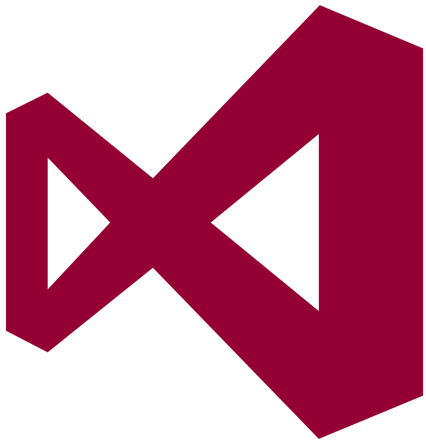


Rapport de projet

Introduction à l'Algorithmique et la Programmation

Antoine BANHA
Groupe 102

Logan TANN
Groupe 102

DUT 1ère Année

Université de Paris

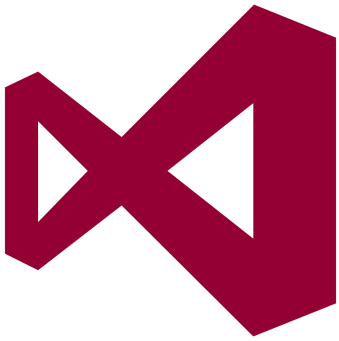


Table des matières

Antoine BANHA
Groupe 102

Logan TANN
Groupe 102

Université de Paris

Présentation du problème

Nounours Experts, une entreprise qui a débuté en étant une start-up de services Business to Business, connaît un succès fou et se développe à une vitesse vertigineuse. Très demandée et efficace aux entreprises durant cette période de crise sanitaire, le nombre de ses commandes, qui se comptait auparavant sur les doigts de la main, se totalise aujourd'hui à 300 commandes ! Pour cela, elle doit embaucher de nouveaux travailleurs, qui, tel une petite usine de travailleurs modernes, sont regroupés dans dix secteurs : développeurs, designers, chefs de projets, graphistes… Il est également possible qu'une personne puisse être dans deux secteurs à la fois (graphiste + designer par exemple). Devant un nombre de travailleurs, de clients et de demandes devenu si grand, l'entreprise se doit de connaître de respecter son efficacité, en particulier connaître l'état d'avancement des commandes, des tâches des travailleurs, ou même calculer la facturation de chaque client. Autrefois fait manuellement par une petite équipe de managers, l'homme reste imparfait, se fatigue vite, et coûte aussi.

Cherchant à gagner en simplicité, en rapidité et en fiabilité, l'entreprise décide de se tourner vers une solution informatique sur mesure. Il n'y aurait alors qu'une seule personne formée pour faire tout ce petit boulot, à contrôler le programme. Seulement, il est parfois difficile de faire communiquer l'humain à la machine, car la machine ne parle pas et ne comprends pas le français. C'est pour cela que le but de ce projet est d'implémenter un interpréteur de commandes : La personne formée n'aura qu'à entrer une suite d'instructions, et la machine s'occupera de traiter celle-ci, mais aussi de stocker les données nécessaires, effectuer les routines d'automatisation, et communiquer avec cette personne.

Afin de réaliser le projet, un cahier des charges très précis a été fourni, et afin que le code puisse être repris par l'entreprise, il convient de respecter les standards C, mais aussi respecter certains critères de qualité du code, tel qu'une documentation et un prototypage de chacune des fonctions utilisées dans le code, mais aussi un maximum de 80 caractères par lignes si l'équipe qui reprendra notre programme dans le futur code sur un terminal. Le choix du C est sa puissance, sa légèreté et la possibilité de le porter vers des systèmes embarqués. Cependant, le C étant un langage bas niveau, les possibilités d'erreurs sont

d’autant plus faciles à commettre (car rappelons qu’un programme est codé par un humain), c’est pourquoi il est nécessaire d’effectuer et de scrupuleusement respecter les tests unitaires (avec des assertions) et end-to-end (avec des fichiers “in” et “out”).

Le programme suit les spécifications fournies dans le cahier des charges, c’est-à-dire, embarquer les spécialités, les travailleurs, les clients et les commandes dans des tableaux de structures. À la différence de cette même structure, nous avons empaqueté l’ensemble du stockage dans une même structure appelée “store”. Celle-ci a très peu d’incidence sur la taille mémoire employée, car les fonctions de traitements se serviront d’un pointeur plutôt que de copier tout son contenu dans une variable locale, tout en limitant le nombre d’arguments nécessaires dans ces mêmes fonctions.

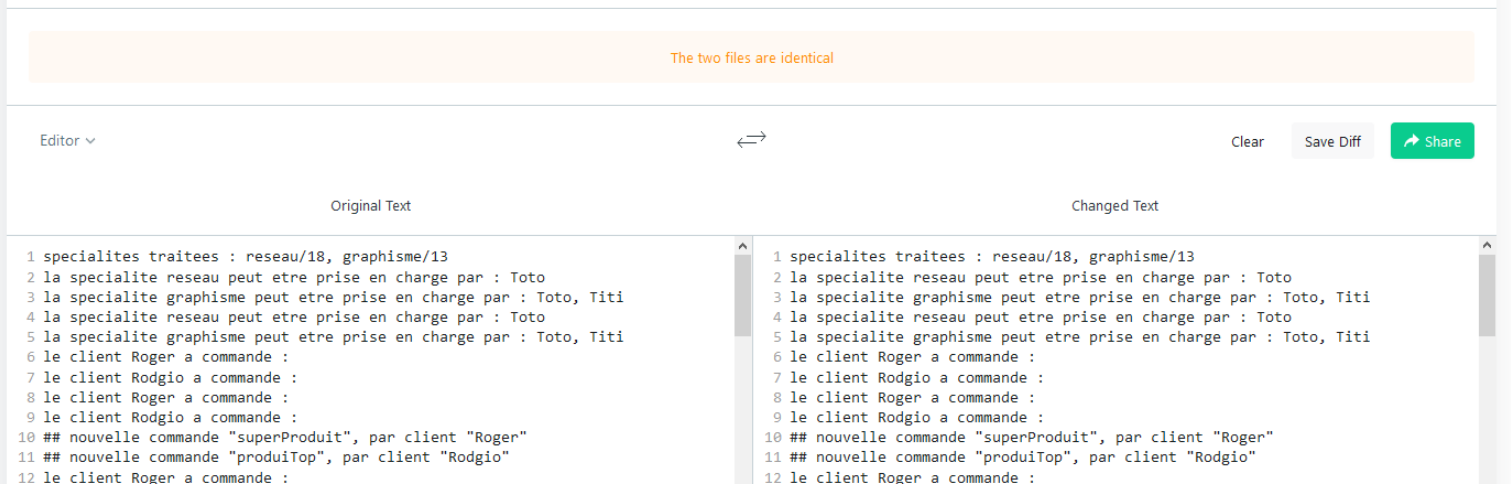
(le contexte est évidemment fictif, mais toute l’explication se tient dans le sujet et le contenu du projet)

Preuves de validation

Le sprint 6 ayant été validé en soutenance, le diffchecker de celui-ci n'est pas présent ici.

Sprint 1-release

Sprint 2-release

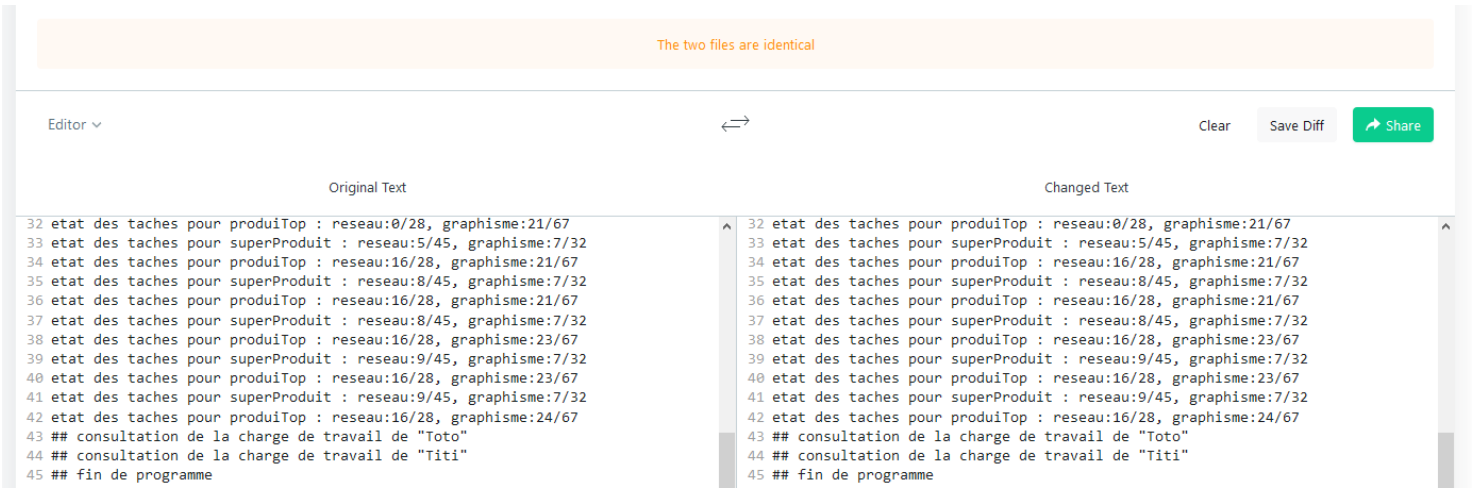


Antoine BANHA – Logan TANN

Projet IAP

Université de Paris

Sprint 3-release



Sprint 4-release

Sprint 5-release

Bilan du projet

Nous avons choisi de faire chacun notre bilan, en essayant d'être le plus honnête et personnel possible.

Logan

En premier lieu, parlons de notre expérience. Antoine et moi-même avons eu plusieurs années de programmation derrière nous.

Pourtant, moi et Antoine avons une manière de coder totalement différente. De mon point de vue, cela a été ma principale difficulté. Antoine est remarquablement rapide pour pondre du code, tout le contraire de moi-même qui doit analyser de tête le comportement du code que je m'apprête à taper, ce qui fait que lorsque nous codons ensemble via LiveShare, j'étais souvent à la traîne. J'ai finalement pu me démarquer en trouvant les principaux bugs qui affectaient le sprint 6-r lors de notre soutenance.

Ainsi, pas de réelles difficultés. Si je devais choisir le sprint le plus difficile, cela aurait été le 3 car c'est celui qui m'a pris le plus de temps. J'ai fait la majeure partie *code-refactoring* (documentations, 80 chars/lignes…) ainsi que la rédaction de la partie de ce dossier à propos de la présentation de la problèmatique et du programme, cela m'a également pris beaucoup de temps.

Notre plus grande frayeur a été d'une part le bug du sprint 6-r, qui était causé par un supérieur ou égal au lieu d'un supérieur strict ainsi que d'un printf ayant un mauvais argument et faisait crasher le programme, ainsi que le diffchecker retournant plein d'erreurs sur notre programme corrigé et passé avec succès en soutenance passée le jour précédent, qui se relevait être une erreur de ma part : j'avais juste mis le fichier out de notre ancien programme bogué, au lieu du out donné par le professeur !

Le souci, qui est aussi le cas de tous les groupes, est de devoir comprendre le code qu'a fait son partenaire pendant notre absence. Quoi qu'il en soit, nous n'avons pas eu de

réels conflits de mon point de vue, et je trouve cela ironiquement étonnant car c'est selon moi la chose qui arrive le plus facilement dans des projets de groupe.

Passons aux outils utilisés. Nous nous sommes servis de Git et la plateforme Github pour archiver le code de chaque sprint, ainsi que Visual Studio et son outil de collaboration LiveShare pour coder le projet. Nous nous sommes entièrement organisés avec la messagerie Discord, et petite particularité qui en fait notre marque de fabrique, le rapport a été entièrement fait sur LibreOffice Writer. Antoine ayant l'habitude de créer ses rapports sur Draw et moi-même sur Impress, le choix s'est finalement porté sur le style graphique d'Antoine et l'utilisation de Writer, afin de faciliter le collage du code source de l'ensemble des sprints. J'espère que celui-ci vous plaira !

Enfin, le principal point d'amélioration était que certaines parties du code auraient pu être segmentées en deux fonctions. Les instructions "tous" faisaient par exemple la même chose que le traitement d'un seul argument, mais on répétait pour chaque client ou commandes … Dans le développement Agile, un des principes fondateurs est une chose par fonctions. J'aurais été capable de le faire, mais vu que ça fonctionnait, je n'ai pas voulu perdre mon temps à déplacer mon code dans de nouvelles fonctions, ma priorité était la documentation du code et les révisions du DST, sans compter la rédaction de ces lignes !

Antoine

Ce premier projet a été très enrichissant pour ma part et l'amphithéâtre de présentation ne m'a absolument pas déçu. L'énoncé était austère mais assez complet et clair pour que nous puissions mener à bien ce projet. Les TP et les TD de IAP nous ont été très utiles dans notre compréhension du C et c'est bien sûr grâce à eux que nous avons pu améliorer considérablement le code progressivement au point on nous avons dû parfois revenir en arrière dans notre code pour corriger des imperfections que nous avons remarqué un peu plus tard et les branches de Git s'est avéré très utile dans ce cas précis.

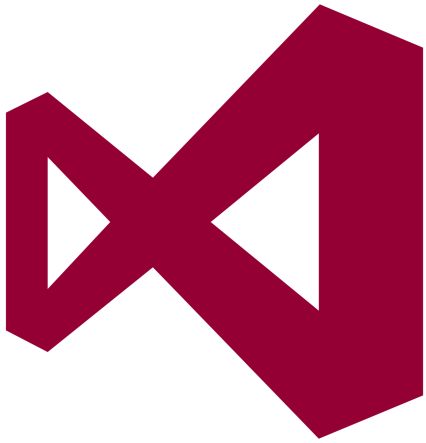
Nous avons pu également découvrir plusieurs fonctionnalités propres au C, tel que les structures, les typedefs mais également les pointeurs et pouvoir mettre en pratique

tout cela dans un projet concret a été très instructif.

