(&cl, 'C');
    circular_print(&cl, cl.num_elements, true);
    printf("\n");
    circular_delete(&cl);
    return 0;
}
```

On souhaiterait avoir le comportement suivant lors de la compilation et de l'exécution.

```
$ gcc -o circular circular.c
$ ./circular
Initially, the circular list is empty: ( )
Inserting the characters A, C, A, T, T, A, G: ( G A T T A C A )
Printed backward: ( G A C A T T A )
Printing 18 elements: ( G A T T A C A G A T T A C A G A T T )
Printing 18 elements backward: ( G A C A T T A G A C A T T A G A C A )
Shifted by 2: ( C A G A T T A )
Shifted by -3: ( A T T A C A G )
Removing current node: ( T T A C A G ) forward and ( T G A C A T ) backward
Removing 4 nodes: ( A G )
Inserting C: ( C A G )
```

Complétez l'implémentation des fonctions

(1) (6 points) `circular_is_empty` (`// TODO 1 //`),

(2) (8 points) `circular_print` (`// TODO 2 //`),

(3) (8 points) `circular_pop` (`// TODO 3 //`) et

(4) (8 points) `circular_shift` (`// TODO 4 //`)
de telle sorte que le comportement du programme soit celui affiché plus haut.
Remarque : Si vous ne savez pas comment implémenter une fonction, vous pouvez sans problème faire appel à elle dans une autre fonction en supposant qu'elle est implémentée. Par exemple, si vous n'avez pas réussi à implémenter `circular_is_empty` et que vous en avez besoin dans `circular_shift`, vous pouvez y faire appel comme si vous l'aviez correctement implémentée.

Solution:

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

// ----- //
// Types //
// ----- //

struct Node {           // A node in a circular list
    char value;         // The value
    struct Node *prev; // The previous node
    struct Node *next; // The next node
};

/**
 * A circular list is exactly like a doubly linked list, with the exception
 * that the first and last nodes are also linked, making a loop.
 */
struct CircularList {           // A circular list
    struct Node *cursor;        // The current node
    unsigned int num_elements; // The number of elements
};

// --------- //
// Functions //
// --------- //

/**
 * Creates a circular list
 *
 * Note: Do not forget to use `circular_delete` when you are finished.
 *
 * @return  The circular list
 */
struct CircularList circular_create() {
    struct CircularList cl;
    cl.cursor = NULL;
    cl.num_elements = 0;
    return cl;
}

/**
 * Returns true if and only if a circular list is empty
 *
 * @param cl  The circular list
 * @return    True if the list is empty, false otherwise
 */
bool circular_is_empty(const struct CircularList *cl) {
    return cl->num_elements == 0;
}
```

```c
/**
 * Prints the `n` first elements of a circular list to `stdout`,
 * starting from the current node (cursor)
 *
 * Since the list is circular, `n` can be larger than the number of elements in
 * the list, and the elements are simply repeated as many times as needed.
 *
 * If the list is empty, then it prints "( )", whatever the value of `n`.
 *
 * @param cl       A pointer to a circular list
 * @param n        The number of elements to print
 * @param forward  If true, the list is printed forward
 *                 otherwise, it is printed backward
 */
void circular_print(const struct CircularList *cl,
                    unsigned int n,
                    bool forward) {
    printf("( ");
    struct Node *current = cl->cursor;
    n = cl->num_elements == 0 ? 0 : n;
    for (unsigned int i = 0; i < n; ++i) {
        printf("%c ", current->value);
        current = forward ? current->next : current->prev;
    }
    printf(")");
}

/**
 * Inserts a value into a circular list
 *
 * @param cl     The circular list
 * @param value  The value to be inserted
 */
void circular_insert(struct CircularList *cl, char value) {
    struct Node *node = malloc(sizeof(struct Node));
    node->value = value;
    if (circular_is_empty(cl)) {
        node->next = node;
        node->prev = node;
    } else {
        node->next = cl->cursor;
        node->prev = cl->cursor->prev;
        cl->cursor->prev = node;
        if (cl->cursor->next == cl->cursor) {
            cl->cursor->next = node;
        }
        node->prev->next = node;
    }
    cl->cursor = node;
    ++cl->num_elements;
}

/**
 * Pops the current value (at the cursor) from a circular list
 *
 * In other words, it removes the node pointed by the cursor from the circular
 * list and returns the character contained in it.
 *
 * Note: If the list is empty, the character '?' is returned and an error is
 * printed on stderr.
 *
 * @param cl  The circular list
 * @return    The character stored at the cursor
 */
```

```
char circular_pop(struct CircularList *cl) {
    if (!circular_is_empty(cl)) {
        struct Node *node = cl->cursor;
        char value = cl->cursor->value;
        cl->cursor = node->next;
        node->next->prev = node->prev;
        node->prev->next = node->next;
        free(node);
        --cl->num_elements;
        return value;
    } else {
        fprintf(stderr, "Error: cannot pop from empty circular list\n");
        return '?';
    }
}

/**
 * Shifts the cursor of a circular list from `i` elements.
 *
 * If the list is empty, nothing happens.
 * If `i > 0`, then the cursor is shifted by `i` elements forward (next
 * direction).
 * If `i < 0`, then the cursor is shifted by `-i` elements backward (prev
 * direction).
 *
 * @param cl  The circular list
 * @param i   The shift value
 */
void circular_shift(struct CircularList *cl, int i) {
    if (!circular_is_empty(cl)) {
        while (i != 0) {
            if (i > 0) {
                cl->cursor = cl->cursor->prev;
                --i;
            } else {
                cl->cursor = cl->cursor->next;
                ++i;
            }
        }
    }
}

/**
 * Deletes a circular list
 *
 * @param cl  The circular list
 */
void circular_delete(struct CircularList *cl) {
    while (!circular_is_empty(cl)) {
        circular_pop(cl);
    }
}

// ---- //
// Main //
// ---- //

int main() {
    struct CircularList cl = circular_create();
    printf("Initially, the circular list is empty: ");
    circular_print(&cl, cl.num_elements, true);
    printf("\nInserting the characters A, C, A, T, T, A, G: ");
    char *gattaca = "ACATTAG";
    for (unsigned int i = 0; i < 7; ++i) {
```

```
        circular_insert(&cl, gattaca[i]);
    }
    circular_print(&cl, cl.num_elements, true);
    printf("\nPrinted backward: ");
    circular_print(&cl, cl.num_elements, false);
    printf("\nPrinting 18 elements: ");
    circular_print(&cl, 18, true);
    printf("\nPrinting 18 elements backward: ");
    circular_print(&cl, 18, false);
    printf("\nShifted by 2: ");
    circular_shift(&cl, 2);
    circular_print(&cl, cl.num_elements, true);
    printf("\nShifted by -3: ");
    circular_shift(&cl, -3);
    circular_print(&cl, cl.num_elements, true);
    printf("\nRemoving current node: ");
    circular_pop(&cl);
    circular_print(&cl, cl.num_elements, true);
    printf(" forward and ");
    circular_print(&cl, cl.num_elements, false);
    printf(" backward\n");
    printf("Removing 4 nodes: ");
    for (unsigned int i = 0; i < 4; ++i) {
        circular_pop(&cl);
    }
    circular_print(&cl, cl.num_elements, true);
    printf("\nInserting C: ");
    circular_insert(&cl, 'C');
    circular_print(&cl, cl.num_elements, true);
    printf("\n");
    circular_delete(&cl);
    return 0;
}
```