Malgré cette période assez chaotique, nous avons pu nous organiser pour travailler à distance en mettant sur la table divers outils de collaboration, dont Git/GitHub, mais aussi le plugin Visual Studio Live Share et bien évidemment Discord. Moi et Logan avons pu collaborer très facilement et en question de productivité, je pense que nous avons été beaucoup plus productifs à la maison qu'en présentiel.

Nous avons également appris plus sur les méthodes de documentation et nous avons décidé de nous y prendre assez tôt dans le projet. Nous avons regardé ce que Doxygen pouvait nous proposer mais avons décidé de plutôt nous pencher sur la méthode donnée en Amphithéâtre.

Bilan global pour ma part : très enrichissant et totalement à refaire.



Annexes

Sprint 1-release au sprint 6-release

```c
/*
   sprint1-r
*/

#pragma warning(disable:4996)
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <locale.h>

// Messages emis par les instructions

#define MSG_DEVELOPPE "## nouvelle specialite \"%s\" ; cout horaire \"%d\"\n"
#define MSG_SPECIALITES "## consultation des specialites\n"
#define MSG_INTERRUPTION "## fin de programme\n"
#define MSG_EMBAUCHE "## nouveau travailleur \"%s\" competent pour la specialite \"%s\"\n"
#define MSG_TRAVAILLEURS "## consultation des travailleurs competents pour la
specialite \"%s\"\n"
#define MSG_TRAVAILLEURS_TOUS "## consultation des travailleurs competents pour chaque
specialite\n"
#define MSG_DEMARCHE "## nouveau client \"%s\"\n"
#define MSG_CLIENT "## consultation des commandes effectuees par \"%s\"\n"
#define MSG_CLIENT_TOUS "## consultation des commandes effectuees par chaque client\n"
#define MSG_COMMANDE "## nouvelle commande \"%s\", par client \"%s\"\n"
#define MSG_SUPERVISION "## consultation de l'avancement des commandes\n"
#define MSG_TACHE "## la commande \"%s\" requiere la specialite \"%s\" (nombre
d'heures \"%d\")\n"
#define MSG_CHARGE   "## consultation de la charge de travail de \"%s\"\n"
#define MSG_PROGRESSION "## pour la commande \"%s\", pour la specialite \"%s\" : \"%d\"
heures de plus ont ete realisees\n"
#define MSG_PASSE "## une reallocation est requise\n"

// Lexemes

#define LGMOT 35
#define NBCHIFFREMAX 5
typedef char Mot[LGMOT + 1]; // Définition du type Mot

typedef enum { FAUX = 0, VRAI = 1 } Booleen;
Booleen EchoActif = FAUX;

/**
   void get_id
   @param id Mot:
**/

void get_id(Mot id) {
    scanf("%s", id);
    if (EchoActif) printf(">>echo %s\n", id);
}

/**
   int get_int()
   get the current word from the input, and returns it with the (int) type
**/

int get_int() {
    char buffer[NBCHIFFREMAX + 1];
    scanf("%s", buffer);
    if (EchoActif) printf(">>echo %s\n", buffer);
    return atoi(buffer);
}
```

```c
// Commandes

/*
* traite_developpe()
* developpe <Mot nom_specialite> <int cout_horaire>
* Ajoute une specialité au programme
*/
void traite_developpe() {
    Mot nom_specialite;
    get_id(nom_specialite);
    int cout_horaire = get_int();
    printf(MSG_DEVELOPPE, nom_specialite, cout_horaire);
}

/*
* traite_specialites()
* specialites
* Affiche les specialités enregistrées dans le programme
*/
void traite_specialites() {
    printf(MSG_SPECIALITES);
}

/*
* traite_embauche() : traite les arguments de la commande suivante :
* embauche <Mot travailleur> <Mot specialite>
*/
void traite_embauche() {
    Mot travailleur, specialite;
    get_id(travailleur);
    get_id(specialite);
    printf(MSG_EMBAUCHE, travailleur, specialite);
}

/*
* traite_travailleurs()
* travailleurs <Mot specialite = "tous">
* Affiche les travailleurs selon la specialité choisie ou non
*/
void traite_travailleurs() {
    Mot specialite;
    get_id(specialite);
    if (strcmp(specialite, "tous") == 0)
        printf(MSG_TRAVAILLEURS_TOUS);
    else
        printf(MSG_TRAVAILLEURS, specialite);
}

/*
* traite_demarche()
* demarche <Mot nom_client>
* Ajoute un client au programme
*/
void traite_demarche() {
    Mot nom_client;
    get_id(nom_client);
    printf(MSG_DEMARCHE, nom_client);
}
```

```c
/*
* traite_client()
* client <Mot nom_client>
* Affiche les commandes d'un client
*/
void traite_client() {
    Mot nom_client;
    get_id(nom_client);
    if (strcmp(nom_client, "tous") == 0)
        printf(MSG_CLIENT_TOUS);
    else
        printf(MSG_CLIENT, nom_client);
}

/*
* traite_commande()
* commande <Mot produit> <Mot nom_client>
* Ajoute une commande au programme
*/
void traite_commande() {
    Mot produit, nom_client;
    get_id(produit);
    get_id(nom_client);
    printf(MSG_COMMANDE, produit, nom_client);
}

/*
* traite_supervision()
* supervision
* Affiche l'avancement actuel des commandes
*/
void traite_supervision() {
    printf(MSG_SUPERVISION);
}

/*
* traite_tache()
* tache <Mot produit> <Mot specialite> <int heures>
* Ajoute une specialité ainsi que le nombre d'heures requises à une commande
*/
void traite_tache() {
    Mot produit, nom_client;
    get_id(produit);
    get_id(nom_client);
    int heures = get_int();
    printf(MSG_TACHE, produit, nom_client, heures);
}

/*
* traite_charge()
* charge <Mot nom_travailleur>
* A
*/
void traite_charge() {
    Mot nom_travailleur;
    get_id(nom_travailleur);
    printf(MSG_CHARGE, nom_travailleur);
}
```

```c
/*
* traite_progression()
* tache <Mot produit> <Mot specialite> <int heures_travaillees>
* Enregistre une progression aux seins d'une specialité d'une commande
*/
void traite_progression() {
    Mot produit, nom_client;
    get_id(produit);
    get_id(nom_client);
    int heures_travaillees = get_int();
    printf(MSG_PROGRESSION, produit, nom_client, heures_travaillees);
}

/*
* traite_passe()
* passe
* Traitement des réallocations des dernières progressions
*/
void traite_passe() {
    printf(MSG_PASSE);
}

/*
* traite_interruption()
* interruption
* Interromp le programme
*/
void traite_interruption() {
    printf(MSG_INTERRUPTION);
}

// Porte d'entrée du programme
int main(int argc, char* argv[]) {
    if (argc >= 2 && strcmp("echo", argv[1]) == 0) {
        EchoActif = VRAI;
    }
    setlocale(LC_ALL, "fr-FR");
    Mot buffer;
    while (VRAI) {
        get_id(buffer);
        if (strcmp(buffer, "developpe") == 0) {
            traite_developpe();
            continue;
        }
        if (strcmp(buffer, "specialites") == 0) {
            traite_specialites();
            continue;
        }
        if (strcmp(buffer, "embauche") == 0) {
            traite_embauche();
            continue;
        }
        if (strcmp(buffer, "travailleurs") == 0) {
            traite_travailleurs();
            continue;
        }
        if (strcmp(buffer, "demarche") == 0) {
            traite_demarche();
            continue;
        }
        if (strcmp(buffer, "client") == 0) {
            traite_client();
            continue;
        }
        if (strcmp(buffer, "commande") == 0) {
            traite_commande();
            continue;
```

```
        }
        if (strcmp(buffer, "supervision") == 0) {
            traite_supervision();
            continue;
        }
        if (strcmp(buffer, "tache") == 0) {
            traite_tache();
            continue;
        }
        if (strcmp(buffer, "charge") == 0) {
            traite_charge();
            continue;
        }
        if (strcmp(buffer, "progression") == 0) {
            traite_progression();
            continue;
        }
        if (strcmp(buffer, "passe") == 0) {
            traite_passe();
            continue;
        }
        if (strcmp(buffer, "interruption") == 0) {
            traite_interruption();
            break;
        }
        printf("!!! instruction inconnue >%s< !!!\n", buffer);
    }
    return 0;
}
```

```c
/*
   sprint2-r
*/

#pragma warning(disable:4996)
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <locale.h>

// Messages emis par les instructions

#define MSG_DEVELOPPE "## nouvelle specialite \"%s\" ; cout horaire \"%d\"\n"
#define MSG_SPECIALITES "specialites traitees : "
#define MSG_INTERRUPTION "## fin de programme\n"
#define MSG_TRAVAILLEURS "la specialite %s peut etre prise en charge par : "
#define MSG_CLIENT "le client %s a commande : "
#define MSG_COMMANDE "## nouvelle commande \"%s\", par client \"%s\"\n"
#define MSG_SUPERVISION "## consultation de l'avancement des commandes\n"
#define MSG_TACHE "## la commande \"%s\" requiere la specialite \"%s\" (nombre
d'heures \"%d\")\n"
#define MSG_CHARGE   "## consultation de la charge de travail de \"%s\"\n"
#define MSG_PROGRESSION "## pour la commande \"%s\", pour la specialite \"%s\" : \"%d\"
heures de plus ont ete realisees\n"
#define MSG_PASSE "## une reallocation est requise\n"

// Lexemes

#define LGMOT 35
#define NBCHIFFREMAX 5
typedef char Mot[LGMOT + 1]; // Définition du type Mot

typedef enum { FAUX = 0, VRAI = 1 } Booleen;
Booleen EchoActif = FAUX;

/**
   void get_id
   @param id Mot:
**/
void get_id(Mot* id) {
    scanf("%s", id);
    if (EchoActif) printf(">>echo %s\n", id);
}

/**
   int get_int()
   get the current word from the input, and returns it with the (int) type
**/
int get_int() {
    char buffer[NBCHIFFREMAX + 1];
    scanf("%s", buffer);
    if (EchoActif) printf(">>echo %s\n", buffer);
    return atoi(buffer);
}

// Stockage

const enum {
    SPECIALITE_SIZE = 10,
    TRAVAILLEURS_SIZE = 50,
    CLIENTS_SIZE = 100,
    COMMANDES_SIZE = 500
};
```

```c
// Specialité

typedef struct {
    Mot nom;
    int cout_horaire;
} Specialite;
typedef struct {
    Specialite table[SPECIALITE_SIZE];
    int inserted;
} Specialites;

// Travailleurs

typedef struct {
    Mot nom;
    Booleen tag_specialite[SPECIALITE_SIZE];
} Travailleur;
typedef struct {
    Travailleur table[TRAVAILLEURS_SIZE];
    int inserted;
} Travailleurs;

// Client

typedef struct {
    Mot nom;
} Client;
typedef struct {
    Client table[CLIENTS_SIZE];
    int inserted;
} Clients;

// Commande

typedef struct {
    Mot produit;
    Mot client;
} Commande;
typedef struct {
    Commande table[COMMANDES_SIZE];
    int inserted;
} Commandes;

// Stockage global

typedef struct {
    Specialites specialites;
    Travailleurs travailleurs;
    Clients clients;
    Commandes commandes;
} Stockage;


// Helpers

int getIndex_trv(Travailleurs* travailleurs, Mot nom) {
    for (unsigned int i = 0; i < travailleurs->inserted; i++) {
        if (strcmp(travailleurs->table[i].nom, nom) == 0) {
            return i;
        }
    }
    return -1; //fallback
}
```

```c
int getIndex_spe(Specialites* specialites, Mot nom) {
    for (unsigned int i = 0; i < specialites->inserted; i++) {
        if (strcmp(specialites->table[i].nom, nom) == 0) {
            return i;
        }
    }
    return 0; //fallback
}

// Commandes

/*
* traite_developpe()
* developpe <Mot nom_specialite> <int cout_horaire>
* Ajoute une specialité au programme
*/
void traite_developpe(Stockage* store) {
    Mot nom_specialite;
    get_id(&nom_specialite);
    int cout_horaire = get_int();
    if (store->specialites.inserted < SPECIALITE_SIZE) {
        strcpy(store->specialites.table[store->specialites.inserted].nom, &nom_specialite);
        store->specialites.table[store->specialites.inserted].cout_horaire = cout_horaire;
        store->specialites.inserted++;
    }
}

/*
* traite_specialites()
* specialites
* Affiche les specialités enregistrées dans le programme
*/
void traite_specialites(Stockage* store) {
    printf(MSG_SPECIALITES);
    for (int i = 0; i < store->specialites.inserted; ++i) {
        if (i == 0)
            printf("%s/%d", store->specialites.table[i].nom, store-
>specialites.table[i].cout_horaire);
        else
            printf(", %s/%d", store->specialites.table[i].nom, store-
>specialites.table[i].cout_horaire);
    }
    printf("\n");
}

/*
* traite_embauche() : traite les arguments de la commande suivante :
* embauche <Mot travailleur> <Mot specialite>
*/
void traite_embauche(Stockage* store) {
    Mot travailleur, specialite;
    get_id(&travailleur);
    get_id(&specialite);
    if (store->travailleurs.inserted < TRAVAILLEURS_SIZE) {
        int travailleurExistant = getIndex_trv(&store->travailleurs, travailleur);
        if(travailleurExistant >= 0) {
            int indexSpe = getIndex_spe(&store->specialites, specialite);
            store->travailleurs.table[travailleurExistant].tag_specialite[indexSpe] = VRAI;
        }
        else {
            strcpy(store->travailleurs.table[store->travailleurs.inserted].nom, travailleur);
            int indexSpe = getIndex_spe(&store->specialites, specialite);
            for (unsigned int i = 0; i < SPECIALITE_SIZE; ++i) {
                store->travailleurs.table[store->travailleurs.inserted]
                        .tag_specialite[i] = (i == indexSpe) ? VRAI : FAUX;
```

```
                // Nécessité d'initialiser chaque cases pour éviter des bugs.
            }

            store->travailleurs.inserted++;
        }
    }
}

/*
* traite_travailleurs()
* travailleurs <Mot specialite = "tous">
* Affiche les travailleurs selon la specialité choisie ou non
*/
void traite_travailleurs(Stockage* store) {
    Mot specialite;
    get_id(&specialite);
    if (strcmp(specialite, "tous") == 0) {
        for (int specialitesI = 0; specialitesI < store->specialites.inserted; +
+specialitesI) {
            printf(MSG_TRAVAILLEURS, store->specialites.table[specialitesI].nom);
            int passedCheck = 0;
            for (int travailleursI = 0; store->travailleurs.inserted < travailleursI; +
+travailleursI) {
                if (store->travailleurs.table[travailleursI].tag_specialite[specialitesI] ==
VRAI) {
                    if (passedCheck == 0)
                        printf("%s", store->travailleurs.table[travailleursI].nom);
                    else
                        printf(", %s", store->travailleurs.table[travailleursI].nom);
                    passedCheck++;
                }
            }
            printf("\n");
        }
    }
    else {
        printf(MSG_TRAVAILLEURS, specialite);
        int indexSpe = getIndex_spe(&store->specialites, specialite);
        int passedCheck = 0;
        for (int travailleursI = 0; store->travailleurs.inserted < travailleursI; +
+travailleursI) {
            if (store->travailleurs.table[travailleursI].tag_specialite[indexSpe] == VRAI) {
                if (passedCheck == 0)
                    printf("%s", store->travailleurs.table[travailleursI].nom);
                else
                    printf(", %s", store->travailleurs.table[travailleursI].nom);
                passedCheck++;
            }
        }
        printf("\n");
    }
}

/*
* traite_demarche()
* demarche <Mot nom_client>
* Ajoute un client au programme
*/
void traite_demarche(Stockage* store) {
    Mot nom_client;
    get_id(&nom_client);
    if (store->clients.inserted < CLIENTS_SIZE) {
        strcpy(store->clients.table[store->clients.inserted].nom, &nom_client);
        store->clients.inserted++;
    }
}
```

```c
/*
* traite_client()
* client <Mot nom_client>
* Affiche les commandes d'un client
*/
void traite_client(Stockage* store) {
    Mot nom_client;
    get_id(&nom_client);
    if (strcmp(nom_client, "tous") == 0) {
        for (int clientsI = 0; clientsI < store->clients.inserted; ++clientsI) {
            printf(MSG_CLIENT, store->clients.table[clientsI].nom);
            int passedCheck = 0;
            for (int commandesI = 0; commandesI < store->commandes.inserted; ++commandesI) {
                if (strcmp(store->commandes.table[commandesI].client, store-
>clients.table[clientsI].nom) == 0) {
                    if (passedCheck == 0)
                        printf("%s", store->commandes.table[commandesI].produit);
                    else
                        printf(", %s", store->commandes.table[commandesI].produit);
                    passedCheck++;
                }
            }
            printf("\n");
        }
    }
    else {
        printf(MSG_CLIENT, nom_client);
        int passedCheck = 0;
        for (int i = 0; i < store->commandes.inserted; ++i) {
            if (strcmp(store->commandes.table[i].client, nom_client) == 0) {
                if (passedCheck == 0)
                    printf("%s", store->commandes.table[i].produit);
                else
                    printf(", %s", store->commandes.table[i].produit);
                passedCheck++;
            }
        }
        printf("\n");
    }
}

/*
* traite_commande()
* commande <Mot produit> <Mot nom_client>
* Ajoute une commande au programme
*/
void traite_commande(Stockage* store) {
    Mot produit, nom_client;
    get_id(&produit);
    get_id(&nom_client);
    printf(MSG_COMMANDE, produit, nom_client);
}

/*
* traite_supervision()
* supervision
* Affiche l'avancement actuel des commandes
*/
void traite_supervision(Stockage* store) {
    printf(MSG_SUPERVISION);
}
```

```c
/*
* traite_tache()
* tache <Mot produit> <Mot specialite> <int heures>
* Ajoute une specialité ainsi que le nombre d'heures requises à une commande
*/
void traite_tache(Stockage* store) {
    Mot produit, nom_client;
    get_id(&produit);
    get_id(&nom_client);
    int heures = get_int();
    printf(MSG_TACHE, produit, nom_client, heures);
}

/*
* traite_charge()
* charge <Mot nom_travailleur>
* A
*/
void traite_charge(Stockage* store) {
    Mot nom_travailleur;
    get_id(&nom_travailleur);
    printf(MSG_CHARGE, nom_travailleur);
}

/*
* traite_progression()
* tache <Mot produit> <Mot specialite> <int heures_travaillees>
* Enregistre une progression aux seins d'une specialité d'une commande
*/
void traite_progression(Stockage* store) {
    Mot produit, nom_client;
    get_id(&produit);
    get_id(&nom_client);
    int heures_travaillees = get_int();
    printf(MSG_PROGRESSION, produit, nom_client, heures_travaillees);
}

/*
* traite_passe()
* passe
* Traitement des réallocations des dernières progressions
*/
void traite_passe() {
    printf(MSG_PASSE);
}

/*
* traite_interruption()
* interruption
* Interromp le programme
*/
void traite_interruption() {
    printf(MSG_INTERRUPTION);
}
```

```c
int main(int argc, char* argv[]) {
    if (argc >= 2 && strcmp("echo", argv[1]) == 0)
        EchoActif = VRAI;
    setlocale(LC_ALL, "fr-FR");
    Stockage globalStore;
    globalStore.specialites.inserted = 0;
    globalStore.travailleurs.inserted = 0;
    globalStore.clients.inserted = 0;
    globalStore.commandes.inserted = 0;
    Mot buffer;
    while (VRAI) {
        get_id(&buffer);
        if (strcmp(buffer, "developpe") == 0) {
            traite_developpe(&globalStore);
            continue;
        }
        if (strcmp(buffer, "specialites") == 0) {
            traite_specialites(&globalStore);
            continue;
        }
        if (strcmp(buffer, "embauche") == 0) {
            traite_embauche(&globalStore);
            continue;
        }
        if (strcmp(buffer, "travailleurs") == 0) {
            traite_travailleurs(&globalStore);
            continue;
        }
        if (strcmp(buffer, "demarche") == 0) {
            traite_demarche(&globalStore);
            continue;
        }
        if (strcmp(buffer, "client") == 0) {
            traite_client(&globalStore);
            continue;
        }
        if (strcmp(buffer, "commande") == 0) {
            traite_commande(&globalStore);
            continue;
        }
        if (strcmp(buffer, "supervision") == 0) {
            traite_supervision(&globalStore);
            continue;
        }
        if (strcmp(buffer, "tache") == 0) {
            traite_tache(&globalStore);
            continue;
        }
        if (strcmp(buffer, "charge") == 0) {
            traite_charge(&globalStore);
            continue;
        }
        if (strcmp(buffer, "progression") == 0) {
            traite_progression(&globalStore);
            continue;
        }
        if (strcmp(buffer, "passe") == 0) {
            traite_passe();
            continue;
        }
        if (strcmp(buffer, "interruption") == 0) {
            traite_interruption();
            break;
        }
        printf("!!! instruction inconnue >%s< !!!\n", buffer);
    }
    return 0;
}
```

```c
/*
   sprint3-r
*/

#pragma warning(disable:4996)
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <locale.h>

// Messages emis par les instructions

#define MSG_DEVELOPPE "## nouvelle specialite \"%s\" ; cout horaire \"%d\"\n"
#define MSG_SPECIALITES "specialites traitees : "
#define MSG_INTERRUPTION "## fin de programme\n"
#define MSG_TRAVAILLEURS "la specialite %s peut etre prise en charge par : "
#define MSG_CLIENT "le client %s a commande : "
#define MSG_COMMANDE "## nouvelle commande \"%s\", par client \"%s\"\n"
#define MSG_SUPERVISION "## consultation de l'avancement des commandes\n"
#define MSG_TACHE "## la commande \"%s\" requiere la specialite \"%s\" (nombre
d'heures \"%d\")\n"
#define MSG_CHARGE   "## consultation de la charge de travail de \"%s\"\n"
#define MSG_PROGRESSION "## pour la commande \"%s\", pour la specialite \"%s\" : \"%d\"
heures de plus ont ete realisees\n"
#define MSG_PASSE "## une reallocation est requise\n"

// Lexemes

#define LGMOT 35
#define NBCHIFFREMAX 5
#define MAX_COMMANDES 500
#define MAX_SPECIALITES 10
typedef char Mot[LGMOT + 1]; // Définition du type Mot

typedef enum { FAUX = 0, VRAI = 1 } Booleen;
Booleen EchoActif = FAUX;

/**
   void get_id
   @param id Mot:
**/
void get_id(Mot* id) {
    scanf("%s", id);
    if (EchoActif) printf(">>echo %s\n", id);
}

/**
   int get_int()
   get the current word from the input, and returns it with the (int) type
**/
int get_int() {
    char buffer[NBCHIFFREMAX + 1];
    scanf("%s", buffer);
    if (EchoActif) printf(">>echo %s\n", buffer);
    return atoi(buffer);
}

// Stockage
const enum {
    SPECIALITE_SIZE = 10,
    TRAVAILLEURS_SIZE = 50,
    CLIENTS_SIZE = 100,
    COMMANDES_SIZE = 500
};
```

```c
// Specialité

typedef struct {
    Mot nom;
    int cout_horaire;
} Specialite;
typedef struct {
    Specialite table[SPECIALITE_SIZE];
    int inserted;
} Specialites;

// Travailleurs

typedef struct {
    Mot nom;
    Booleen tag_specialite[SPECIALITE_SIZE];
} Travailleur;
typedef struct {
    Travailleur table[TRAVAILLEURS_SIZE];
    int inserted;
} Travailleurs;

// Client

typedef struct {
    Mot nom;
} Client;
typedef struct {
    Client table[CLIENTS_SIZE];
    int inserted;
} Clients;

// Commande et taches

typedef  struct {
    unsigned int nb_heures_requises;
    unsigned int nb_heures_effectuees;
} Tache;
typedef struct {
    Mot produit;
    Mot nom_client;
    Tache liste_taches[SPECIALITE_SIZE];
} Commande;
typedef struct {
    Commande table[COMMANDES_SIZE];
    int inserted;
} Commandes;

// Stockage global

typedef struct {
    Specialites specialites;
    Travailleurs travailleurs;
    Clients clients;
    Commandes commandes;
} Stockage;

// Helpers

int getIndex_cmd(Commandes* commandes, Mot nom_produit) {
    for (unsigned int i = 0; i < commandes->inserted; i++) {
        if (strcmp(commandes->table[i].produit, nom_produit) == 0) {
            return i;
        }
    }
    return 0; //fallback
}
```

```c
int getIndex_trv(Travailleurs* travailleurs, Mot nom) {
    for (unsigned int i = 0; i < travailleurs->inserted; i++) {
        if (strcmp(travailleurs->table[i].nom, nom) == 0) {
            return i;
        }
    }
    return -1; //fallback
}

int getIndex_spe(Specialites* specialites, Mot nom) {
    for (unsigned int i = 0; i < specialites->inserted; i++) {
        if (strcmp(specialites->table[i].nom, nom) == 0) {
            return i;
        }
    }
    return 0; //fallback
}

// Commandes

/*
* traite_developpe()
* developpe <Mot nom_specialite> <int cout_horaire>
* Ajoute une specialité au programme
*/
void traite_developpe(Stockage* store) {
    Mot nom_specialite;
    get_id(&nom_specialite);
    int cout_horaire = get_int();
    if (store->specialites.inserted < SPECIALITE_SIZE) {
        strcpy(store->specialites.table[store->specialites.inserted].nom, nom_specialite);
        store->specialites.table[store->specialites.inserted].cout_horaire = cout_horaire;
        store->specialites.inserted++;
    }
}

/*
* traite_specialites()
* specialites
* Affiche les specialités enregistrées dans le programme
*/
void traite_specialites(Stockage* store) {
    printf(MSG_SPECIALITES);
    for (int i = 0; i < store->specialites.inserted; ++i) {
        if (i == 0)
            printf("%s/%d", store->specialites.table[i].nom, store-
>specialites.table[i].cout_horaire);
        else
            printf(", %s/%d", store->specialites.table[i].nom, store-
>specialites.table[i].cout_horaire);
    }
    printf("\n");
}

/*
* traite_embauche() : traite les arguments de la commande suivante :
* embauche <Mot travailleur> <Mot specialite>
*/
void traite_embauche(Stockage* store) {
    Mot travailleur, specialite;
    get_id(&travailleur);
    get_id(&specialite);
    if (store->travailleurs.inserted < TRAVAILLEURS_SIZE) {
        int travailleurExistant = getIndex_trv(&store->travailleurs, travailleur);
        if(travailleurExistant >= 0) {
            int indexSpe = getIndex_spe(&store->specialites, specialite);
            store->travailleurs.table[travailleurExistant].tag_specialite[indexSpe] = VRAI;
        }
```

```
        else {
            strcpy(store->travailleurs.table[store->travailleurs.inserted].nom, travailleur);
            int indexSpe = getIndex_spe(&store->specialites, specialite);
            for (unsigned int i = 0; i < SPECIALITE_SIZE; ++i) {
                store->travailleurs.table[store->travailleurs.inserted]
                        .tag_specialite[i] = (i == indexSpe) ? VRAI : FAUX;
                // Necessité d'initiliaser chaque cases pour éviter des bugs.
            }
            store->travailleurs.inserted++;
        }
    }
}

/*
* traite_travailleurs()
* travailleurs <Mot specialite = "tous">
* Affiche les travailleurs selon la specialité choisie ou non
*/
void traite_travailleurs(Stockage* store) {
    Mot specialite;
    get_id(&specialite);
    if (strcmp(specialite, "tous") == 0) {
        for (int specialitesI = 0; specialitesI < store->specialites.inserted; +
+specialitesI) {
            printf(MSG_TRAVAILLEURS, store->specialites.table[specialitesI].nom);
            int passedCheck = 0;
            for (int travailleursI = store->travailleurs.inserted; travailleursI >= 0;
--travailleursI) {
                if (store->travailleurs.table[travailleursI].tag_specialite[specialitesI] ==
VRAI) {
                    if (passedCheck == 0)
                        printf("%s", store->travailleurs.table[travailleursI].nom);
                    else
                        printf(", %s", store->travailleurs.table[travailleursI].nom);
                    passedCheck++;
                }
            }
            printf("\n");
        }
    }
    else {
        printf(MSG_TRAVAILLEURS, specialite);
        int indexSpe = getIndex_spe(&store->specialites, specialite);
        int passedCheck = 0;
        for (int travailleursI = store->travailleurs.inserted; travailleursI >= 0;
--travailleursI) {
            if (store->travailleurs.table[travailleursI].tag_specialite[indexSpe] == VRAI) {
                if (passedCheck == 0)
                    printf("%s", store->travailleurs.table[travailleursI].nom);
                else
                    printf(", %s", store->travailleurs.table[travailleursI].nom);
                passedCheck++;
            }
        }
        printf("\n");
    }
}

/*
* traite_demarche()
* demarche <Mot nom_client>
* Ajoute un client au programme
*/
void traite_demarche(Stockage* store) {
    Mot nom_client;
    get_id(&nom_client);
    if (store->clients.inserted < CLIENTS_SIZE) {
        strcpy(store->clients.table[store->clients.inserted].nom, nom_client);
```

```
        store->clients.inserted++;
    }
}

/*
* traite_client()
* client <Mot nom_client>
* Affiche les commandes d'un client
*/
void traite_client(Stockage* store) {
    Mot nom_client;
    get_id(&nom_client);
    if (strcmp(nom_client, "tous") == 0) {
        for (int clientsI = 0; clientsI < store->clients.inserted; ++clientsI) {
            printf(MSG_CLIENT, store->clients.table[clientsI].nom);
            int passedCheck = 0;
            for (int commandesI = 0; commandesI < store->commandes.inserted; ++commandesI) {
                if (strcmp(store->commandes.table[commandesI].nom_client, store-
>clients.table[clientsI].nom) == 0) {
                    if (passedCheck == 0)
                        printf("%s", store->commandes.table[commandesI].produit);
                    else
                        printf(", %s", store->commandes.table[commandesI].produit);
                    passedCheck++;
                }
            }
            printf("\n");
        }
    }
    else {
        printf(MSG_CLIENT, nom_client);
        int passedCheck = 0;
        for (int i = 0; i < store->commandes.inserted; ++i) {
            if (strcmp(store->commandes.table[i].nom_client, nom_client) == 0) {
                if (passedCheck == 0)
                    printf("%s", store->commandes.table[i].produit);
                else
                    printf(", %s", store->commandes.table[i].produit);
                passedCheck++;
            }
        }
        printf("\n");
    }
}

/*
* traite_commande()
* commande <Mot produit> <Mot nom_client>
* Ajoute une commande au programme
*/
void traite_commande(Stockage* store) {
    Mot produit, nom_client;
    get_id(&produit);
    get_id(&nom_client);
    const unsigned int i = store->commandes.inserted++; // stockage puis incrémentation
    strcpy(store->commandes.table[i].nom_client, nom_client);
    strcpy(store->commandes.table[i].produit, produit);
    //initialisation de la liste de tâches
    for (unsigned int speNbr = 0; speNbr < MAX_SPECIALITES; speNbr++)
    {
        store->commandes.table[i].liste_taches[speNbr].nb_heures_requises = 0;
        store->commandes.table[i].liste_taches[speNbr].nb_heures_effectuees = 0;
    }
}
```

```c
/*
* traite_supervision()
* Affiche l'avancement actuel des commandes sous le format
* OUT : "etat des taches pour <nom_produit> : [<liste>, ..., <liste>]"
*     avec <liste> valant "<specialite>:<heures effectuées>/<heures nécessaires>"
*/
void traite_supervision(Stockage* store) {
    for (int i_cmd = 0; i_cmd < store->commandes.inserted; i_cmd++)
    {
        printf("etat des taches pour %s : ", store->commandes.table[i_cmd].produit);
        Booleen isFirstTime = VRAI;
        for (int i_spe = 0; i_spe < MAX_SPECIALITES; i_spe++)
        {
            const Tache tacheCourante = store->commandes.table[i_cmd].liste_taches[i_spe];
            if (tacheCourante.nb_heures_requises) { // valeur si vide : 0 = faux
                if (isFirstTime) {
                    isFirstTime = FAUX;
                }
                else {
                    printf(", ");
                }
                printf("%s:", store->specialites.table[i_spe].nom);
                printf("%d/%d", tacheCourante.nb_heures_effectuees,
tacheCourante.nb_heures_requises);
            }
        }
        printf("\n");
    }
}

/*
* traite_tache()
* tache <Mot commande> <Mot specialite> <int heures>
* Ajoute une specialité ainsi que le nombre d'heures requises à une commande
*/
void traite_tache(Stockage* store) {
    Mot commande, specialite;
    get_id(&commande);
    get_id(&specialite);
    int heures = get_int();
    // recherche de la commande concernée puis de la spécialité concernée
    const unsigned int cmd_i = getIndex_cmd(&store->commandes, commande);
    const unsigned int id_spe = getIndex_spe(&store->specialites, specialite);
    // initialisation de la tâche pour la commande en question
    store->commandes.table[cmd_i].liste_taches[id_spe].nb_heures_requises = heures;
}

/*
* traite_charge()
* charge <Mot nom_travailleur>
* A
*/
void traite_charge(Stockage* store) {
    Mot nom_travailleur;
    get_id(&nom_travailleur);
    printf(MSG_CHARGE, nom_travailleur);
}

/*
* traite_progression()
* tache <Mot produit> <Mot specialite> <int heures_travaillees>
* Enregistre une progression aux seins d'une specialité d'une commande
*/
void traite_progression(Stockage* store) {
    Mot commande, specialite;
    get_id(&commande);
    get_id(&specialite);
```

```
    int heures_travaillees = get_int();
    const unsigned int cmd_i = getIndex_cmd(&store->commandes, commande);
    const unsigned int id_spe = getIndex_spe(&store->specialites, specialite);
    store->commandes.table[cmd_i].liste_taches[id_spe].nb_heures_effectuees +=
heures_travaillees;
}

/*
* traite_passe()
* passe
* Traitement des réallocations des dernières progressions
*/
void traite_passe() {
    //printf(MSG_PASSE);
}

/*
* traite_interruption()
* interruption
* Interromp le programme
*/
void traite_interruption() {
    printf(MSG_INTERRUPTION);
}

// Porte d'entrée du programme

int main(int argc, char* argv[]) {
    if (argc >= 2 && strcmp("echo", argv[1]) == 0) {
        EchoActif = VRAI;
    }
    setlocale(LC_ALL, "fr-FR");
    Stockage globalStore;
    globalStore.specialites.inserted = 0;
    globalStore.travailleurs.inserted = 0;
    globalStore.clients.inserted = 0;
    globalStore.commandes.inserted = 0;
    Mot buffer;
    while (VRAI) {
        get_id(&buffer);
        if (strcmp(buffer, "developpe") == 0) {
            traite_developpe(&globalStore);
            continue;
        }
        if (strcmp(buffer, "specialites") == 0) {
            traite_specialites(&globalStore);
            continue;
        }
        if (strcmp(buffer, "embauche") == 0) {
            traite_embauche(&globalStore);
            continue;
        }
        if (strcmp(buffer, "travailleurs") == 0) {
            traite_travailleurs(&globalStore);
            continue;
        }
        if (strcmp(buffer, "demarche") == 0) {
            traite_demarche(&globalStore);
            continue;
        }
        if (strcmp(buffer, "client") == 0) {
            traite_client(&globalStore);
            continue;
        }
        if (strcmp(buffer, "commande") == 0) {
            traite_commande(&globalStore);
            continue;
        }
```

```
        if (strcmp(buffer, "supervision") == 0) {
            traite_supervision(&globalStore);
            continue;
        }
        if (strcmp(buffer, "tache") == 0) {
            traite_tache(&globalStore);
            continue;
        }
        if (strcmp(buffer, "charge") == 0) {
            traite_charge(&globalStore);
            continue;
        }
        if (strcmp(buffer, "progression") == 0) {
            traite_progression(&globalStore);
            continue;
        }
        if (strcmp(buffer, "passe") == 0) {
            traite_passe();
            continue;
        }
        if (strcmp(buffer, "interruption") == 0) {
            traite_interruption();
            break;
        }
        printf("!!! instruction inconnue >%s< !!!\n", buffer);
    }
    return 0;
}
```

```c
/*
      sprint4-r
*/

#pragma warning(disable:4996)
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <locale.h>

// Messages emis par les instructions

#define MSG_DEVELOPPE "## nouvelle specialite \"%s\" ; cout horaire \"%d\"\n"
#define MSG_SPECIALITES "specialites traitees : "
#define MSG_INTERRUPTION "## fin de programme\n"
#define MSG_TRAVAILLEURS "la specialite %s peut etre prise en charge par : "
#define MSG_CLIENT "le client %s a commande : "
#define MSG_COMMANDE "## nouvelle commande \"%s\", par client \"%s\"\n"
#define MSG_SUPERVISION "## consultation de l'avancement des commandes\n"
#define MSG_TACHE "## la commande \"%s\" requiere la specialite \"%s\" (nombre
d'heures \"%d\")\n"
#define MSG_CHARGE   "charge de travail pour %s : "
#define MSG_PROGRESSION "## pour la commande \"%s\", pour la specialite \"%s\" : \"%d\"
heures de plus ont ete realisees\n"
#define MSG_PASSE "## une reallocation est requise\n"

// Lexemes

#define LGMOT 35
#define NBCHIFFREMAX 5
#define MAX_COMMANDES 500
#define MAX_SPECIALITES 10
typedef char Mot[LGMOT + 1]; // Définition du type Mot

typedef enum { FAUX = 0, VRAI = 1 } Booleen;
Booleen EchoActif = FAUX;

/**
   void get_id
   @param id Mot:
**/
void get_id(Mot* id) {
    scanf("%s", id);
    if (EchoActif) printf(">>echo %s\n", id);
}

/**
   int get_int()
   get the current word from the input, and returns it with the (int) type
**/
int get_int() {
    char buffer[NBCHIFFREMAX + 1];
    scanf("%s", buffer);
    if (EchoActif) printf(">>echo %s\n", buffer);
    return atoi(buffer);
}

// Stockage

const enum {
    SPECIALITE_SIZE = 10,
    TRAVAILLEURS_SIZE = 50,
    CLIENTS_SIZE = 100,
    COMMANDES_SIZE = 500
};
```

```c
// Specialité

typedef struct {
    Mot nom;
    int cout_horaire;
} Specialite;
typedef struct {
    Specialite table[SPECIALITE_SIZE];
    int inserted;
} Specialites;

// Travailleurs

typedef struct {
    Mot nom;
    Booleen tag_specialite[SPECIALITE_SIZE];
} Travailleur;
typedef struct {
    Travailleur table[TRAVAILLEURS_SIZE];
    int inserted;
} Travailleurs;

// Client

typedef struct {
    Mot nom;
} Client;
typedef struct {
    Client table[CLIENTS_SIZE];
    int inserted;
} Clients;

// Commande et taches

typedef  struct {
    unsigned int nb_heures_requises;
    unsigned int nb_heures_effectuees;
} Tache;
typedef struct {
    Mot produit;
    Mot nom_client;
    Tache liste_taches[SPECIALITE_SIZE];
    // L'identifiant du travailleur en charge de la tache
    unsigned int en_charge_tache[SPECIALITE_SIZE];
} Commande;
typedef struct {
    Commande table[COMMANDES_SIZE];
    int inserted;
} Commandes;

// Stockage global

typedef struct {
    Specialites specialites;
    Travailleurs travailleurs;
    Clients clients;
    Commandes commandes;
} Stockage;
```

```c
// Helpers

int getIndex_cmd(Commandes* commandes, Mot nom_produit) {
    for (unsigned int i = 0; i < commandes->inserted; i++) {
        if (strcmp(commandes->table[i].produit, nom_produit) == 0) {
            return i;
        }
    }
    return -1; //fallback
}

int getIndex_trv(Travailleurs* travailleurs, Mot nom) {
    for (unsigned int i = 0; i < travailleurs->inserted; i++) {
        if (strcmp(travailleurs->table[i].nom, nom) == 0) {
            return i;
        }
    }
    return -1; //fallback
}

int getIndex_spe(Specialites* specialites, Mot nom) {
    for (unsigned int i = 0; i < specialites->inserted; i++) {
        if (strcmp(specialites->table[i].nom, nom) == 0) {
            return i;
        }
    }
    return -1; //fallback
}

// Commandes

/*
* traite_developpe()
* developpe <Mot nom_specialite> <int cout_horaire>
* Ajoute une specialité au programme
*/
void traite_developpe(Stockage* store) {
    Mot nom_specialite;
    get_id(&nom_specialite);
    int cout_horaire = get_int();
    if (store->specialites.inserted < SPECIALITE_SIZE) {
        strcpy(store->specialites.table[store->specialites.inserted].nom, nom_specialite);
        store->specialites.table[store->specialites.inserted].cout_horaire = cout_horaire;
        store->specialites.inserted++;
    }
}

/*
* traite_specialites()
* specialites
* Affiche les specialités enregistrées dans le programme
*/
void traite_specialites(Stockage* store) {
    printf(MSG_SPECIALITES);
    for (int i = 0; i < store->specialites.inserted; ++i) {
        if (i == 0)
            printf("%s/%d", store->specialites.table[i].nom, store-
>specialites.table[i].cout_horaire);
        else
            printf(", %s/%d", store->specialites.table[i].nom, store-
>specialites.table[i].cout_horaire);
    }
    printf("\n");
}
```

```
/*
* traite_embauche() : traite les arguments de la commande suivante :
* embauche <Mot travailleur> <Mot specialite>
*/
void traite_embauche(Stockage* store) {
    Mot travailleur, specialite;
    get_id(&travailleur);
    get_id(&specialite);
    if (store->travailleurs.inserted < TRAVAILLEURS_SIZE) {
        int travailleurExistant = getIndex_trv(&store->travailleurs, travailleur);
        if(travailleurExistant >= 0) {
            int indexSpe = getIndex_spe(&store->specialites, specialite);
            store->travailleurs.table[travailleurExistant].tag_specialite[indexSpe] = VRAI;
        }
        else {
            strcpy(store->travailleurs.table[store->travailleurs.inserted].nom, travailleur);
            int indexSpe = getIndex_spe(&store->specialites, specialite);
            for (unsigned int i = 0; i < SPECIALITE_SIZE; ++i) {
                store->travailleurs.table[store->travailleurs.inserted]
                        .tag_specialite[i] = (i == indexSpe) ? VRAI : FAUX;
                // Necessité d'initiliaser chaque cases pour éviter des bugs.
            }
            store->travailleurs.inserted++;
        }
    }
}

/*
* traite_travailleurs()
* travailleurs <Mot specialite = "tous">
* Affiche les travailleurs selon la specialité choisie ou non
*/
void traite_travailleurs(Stockage* store) {
    Mot specialite;
    get_id(&specialite);
    if (strcmp(specialite, "tous") == 0) {
        for (int specialitesI = 0; specialitesI < store->specialites.inserted; +
+specialitesI) {
            printf(MSG_TRAVAILLEURS, store->specialites.table[specialitesI].nom);
            int passedCheck = 0;
            for (int travailleursI = 0; travailleursI < store->travailleurs.inserted; +
+travailleursI) {
                if (store->travailleurs.table[travailleursI].tag_specialite[specialitesI] ==
VRAI) {
                    if (passedCheck == 0)
                        printf("%s", store->travailleurs.table[travailleursI].nom);
                    else
                        printf(", %s", store->travailleurs.table[travailleursI].nom);
                    passedCheck++;
                }
            }
            printf("\n");
        }
    }
    else {
        printf(MSG_TRAVAILLEURS, specialite);
        int indexSpe = getIndex_spe(&store->specialites, specialite);
        int passedCheck = 0;
        for (int travailleursI = store->travailleurs.inserted; travailleursI >= 0;
--travailleursI) {
            if (store->travailleurs.table[travailleursI].tag_specialite[indexSpe] == VRAI) {
                if (passedCheck == 0)
                    printf("%s", store->travailleurs.table[travailleursI].nom);
                else
                    printf(", %s", store->travailleurs.table[travailleursI].nom);
                passedCheck++;
            }
        }
```

```
            printf("\n");
        }
    }

    /*
    * traite_demarche()
    * demarche <Mot nom_client>
    * Ajoute un client au programme
    */
    void traite_demarche(Stockage* store) {
        Mot nom_client;
        get_id(&nom_client);
        if (store->clients.inserted < CLIENTS_SIZE) {
            strcpy(store->clients.table[store->clients.inserted].nom, nom_client);
            store->clients.inserted++;
        }
    }

    /*
    * traite_client()
    * client <Mot nom_client>
    * Affiche les commandes d'un client
    */
    void traite_client(Stockage* store) {
        Mot nom_client;
        get_id(&nom_client);
        if (strcmp(nom_client, "tous") == 0) {
            for (int clientsI = 0; clientsI < store->clients.inserted; ++clientsI) {
                printf(MSG_CLIENT, store->clients.table[clientsI].nom);
                int passedCheck = 0;
                for (int commandesI = 0; commandesI < store->commandes.inserted; ++commandesI) {
                    if (strcmp(store->commandes.table[commandesI].nom_client, store-
    >clients.table[clientsI].nom) == 0) {
                        if (passedCheck == 0)
                            printf("%s", store->commandes.table[commandesI].produit);
                        else
                            printf(", %s", store->commandes.table[commandesI].produit);
                        passedCheck++;
                    }
                }
                printf("\n");
            }
        }
        else {
            printf(MSG_CLIENT, nom_client);
            int passedCheck = 0;
            for (int i = 0; i < store->commandes.inserted; ++i) {
                if (strcmp(store->commandes.table[i].nom_client, nom_client) == 0) {
                    if (passedCheck == 0)
                        printf("%s", store->commandes.table[i].produit);
                    else
                        printf(", %s", store->commandes.table[i].produit);
                    passedCheck++;
                }
            }
            printf("\n");
        }
    }
```

```
/*
* traite_commande()
* commande <Mot produit> <Mot nom_client>
* Ajoute une commande au programme
*/
void traite_commande(Stockage* store) {
    Mot produit, nom_client;
    get_id(&produit);
    get_id(&nom_client);
    const unsigned int i = store->commandes.inserted++; // stockage puis incrémentation
    strcpy(store->commandes.table[i].nom_client, nom_client);
    strcpy(store->commandes.table[i].produit, produit);
    //initialisation de la liste de tâches
    for (unsigned int speNbr = 0; speNbr < MAX_SPECIALITES; ++speNbr)
    {
        store->commandes.table[i].liste_taches[speNbr].nb_heures_requises = 0;
        store->commandes.table[i].liste_taches[speNbr].nb_heures_effectuees = 0;
    }
}

/*
* traite_supervision()
* Affiche l'avancement actuel des commandes sous le format
* OUT : "etat des taches pour <nom_produit> : [<liste>, ..., <liste>]"
*     avec <liste> valant "<specialite>:<heures effectuées>/<heures nécessaires>"
*/
void traite_supervision(Stockage* store) {
    for (int i_cmd = 0; i_cmd < store->commandes.inserted; ++i_cmd)
    {
        printf("etat des taches pour %s : ", store->commandes.table[i_cmd].produit);
        Booleen isFirstTime = VRAI;
        for (int i_spe = 0; i_spe < store->specialites.inserted; ++i_spe)
        {
            const Tache tacheCourante = store->commandes.table[i_cmd].liste_taches[i_spe];
            if (tacheCourante.nb_heures_requises) { // valeur si vide : 0 = faux
                if (isFirstTime) {
                    isFirstTime = FAUX;
                }
                else {
                    printf(", ");
                }
                printf("%s:", store->specialites.table[i_spe].nom);
                printf("%d/%d", tacheCourante.nb_heures_effectuees,
tacheCourante.nb_heures_requises);
            }
        }
        printf("\n");
    }
}

/**
 * Détermine le travailleur le plus adapté pour la réalisation d'une tâche.
 * Une tâche étant déterminée par une spécialité, il attends donc en ENTREE
 * l'identifiant d'une spécialitée attendue, et afin de faire ses opérations
 * le pointeur du tableau de travailleurs
 * En SORTIE, il retourne l'index du travailleur à prendre en charge.
**/
unsigned int determiner_travailleur_pour(int id_spe, Stockage* store) {
    unsigned int retval = TRAVAILLEURS_SIZE;
    int totalWorker[TRAVAILLEURS_SIZE];
    for (int i = 0; i < TRAVAILLEURS_SIZE; ++i) totalWorker[i] = 0;
    for (int id_worker = 0; id_worker < store->travailleurs.inserted; ++id_worker) {
        for (int i_cmd = 0; i_cmd < store->commandes.inserted; ++i_cmd) {
            for (int i_spe = 0; i_spe < store->specialites.inserted; ++i_spe) {
                if (store->commandes.table[i_cmd].en_charge_tache[i_spe] == id_worker) {
                    totalWorker[id_worker] += store-
```

```
>commandes.table[i_cmd].liste_taches[i_spe].nb_heures_requises - store-
>commandes.table[i_cmd].liste_taches[i_spe].nb_heures_effectuees;
                }
            }
        }
    }
    int lowestHours = -1;
    for (int id_worker = 0; id_worker < store->travailleurs.inserted; ++id_worker) {
        if (store->travailleurs.table[id_worker].tag_specialite[id_spe] == VRAI) {
            if (lowestHours < 0) {
                lowestHours = totalWorker[id_worker];
            }
            if (EchoActif)
                printf(">>> >>> %s %d: %d (%d)\n", store->travailleurs.table[id_worker].nom,
id_worker, totalWorker[id_worker], lowestHours);
            if (lowestHours >= totalWorker[id_worker]) {
                retval = id_worker;
                lowestHours = totalWorker[id_worker];
            }
        }
    }
    return retval;
}

/*
* traite_tache()
* tache <Mot commande> <Mot specialite> <int heures>
* Ajoute une specialité ainsi que le nombre d'heures requises à une commande
*/
void traite_tache(Stockage* store) {
    Mot commande, specialite;
    get_id(&commande);
    get_id(&specialite);
    int heures = get_int();
    // recherche de la commande concernée puis de la spécialité concernée
    const unsigned int cmd_i = getIndex_cmd(&store->commandes, commande);
    const unsigned int id_spe = getIndex_spe(&store->specialites, specialite);
    // initialisation de la tâche pour la commande en question
    store->commandes.table[cmd_i].liste_taches[id_spe].nb_heures_requises = heures;
    // Assignation du travailleur à la tache
    const unsigned int id_worker = determiner_travailleur_pour(id_spe, store);
    if (id_worker < TRAVAILLEURS_SIZE) {
        store->commandes.table[cmd_i].en_charge_tache[id_spe] = id_worker;
    } else if(EchoActif) {
        printf("$ Erreur : aucun travailleur trouvé pour traiter la spécialité demandée.\n");
    }
    // Il sera nécessaire de stoker le nombre de [tâches||d'heures] au bout d'une
    // affectation de tâche afin de trouver par la suite le meilleur travailleur
    // pour la prochaine tâche
    // pour la beta : set_total_tasks_for(Travailleur* travailleur);
    // pour la release : set_total_hours_for(Travailleur* travailleur);
}

/*
* traite_charge()
* charge <Mot nom_travailleur>
* A
*/
void traite_charge(Stockage* store) {
    Mot nom_travailleur;
    get_id(&nom_travailleur);
    // printf(MSG_CHARGE, nom_travailleur);
    // superProduit/reseau/45heure(s)
    unsigned int travailleursId = getIndex_trv(&store->travailleurs, nom_travailleur);
    printf("charge de travail pour %s : ", nom_travailleur);
    Booleen isFirstTime = VRAI;
    for (int commandesI = 0; commandesI < store->commandes.inserted; commandesI++) {
        for (int specialitesI = 0; specialitesI < SPECIALITE_SIZE; specialitesI++) {
```

```
            const Tache tache = store->commandes.table[commandesI].liste_taches[specialitesI];
            if (store->commandes.table[commandesI].en_charge_tache[specialitesI] == travailleursId && tache.nb_heures_requises != tache.nb_heures_effectuees) {
                if (isFirstTime) {
                    isFirstTime = FAUX;
                } else {
                    printf(", ");
                }
                printf("%s/%s/%dheure(s)", store->commandes.table[commandesI], store->specialites.table[specialitesI].nom, tache.nb_heures_requises - tache.nb_heures_effectuees);
            }
        }
    }
    printf("\n");
}

/*
* traite_progression()
* tache <Mot produit> <Mot specialite> <int heures_travaillees>
* Enregistre une progression aux seins d'une specialité d'une commande
*/
void traite_progression(Stockage* store) {
    Mot commande, specialite;
    get_id(&commande);
    get_id(&specialite);
    int heures_travaillees = get_int();
    const unsigned int cmd_i = getIndex_cmd(&store->commandes, commande);
    const unsigned int id_spe = getIndex_spe(&store->specialites, specialite);
    store->commandes.table[cmd_i].liste_taches[id_spe].nb_heures_effectuees += heures_travaillees;
}

/*
* traite_passe()
* passe
* Traitement des réallocations des dernières progressions
*/
void traite_passe() {
    //printf(MSG_PASSE);
}

/*
* traite_interruption()
* interruption
* Interromp le programme
*/
void traite_interruption() {
    printf(MSG_INTERRUPTION);
}
// Porte d'entrée du programme
int main(int argc, char* argv[]) {
    if (argc >= 2 && strcmp("echo", argv[1]) == 0) {
        EchoActif = VRAI;
    }
    setlocale(LC_ALL, "fr-FR");
    Stockage globalStore;
    globalStore.specialites.inserted = 0;
    globalStore.travailleurs.inserted = 0;
    globalStore.clients.inserted = 0;
    globalStore.commandes.inserted = 0;
    Mot buffer;
    while (VRAI) {
        get_id(&buffer);
        if (strcmp(buffer, "developpe") == 0) {
            traite_developpe(&globalStore);
            continue;
        }
```

```
        if (strcmp(buffer, "specialites") == 0) {
            traite_specialites(&globalStore);
            continue;
        }
        if (strcmp(buffer, "embauche") == 0) {
            traite_embauche(&globalStore);
            continue;
        }
        if (strcmp(buffer, "travailleurs") == 0) {
            traite_travailleurs(&globalStore);
            continue;
        }
        if (strcmp(buffer, "demarche") == 0) {
            traite_demarche(&globalStore);
            continue;
        }
        if (strcmp(buffer, "client") == 0) {
            traite_client(&globalStore);
            continue;
        }
        if (strcmp(buffer, "commande") == 0) {
            traite_commande(&globalStore);
            continue;
        }
        if (strcmp(buffer, "supervision") == 0) {
            traite_supervision(&globalStore);
            continue;
        }
        if (strcmp(buffer, "tache") == 0) {
            traite_tache(&globalStore);
            continue;
        }
        if (strcmp(buffer, "charge") == 0) {
            traite_charge(&globalStore);
            continue;
        }
        if (strcmp(buffer, "progression") == 0) {
            traite_progression(&globalStore);
            continue;
        }
        if (strcmp(buffer, "passe") == 0) {
            traite_passe();
            continue;
        }
        if (strcmp(buffer, "interruption") == 0) {
            traite_interruption();
            break;
        }
        printf("!!! instruction inconnue >%s< !!!\n", buffer);
    }
    return 0;
}
```

```c
/*
      sprint5-r
*/

#pragma warning(disable:4996)
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <locale.h>

// Messages emis par les instructions

#define MSG_DEVELOPPE "## nouvelle specialite \"%s\" ; cout horaire \"%d\"\n"
#define MSG_SPECIALITES "specialites traitees : "
#define MSG_INTERRUPTION "## fin de programme\n"
#define MSG_TRAVAILLEURS "la specialite %s peut etre prise en charge par : "
#define MSG_CLIENT "le client %s a commande : "
#define MSG_COMMANDE "## nouvelle commande \"%s\", par client \"%s\"\n"
#define MSG_SUPERVISION "## consultation de l'avancement des commandes\n"
#define MSG_TACHE "## la commande \"%s\" requiere la specialite \"%s\" (nombre
d'heures \"%d\")\n"
#define MSG_CHARGE   "charge de travail pour %s : "
#define MSG_PROGRESSION "## pour la commande \"%s\", pour la specialite \"%s\" : \"%d\"
heures de plus ont ete realisees\n"
#define MSG_PASSE "## une reallocation est requise\n"

// Lexemes

#define LGMOT 35
#define NBCHIFFREMAX 5
#define MAX_COMMANDES 500
#define MAX_SPECIALITES 10
typedef char Mot[LGMOT + 1]; // Définition du type Mot
typedef unsigned long long Gros_entier;

typedef enum { FAUX = 0, VRAI = 1 } Booleen;
Booleen EchoActif = FAUX;

/**
   void get_id
   @param id Mot:
**/
void get_id(Mot* id) {
    scanf("%s", id);
    if (EchoActif) printf(">>echo %s\n", id);
}

/**
   int get_int()
   get the current word from the input, and returns it with the (int) type
**/
int get_int() {
    char buffer[NBCHIFFREMAX + 1];
    scanf("%s", buffer);
    if (EchoActif) printf(">>echo %s\n", buffer);
    return atoi(buffer);
}

// Stockage

const enum {
    SPECIALITE_SIZE = 10,
    TRAVAILLEURS_SIZE = 50,
    CLIENTS_SIZE = 100,
    COMMANDES_SIZE = 500
};
```

```c
// Specialité

typedef struct {
    Mot nom;
    int cout_horaire;
} Specialite;
typedef struct {
    Specialite table[SPECIALITE_SIZE];
    int inserted;
} Specialites;

// Travailleurs

typedef struct {
    Mot nom;
    Booleen tag_specialite[SPECIALITE_SIZE];
} Travailleur;
typedef struct {
    Travailleur table[TRAVAILLEURS_SIZE];
    int inserted;
} Travailleurs;

// Client

typedef struct {
    Mot nom;
} Client;
typedef struct {
    Client table[CLIENTS_SIZE];
    int inserted;
} Clients;

// Commande et taches

typedef  struct {
    unsigned int nb_heures_requises;
    unsigned int nb_heures_effectuees;
} Tache;
typedef struct {
    Mot produit;
    Mot nom_client;
    Tache liste_taches[SPECIALITE_SIZE];
    Booleen complete;
    // L'identifiant du travailleur en charge de la tache
    unsigned int en_charge_tache[SPECIALITE_SIZE];
} Commande;
typedef struct {
    Commande table[COMMANDES_SIZE];
    int inserted;
} Commandes;

// Stockage global

typedef struct {
    Specialites specialites;
    Travailleurs travailleurs;
    Clients clients;
    Commandes commandes;
} Stockage;
```

```c
// Helpers

int getIndex_cmd(Commandes* commandes, Mot nom_produit) {
    for (unsigned int i = 0; i < commandes->inserted; i++) {
        if (strcmp(commandes->table[i].produit, nom_produit) == 0) {
            return i;
        }
    }
    return -1; //fallback
}

int getIndex_trv(Travailleurs* travailleurs, Mot nom) {
    for (unsigned int i = 0; i < travailleurs->inserted; i++) {
        if (strcmp(travailleurs->table[i].nom, nom) == 0) {
            return i;
        }
    }
    return -1; //fallback
}

int getIndex_spe(Specialites* specialites, Mot nom) {
    for (unsigned int i = 0; i < specialites->inserted; i++) {
        if (strcmp(specialites->table[i].nom, nom) == 0) {
            return i;
        }
    }
    return -1; //fallback
}

// Commandes

/*
* traite_developpe()
* developpe <Mot nom_specialite> <int cout_horaire>
* Ajoute une specialité au programme
*/
void traite_developpe(Stockage* store) {
    Mot nom_specialite;
    get_id(&nom_specialite);
    int cout_horaire = get_int();
    if (store->specialites.inserted < SPECIALITE_SIZE) {
        strcpy(store->specialites.table[store->specialites.inserted].nom, nom_specialite);
        store->specialites.table[store->specialites.inserted].cout_horaire = cout_horaire;
        store->specialites.inserted++;
    }
}

/*
* traite_specialites()
* specialites
* Affiche les specialités enregistrées dans le programme
*/
void traite_specialites(Stockage* store) {
    printf(MSG_SPECIALITES);
    for (int i = 0; i < store->specialites.inserted; ++i) {
        if (i == 0)
            printf("%s/%d", store->specialites.table[i].nom, store-
>specialites.table[i].cout_horaire);
        else
            printf(", %s/%d", store->specialites.table[i].nom, store-
>specialites.table[i].cout_horaire);
    }
    printf("\n");
}
```

```
/*
* traite_embauche() : traite les arguments de la commande suivante :
* embauche <Mot travailleur> <Mot specialite>
*/
void traite_embauche(Stockage* store) {
    Mot travailleur, specialite;
    get_id(&travailleur);
    get_id(&specialite);
    if (store->travailleurs.inserted < TRAVAILLEURS_SIZE) {
        int travailleurExistant = getIndex_trv(&store->travailleurs, travailleur);
        if(travailleurExistant >= 0) {
            int indexSpe = getIndex_spe(&store->specialites, specialite);
            store->travailleurs.table[travailleurExistant].tag_specialite[indexSpe] = VRAI;
        }
        else {
            strcpy(store->travailleurs.table[store->travailleurs.inserted].nom, travailleur);
            int indexSpe = getIndex_spe(&store->specialites, specialite);
            for (unsigned int i = 0; i < SPECIALITE_SIZE; ++i) {
                store->travailleurs.table[store->travailleurs.inserted]
                        .tag_specialite[i] = (i == indexSpe) ? VRAI : FAUX;
                // Necessité d'initiliaser chaque cases pour éviter des bugs.
            }
            store->travailleurs.inserted++;
        }
    }
}

/*
* traite_travailleurs()
* travailleurs <Mot specialite = "tous">
* Affiche les travailleurs selon la specialité choisie ou non
*/
void traite_travailleurs(Stockage* store) {
    Mot specialite;
    get_id(&specialite);
    if (strcmp(specialite, "tous") == 0) {
        for (int specialitesI = 0; specialitesI < store->specialites.inserted; +
+specialitesI) {
            printf(MSG_TRAVAILLEURS, store->specialites.table[specialitesI].nom);
            int passedCheck = 0;
            for (int travailleursI = 0; travailleursI < store->travailleurs.inserted; +
+travailleursI) {
                if (store->travailleurs.table[travailleursI].tag_specialite[specialitesI] ==
VRAI) {
                    if (passedCheck == 0)
                        printf("%s", store->travailleurs.table[travailleursI].nom);
                    else
                        printf(", %s", store->travailleurs.table[travailleursI].nom);
                    passedCheck++;
                }
            }
            printf("\n");
        }
    }
    else {
        printf(MSG_TRAVAILLEURS, specialite);
        int indexSpe = getIndex_spe(&store->specialites, specialite);
        int passedCheck = 0;
        for (int travailleursI = store->travailleurs.inserted; travailleursI >= 0;
--travailleursI) {
            if (store->travailleurs.table[travailleursI].tag_specialite[indexSpe] == VRAI) {
                if (passedCheck == 0)
                    printf("%s", store->travailleurs.table[travailleursI].nom);
                else
                    printf(", %s", store->travailleurs.table[travailleursI].nom);
                passedCheck++;
            }
```

```
        }
        printf("\n");
    }
}

/*
* traite_demarche()
* demarche <Mot nom_client>
* Ajoute un client au programme
*/
void traite_demarche(Stockage* store) {
    Mot nom_client;
    get_id(&nom_client);
    if (store->clients.inserted < CLIENTS_SIZE) {
        strcpy(store->clients.table[store->clients.inserted].nom, nom_client);
        store->clients.inserted++;
    }
}

/*
* traite_client()
* client <Mot nom_client>
* Affiche les commandes d'un client
*/
void traite_client(Stockage* store) {
    Mot nom_client;
    get_id(&nom_client);
    if (strcmp(nom_client, "tous") == 0) {
        for (int clientsI = 0; clientsI < store->clients.inserted; ++clientsI) {
            printf(MSG_CLIENT, store->clients.table[clientsI].nom);
            int passedCheck = 0;
            for (int commandesI = 0; commandesI < store->commandes.inserted; ++commandesI) {
                if (strcmp(store->commandes.table[commandesI].nom_client, store-
>clients.table[clientsI].nom) == 0) {
                    if (passedCheck == 0)
                        printf("%s", store->commandes.table[commandesI].produit);
                    else
                        printf(", %s", store->commandes.table[commandesI].produit);
                    passedCheck++;
                }
            }
            printf("\n");
        }
    }
    else {
        printf(MSG_CLIENT, nom_client);
        int passedCheck = 0;
        for (int i = 0; i < store->commandes.inserted; ++i) {
            if (strcmp(store->commandes.table[i].nom_client, nom_client) == 0) {
                if (passedCheck == 0)
                    printf("%s", store->commandes.table[i].produit);
                else
                    printf(", %s", store->commandes.table[i].produit);
                passedCheck++;
            }
        }
        printf("\n");
    }
}

/*
* traite_commande()
* commande <Mot produit> <Mot nom_client>
* Ajoute une commande au programme
*/
void traite_commande(Stockage* store) {
    Mot produit, nom_client;
    get_id(&produit);
```

```
    get_id(&nom_client);
    const unsigned int i = store->commandes.inserted++; // stockage puis incrémentation
    strcpy(store->commandes.table[i].nom_client, nom_client);
    strcpy(store->commandes.table[i].produit, produit);
    store->commandes.table[i].complete = FAUX;
    //initialisation de la liste de tâches
    for (unsigned int speNbr = 0; speNbr < MAX_SPECIALITES; ++speNbr)
    {
        store->commandes.table[i].liste_taches[speNbr].nb_heures_requises = 0;
        store->commandes.table[i].liste_taches[speNbr].nb_heures_effectuees = 0;
    }
}

/*
* traite_supervision()
* Affiche l'avancement actuel des commandes sous le format
* OUT : "etat des taches pour <nom_produit> : [<liste>, ..., <liste>]"
*     avec <liste> valant "<specialite>:<heures effectuées>/<heures nécessaires>"
*/
void traite_supervision(Stockage* store) {
    for (int i_cmd = 0; i_cmd < store->commandes.inserted; ++i_cmd)
    {
        printf("etat des taches pour %s : ", store->commandes.table[i_cmd].produit);
        Booleen isFirstTime = VRAI;
        for (int i_spe = 0; i_spe < store->specialites.inserted; ++i_spe)
        {
            const Tache tacheCourante = store->commandes.table[i_cmd].liste_taches[i_spe];
            if (tacheCourante.nb_heures_requises) { // valeur si vide : 0 = faux
                if (isFirstTime) {
                    isFirstTime = FAUX;
                }
                else {
                    printf(", ");
                }
                printf("%s:", store->specialites.table[i_spe].nom);
                printf("%d/%d", tacheCourante.nb_heures_effectuees,
tacheCourante.nb_heures_requises);
            }
        }
        printf("\n");
    }
}

/**
 * Détermine le travailleur le plus adapté pour la réalisation d'une tâche.
 * Une tâche étant déterminée par une spécialité, il attends donc en ENTREE
 * l'identifiant d'une spécialitée attendue, et afin de faire ses opérations
 * le pointeur du tableau de travailleurs
 * En SORTIE, il retourne l'index du travailleur à prendre en charge.
**/
unsigned int determiner_travailleur_pour(const Stockage* store, int id_spe) {
    unsigned int retval = TRAVAILLEURS_SIZE;
    int totalWorker[TRAVAILLEURS_SIZE];
    for (int i = 0; i < TRAVAILLEURS_SIZE; ++i) totalWorker[i] = 0;
    for (int id_worker = 0; id_worker < store->travailleurs.inserted; ++id_worker) {
        for (int i_cmd = 0; i_cmd < store->commandes.inserted; ++i_cmd) {
            for (int i_spe = 0; i_spe < store->specialites.inserted; ++i_spe) {
                if (store->commandes.table[i_cmd].en_charge_tache[i_spe] == id_worker) {
                    totalWorker[id_worker] += store-
>commandes.table[i_cmd].liste_taches[i_spe].nb_heures_requises - store-
>commandes.table[i_cmd].liste_taches[i_spe].nb_heures_effectuees;
                }
            }
        }
    }
    int lowestHours = -1;
    for (int id_worker = 0; id_worker < store->travailleurs.inserted; ++id_worker) {
        if (store->travailleurs.table[id_worker].tag_specialite[id_spe] == VRAI) {
```

```
            if (lowestHours < 0) {
                lowestHours = totalWorker[id_worker];
                retval = id_worker;
            }
            if (EchoActif)
                printf(">>> >>> %s %d: %d (%d)\n", store->travailleurs.table[id_worker].nom,
id_worker, totalWorker[id_worker], lowestHours);
            if (lowestHours > totalWorker[id_worker]) {
                retval = id_worker;
                lowestHours = totalWorker[id_worker];
            }
        }
    }
    return retval;
}

/*
* traite_tache()
* tache <Mot commande> <Mot specialite> <int heures>
* Ajoute une specialité ainsi que le nombre d'heures requises à une commande
*/
void traite_tache(Stockage* store) {
    Mot commande, specialite;
    get_id(&commande);
    get_id(&specialite);
    int heures = get_int();
    // recherche de la commande concernée puis de la spécialité concernée
    const unsigned int cmd_i = getIndex_cmd(&store->commandes, commande);
    const unsigned int id_spe = getIndex_spe(&store->specialites, specialite);
    // initialisation de la tâche pour la commande en question
    store->commandes.table[cmd_i].liste_taches[id_spe].nb_heures_requises = heures;
    // Assignation du travailleur à la tache
    const unsigned int id_worker = determiner_travailleur_pour(id_spe, store);
    if (id_worker < TRAVAILLEURS_SIZE) {
        store->commandes.table[cmd_i].en_charge_tache[id_spe] = id_worker;
    } else if(EchoActif) {
        printf("$ Erreur : aucun travailleur trouvé pour traiter la spécialité demandée.\n");
    }
    // Il sera nécessaire de stoker le nombre de [tâches||d'heures] au bout d'une
    // affectation de tâche afin de trouver par la suite le meilleur travailleur
    // pour la prochaine tâche
    // pour la beta : set_total_tasks_for(Travailleur* travailleur);
    // pour la release : set_total_hours_for(Travailleur* travailleur);
}

/*
* traite_charge()
* charge <Mot nom_travailleur>
* A
*/
void traite_charge(Stockage* store) {
    Mot nom_travailleur;
    get_id(&nom_travailleur);
    // printf(MSG_CHARGE, nom_travailleur);
    // superProduit/reseau/45heure(s)
    unsigned int travailleursId = getIndex_trv(&store->travailleurs, nom_travailleur);
    printf("charge de travail pour %s : ", nom_travailleur);
    Booleen isFirstTime = VRAI;
    for (int commandesI = 0; commandesI < store->commandes.inserted; commandesI++) {
        for (int specialitesI = 0; specialitesI < SPECIALITE_SIZE; specialitesI++) {
            const Tache tache = store-
>commandes.table[commandesI].liste_taches[specialitesI];
            if (store->commandes.table[commandesI].en_charge_tache[specialitesI] ==
travailleursId
                && tache.nb_heures_requises != tache.nb_heures_effectuees) {
                if (isFirstTime) {
                    isFirstTime = FAUX;
                } else {
```

```
                    printf(", ");
                }
                printf("%s/%s/%dheure(s)", store->commandes.table[commandesI], store->specialites.table[specialitesI].nom, tache.nb_heures_requises - tache.nb_heures_effectuees);
            }
        }
    }
    printf("\n");
}

/*
* verif_facturation()
* Vérification si les tâches sont complétées.
*/
Booleen verif_facturation(Stockage* store, const unsigned int cmd) {
    int compteur = 0;
    for (int spec = 0; spec < SPECIALITE_SIZE; ++spec) {
        Tache tacheCourante = store->commandes.table[cmd].liste_taches[spec];
        if (tacheCourante.nb_heures_effectuees >= tacheCourante.nb_heures_requises) {
            compteur++;
        }
    }
    if (compteur >= SPECIALITE_SIZE) {
        return VRAI;
    }
    else {
        return FAUX;
    }
}

/*
* verif_facturationsGlobales()
* Vérification si absolutement toutes les commandes sont complétées.
*/
Booleen verif_facturationsGlobales(Stockage* store) {
    int compteurGlobal = 0;
    for (int cmd = 0; cmd < store->commandes.inserted; ++cmd) {
        if (store->commandes.table[cmd].complete == VRAI) {
            compteurGlobal++;
        }
    }
    if (compteurGlobal == store->commandes.inserted) {
        return VRAI;
    }
    else {
        return FAUX;
    }
}

/*
* traite_progression()
* tache <Mot produit> <Mot specialite> <int heures_travaillees>
* Enregistre une progression aux seins d'une specialité d'une commande
*/
Booleen traite_progression(Stockage* store) {
    Mot commande, specialite;
    get_id(&commande);
    get_id(&specialite);
    int heures_travaillees = get_int();
    const unsigned int cmd_i = getIndex_cmd(&store->commandes, commande);
    const unsigned int id_spe = getIndex_spe(&store->specialites, specialite);
    store->commandes.table[cmd_i].liste_taches[id_spe].nb_heures_effectuees += heures_travaillees;
    Booleen checkFacturation = verif_facturation(store, cmd_i);
    if (checkFacturation && store->commandes.table[cmd_i].complete == FAUX) {
        printf("facturation %s : ", commande);
        Booleen firstDone = VRAI;
        for (int specCommande = 0; specCommande < SPECIALITE_SIZE; ++specCommande) {
```

```
            if (store->commandes.table[cmd_i].liste_taches[specCommande].nb_heures_requises !
= 0) {
                if (firstDone) {
                    firstDone = FAUX;
                }
                else {
                    printf(", ");
                }
                Gros_entier cout_commande = store-
>commandes.table[cmd_i].liste_taches[specCommande].nb_heures_effectuees * store-
>specialites.table[specCommande].cout_horaire;
                printf("%s:%llu", store->specialites.table[specCommande].nom, cout_commande);
                store->commandes.table[cmd_i].complete = VRAI;
            }
        }
        printf("\n");
    }
    Booleen checkFacturationsGlobales = verif_facturationsGlobales(store);
    if (checkFacturationsGlobales) {
        printf("facturations : ");
        Booleen firstDone = VRAI;
        for (int client = 0; client < store->clients.inserted; ++client) {
            Gros_entier cout_total_client = 0;
            for (int commande = 0; commande < store->commandes.inserted; ++commande) {
                if(strcmp(store->commandes.table[commande].nom_client, store-
>clients.table[client].nom) == 0) {
                    for (int spec = 0; spec < SPECIALITE_SIZE; ++spec) {
                        Tache tacheCourante = store-
>commandes.table[commande].liste_taches[spec];
                        cout_total_client += tacheCourante.nb_heures_effectuees * store-
>specialites.table[spec].cout_horaire;
                    }
                }
            }
            if (firstDone) {
                firstDone = FAUX;
            }
            else {
                printf(", ");
            }
            printf("%s:%llu", store->clients.table[client].nom, cout_total_client);
        }
        printf("\n");
        return FAUX;
    }
    return VRAI;
}

/*
* traite_passe()
* passe
* Traitement des réallocations des dernières progressions
*/
void traite_passe() {
    //printf(MSG_PASSE);
}

/*
* traite_interruption()
* interruption
* Interromp le programme
*/
void traite_interruption() {
    //printf(MSG_INTERRUPTION);
}
```

```c
// Porte d'entrée du programme

int main(int argc, char* argv[]) {
    if (argc >= 2 && strcmp("echo", argv[1]) == 0) {
        EchoActif = VRAI;
    }
    setlocale(LC_ALL, "fr-FR");
    Stockage globalStore;
    globalStore.specialites.inserted = 0;
    globalStore.travailleurs.inserted = 0;
    globalStore.clients.inserted = 0;
    globalStore.commandes.inserted = 0;
    Mot buffer;
    while (VRAI) {
        get_id(&buffer);
        if (strcmp(buffer, "developpe") == 0) {
            traite_developpe(&globalStore);
            continue;
        }
        if (strcmp(buffer, "specialites") == 0) {
            traite_specialites(&globalStore);
            continue;
        }
        if (strcmp(buffer, "embauche") == 0) {
            traite_embauche(&globalStore);
            continue;
        }
        if (strcmp(buffer, "travailleurs") == 0) {
            traite_travailleurs(&globalStore);
            continue;
        }
        if (strcmp(buffer, "demarche") == 0) {
            traite_demarche(&globalStore);
            continue;
        }
        if (strcmp(buffer, "client") == 0) {
            traite_client(&globalStore);
            continue;
        }
        if (strcmp(buffer, "commande") == 0) {
            traite_commande(&globalStore);
            continue;
        }
        if (strcmp(buffer, "supervision") == 0) {
            traite_supervision(&globalStore);
            continue;
        }
        if (strcmp(buffer, "tache") == 0) {
            traite_tache(&globalStore);
            continue;
        }
        if (strcmp(buffer, "charge") == 0) {
            traite_charge(&globalStore);
            continue;
        }
        if (strcmp(buffer, "progression") == 0) {
            if (traite_progression(&globalStore)) {
                continue;
            } else {
                break;
            }
        }
        if (strcmp(buffer, "passe") == 0) {
            traite_passe();
            continue;
        }
        if (strcmp(buffer, "interruption") == 0) {
            traite_interruption();
```

```
            break;
        }
        printf("!!! instruction inconnue >%s< !!!\n", buffer);
    }
    return 0;
}
```

```c
/*
   sprint6-r (soutenance)
*/

#pragma warning(disable:4996)
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <locale.h>

// Messages emis par les instructions

#define MSG_DEVELOPPE "## nouvelle specialite \"%s\" ; cout horaire \"%d\"\n"
#define MSG_SPECIALITES "specialites traitees : "
#define MSG_INTERRUPTION "## fin de programme\n"
#define MSG_TRAVAILLEURS "la specialite %s peut etre prise en charge par : "
#define MSG_CLIENT "le client %s a commande : "
#define MSG_COMMANDE "## nouvelle commande \"%s\", par client \"%s\"\n"
#define MSG_SUPERVISION "## consultation de l'avancement des commandes\n"
#define MSG_TACHE "## la commande \"%s\" requiere la specialite \"%s\" (nombre
d'heures \"%d\")\n"
#define MSG_CHARGE   "charge de travail pour %s : "
#define MSG_PROGRESSION "## pour la commande \"%s\", pour la specialite \"%s\" : \"%d\"
heures de plus ont ete realisees\n"
#define MSG_PASSE "## une reallocation est requise\n"

// Lexemes

#define LGMOT 35
#define NBCHIFFREMAX 5
#define MAX_COMMANDES 500
#define MAX_SPECIALITES 10
typedef char Mot[LGMOT + 1]; // Définition du type Mot
typedef unsigned long long Gros_entier;

typedef enum { FAUX = 0, VRAI = 1 } Booleen;
Booleen EchoActif = FAUX;

void get_id(Mot* id) {
    scanf("%s", id);
    if (EchoActif) printf(">>echo %s\n", id);
}

int get_int() {
    char buffer[NBCHIFFREMAX + 1];
    scanf("%s", buffer);
    if (EchoActif) printf(">>echo %s\n", buffer);
    return atoi(buffer);
}

// Déclarations des constantes

const enum {
    SPECIALITE_SIZE = 10,
    TRAVAILLEURS_SIZE = 50,
    CLIENTS_SIZE = 100,
    COMMANDES_SIZE = 500
};
```

```c
// Specialité

typedef struct {
    Mot nom;
    int cout_horaire;
} Specialite;
typedef struct {
    Specialite table[SPECIALITE_SIZE];
    int inserted;
} Specialites;

// Travailleurs

typedef struct {
    Mot nom;
    Booleen tag_specialite[SPECIALITE_SIZE];
    int heuresRealises;
} Travailleur;
typedef struct {
    Travailleur table[TRAVAILLEURS_SIZE];
    int inserted;
} Travailleurs;

// Client

typedef struct {
    Mot nom;
} Client;
typedef struct {
    Client table[CLIENTS_SIZE];
    int inserted;
} Clients;

// Commande et taches

typedef  struct {
    unsigned int nb_heures_requises;
    unsigned int nb_heures_effectuees;
} Tache;
typedef struct {
    Mot produit;
    Mot nom_client;
    Tache liste_taches[SPECIALITE_SIZE];
    Booleen complete;
    // L'identifiant du travailleur en charge de la tache
    unsigned int en_charge_tache[SPECIALITE_SIZE];
} Commande;
typedef struct {
    Commande table[COMMANDES_SIZE];
    int inserted;
} Commandes;

// Stockage global

typedef struct {
    Specialites specialites;
    Travailleurs travailleurs;
    Clients clients;
    Commandes commandes;
    // pour la commande progression passe
    int lastCommande;
    int lastSpecialite;
    int lastProgression;
} Stockage;
```

```c
// Declaration des prototypes de fonction
//-------------------------------------------------//
// fonctions helpers
/**
* [brief] Permet de récupérer une entrée écrite par l'utilisateur.
* id [out] Pointeur de type Mot qui permettra la modification de la variable originale.
* [pre] (l'entrée de l'utilisateur doit être valide)
**/
void get_id(Mot* id);

/**
* [brief] A le même effet que get_id, mais retourne l'entrée en tant que int plutôt que
affecter dans l'argument via pointeur.
* [pre] (l'entrée de l'utilisateur doit être un entier naturel)
* return: l'entier qui a été entré par l'utilisateur.
**/
int get_int();

/**
* [brief] Donné une structure Commandes (in 1) et le nom d'un produit (in 2), retourne
l'indice du produit concerné dans le tableau de la structure Commandes.
*  ie. trouve et retourne l'entier n tel que Commandes->table[n].produit est égale
nom_produit
* commandes [in] : pointeur constant d'une struct de type Commandes. Spécifie la struct dans
laquelle on va chercher.
* nom_produit [in] : de type Mot. Spécifie le nom du produit à rechercher dans la struct
donnée en tant que premier argument.
* return: l'indice du produit concerné dans le tableau de la structure Commandes.
**/
int getIndex_cmd(const Commandes* commandes, Mot nom_produit);

/**
* [brief] Donné une structure Travailleurs (in 1) et le nom d'un travailleur (in 2), retourne
l'indice du travailleur concerné dans le tableau de la structure Travailleurs.
*  ie. trouve et retourne l'entier n tel que Travailleurs->table[n].nom est égale nom
* travailleurs [in] : pointeur constant d'une struct de type Travailleurs. Spécifie la struct
dans laquelle on va chercher.
* nom [in] : de type Mot. Spécifie le nom du travailleur à rechercher dans la struct donnée
en tant que premier argument.
* return: l'indice du travailleur concerné dans le tableau de la structure Travailleurs.
**/
int getIndex_trv(const Travailleurs* travailleurs, Mot nom);

/**
* [brief] Donné une structure Specialites (in 1) et le nom d'une spécialité (in 2), retourne
l'indice de la spécialité concerné dans le tableau de la structure Specialites.
*   ie. trouve et retourne l'entier n tel que Specialites->table[n].nom est égale nom
* specialites [in] : pointeur constant d'une struct de type Specialites. Spécifie la struct
dans laquelle on va chercher.
* nom [in] : de type Mot. Spécifie le nom de la spécialité à rechercher dans la struct donnée
en tant que premier argument.
* return: l'indice de la spécialité concerné dans le tableau de la structure Specialites.
**/
int getIndex_spe(const Specialites* specialites, Mot nom);

/**
* [brief] Une spécialité sera créée à l'aide de l'entrée utilisateur demandée.
* store [in-out] : pointeur constant d'une struct de type Stockage. Spécifie la struct dans
laquelle on va chercher et éditer.
**/
void traite_developpe(Stockage* store);
```

```c
/**
* [brief] La liste des spécialités sera affichée à l'utilisateur.
* store [in] : pointeur constant d'une struct de type Stockage. Spécifie la struct dans
laquelle on va chercher.
**/
void traite_specialites(const Stockage* store);

/**
* [brief] Un nouveau travailleur sera crée à l'aide de l'entrée utilisateur.
* store [in-out] : pointeur constant d'une struct de type Stockage. Spécifie la struct dans
laquelle on va chercher et éditer.
**/
void traite_embauche(Stockage* store);

/**
* [brief] La liste des travailleurs sera affichée à l'utilisateur selon la spécialité
mentionnée ou "tous".
* store [in] : pointeur constant d'une struct de type Stockage. Spécifie la struct dans
laquelle on va chercher.
**/
void traite_travailleurs(const Stockage* store);

/**
* [brief] Un nouveau client sera crée à l'aide de l'entrée utilisateur.
* store [in-out] : pointeur constant d'une struct de type Stockage. Spécifie la struct dans
laquelle on va chercher et éditer.
**/
void traite_demarche(Stockage* store);

/**
* [brief] La liste des commandes d'un client donnée par l'utilisateur sera affichée à
l'utilisateur.
*       L'utilisateur peut aussi mentionner "tous" si il veut voir toutes les commandes de
chaque client.
* store [in] : pointeur constant d'une struct de type Stockage. Spécifie la struct dans
laquelle on va chercher.
**/
void traite_client(const Stockage* store);

/**
* [brief] Une nouvelle commande sera créée à l'aide de l'entrée utilisateur.
* store [in-out] : pointeur constant d'une struct de type Stockage. Spécifie la struct dans
laquelle on va chercher et éditer.
**/
void traite_commande(Stockage* store);

/**
* [brief] Affiche l'avancement actuel des commandes.
* store [in] : pointeur constant d'une struct de type Stockage. Spécifie la struct dans
laquelle on va chercher.
**/
void traite_supervision(const Stockage* store);

/**
* [brief] Determine le travailleur le plus adéquat pour une spécialité et le renvoie sous
forme de char car le MAX_TRAVAILLEURS < CHAR_MAX.
* store [in] : pointeur constant d'une struct de type Stockage. Spécifie la struct dans
laquelle on va chercher.
* id_spe [in] : indice de la spécialité.
* return: indice du travailleur determiné.
**/
char determiner_travailleur_pour(const Stockage* store, int id_spe);
```

```c
/**
* [brief] Ajoute une specialité ainsi que le nombre d'heures requises à une commande
* store [in-out] : pointeur constant d'une struct de type Stockage. Spécifie la struct dans
laquelle on va chercher et éditer.
**/
void traite_tache(Stockage* store);

/**
* [brief] Affiche la charge de travail d'un travailleur.
* store [in] : pointeur constant d'une struct de type Stockage. Spécifie la struct dans
laquelle on va chercher.
**/
void traite_charge(const Stockage* store);

/**
* [brief] Vérifie si toutes les tâches de la commande spécifiée dans l'argument cmd sont
complétées.
* store [in] : pointeur constant d'une struct de type Stockage. Spécifie la struct dans
laquelle on va chercher.
* cmd [in] : indice de la commande
* return: booleen VRAI ou FAUX indiquant si les tâches de la commande spécifiée sont bien
completées.
**/
Booleen verif_facturation(const Stockage* store, unsigned int cmd);

/**
* [brief] Vérifie si absolutement toutes les commandes sont complétées.
* store [in] : pointeur constant d'une struct de type Stockage. Spécifie la struct dans
laquelle on va chercher.
* return: booleen VRAI ou FAUX indiquant si toutes les tâches de toutes les commande sont
bien completées.
**/
Booleen verif_facturationsGlobales(const Stockage* store);

/**
* [brief] Indique une progression pour une tâche donnée par l'entrée utilisateur.
* store [in] : pointeur constant d'une struct de type Stockage. Spécifie la struct dans
laquelle on va chercher et éditer.
* return: booleen VRAI ou FAUX indiquant si toutes les tâches de toutes les commande sont
bien completées en utilisant les fonctions de vérifications au dessus.
**/
Booleen traite_progression(Stockage* store);

/**
* [brief] Réaffecte un nouveau travailleur pour la dernière tâche passée dans l'instruction
progression.
* store [in] : pointeur constant d'une struct de type Stockage. Spécifie la struct dans
laquelle on va chercher et éditer.
**/
void traite_passe(Stockage* store);
// Porte d'entrée du programme
int main(int argc, char* argv[]) {
    if (argc >= 2 && strcmp("echo", argv[1]) == 0) {
        EchoActif = VRAI;
    }
    setlocale(LC_ALL, "fr-FR");
    Stockage globalStore;
    globalStore.specialites.inserted = 0;
    globalStore.travailleurs.inserted = 0;
    globalStore.clients.inserted = 0;
    globalStore.commandes.inserted = 0;
    globalStore.lastCommande = -1;
    globalStore.lastSpecialite = -1;
    globalStore.lastProgression = -1;
    Mot buffer;
    while (VRAI) {
        get_id(&buffer);
        if (strcmp(buffer, "developpe") == 0) {
```

```
            traite_developpe(&globalStore);
            continue;
        }
        if (strcmp(buffer, "specialites") == 0) {
            traite_specialites(&globalStore);
            continue;
        }
        if (strcmp(buffer, "embauche") == 0) {
            traite_embauche(&globalStore);
            continue;
        }
        if (strcmp(buffer, "travailleurs") == 0) {
            traite_travailleurs(&globalStore);
            continue;
        }
        if (strcmp(buffer, "demarche") == 0) {
            traite_demarche(&globalStore);
            continue;
        }
        if (strcmp(buffer, "client") == 0) {
            traite_client(&globalStore);
            continue;
        }
        if (strcmp(buffer, "commande") == 0) {
            traite_commande(&globalStore);
            continue;
        }
        if (strcmp(buffer, "supervision") == 0) {
            traite_supervision(&globalStore);
            continue;
        }
        if (strcmp(buffer, "tache") == 0) {
            traite_tache(&globalStore);
            continue;
        }
        if (strcmp(buffer, "charge") == 0) {
            traite_charge(&globalStore);
            continue;
        }
        if (strcmp(buffer, "progression") == 0) {
            if (traite_progression(&globalStore)) {
                continue;
            }
            else {
                break;
            }
        }
        if (strcmp(buffer, "passe") == 0) {
            traite_passe(&globalStore);
            continue;
        }
        if (strcmp(buffer, "interruption") == 0) {
            break;
        }
        printf("!!! instruction inconnue >%s< !!!\n", buffer);
    }
    return 0;
}
```

```c
// Helpers

int getIndex_cmd(const Commandes* commandes, Mot nom_produit) {
    // dans le cas où commandes->inserted vaut 0, la boucle n'est pas executée.
    // Il ne s'agit donc pas d'une précondition
    for (unsigned int i = 0; i < commandes->inserted; i++) {
        if (strcmp(commandes->table[i].produit, nom_produit) == 0) {
            return i;
        }
    }
    return -1; //fallback
}

int getIndex_trv(const Travailleurs* travailleurs, Mot nom) {
    for (unsigned int i = 0; i < travailleurs->inserted; i++) {
        if (strcmp(travailleurs->table[i].nom, nom) == 0) {
            return i;
        }
    }
    return -1; //fallback
}

int getIndex_spe(const Specialites* specialites, Mot nom) {
    for (unsigned int i = 0; i < specialites->inserted; i++) {
        if (strcmp(specialites->table[i].nom, nom) == 0) {
            return i;
        }
    }
    return -1; //fallback
}

// Commandes

/*
* traite_developpe() || CF. prototype pour la doc de cette fonction
*
* Entrée utilisateur attendue : developpe <Mot nom_specialite> <int cout_horaire>
* où nom_specialite est le nom de la spécialité à créer
* et cout_horaire le prix de facturation d'une heure de travail pour cette spé
*/
void traite_developpe(Stockage* store) {
    Mot nom_specialite;
    get_id(&nom_specialite);
    int cout_horaire = get_int();
    if (store->specialites.inserted < SPECIALITE_SIZE) {
        strcpy(store->specialites.table[store->specialites.inserted].nom, nom_specialite);
        store->specialites.table[store->specialites.inserted].cout_horaire = cout_horaire;
        store->specialites.inserted++;
    }
}

/*
* traite_specialites()|| CF. prototype pour la doc de cette fonction
*
* Entrée utilisateur attendue : specialites
* Cette instruction affiche les specialités enregistrées dans le programme dans l'ordre de
leur déclaration, ainsi que le coût associé
*/
void traite_specialites(Stockage* store) {
    printf(MSG_SPECIALITES);
    for (int i = 0; i < store->specialites.inserted; ++i) {
        if (i == 0)
            printf("%s/%d", store->specialites.table[i].nom, store-
>specialites.table[i].cout_horaire);
        else
            printf(", %s/%d", store->specialites.table[i].nom, store-
>specialites.table[i].cout_horaire);
    }
```

```
    printf("\n");
}

/*
* traite_embauche()|| CF. prototype pour la doc de cette fonction
*
* Entrée utilisateur attendue : embauche <Mot travailleur> <Mot specialite>
* Recherche le travailleur qui a pour nom le premier argument (ci celui-ci n'existe pas,
alors il sera créé),
* et lui assigne la spécialité donnée en tant que second argument
*/
void traite_embauche(Stockage* store) {
    Mot travailleur, specialite;
    get_id(&travailleur);
    get_id(&specialite);
    if (store->travailleurs.inserted < TRAVAILLEURS_SIZE) {
        int travailleurExistant = getIndex_trv(&store->travailleurs, travailleur);
        if (travailleurExistant >= 0) {
            int indexSpe = getIndex_spe(&store->specialites, specialite);
            store->travailleurs.table[travailleurExistant].tag_specialite[indexSpe] = VRAI;
        }
        else {
            strcpy(store->travailleurs.table[store->travailleurs.inserted].nom, travailleur);
            int indexSpe = getIndex_spe(&store->specialites, specialite);
            for (unsigned int i = 0; i < SPECIALITE_SIZE; ++i) {
                store->travailleurs.table[store->travailleurs.inserted]
                        .tag_specialite[i] = (i == indexSpe) ? VRAI : FAUX;
                // Necessité d'initiliaser chaque cases pour éviter des bugs.
            }
            store->travailleurs.table[store->travailleurs.inserted].heuresRealises = 0;
            store->travailleurs.inserted++;
        }
    }
}

/*
* traite_travailleurs() || CF. prototype pour la doc de cette fonction
*
* Entrée utilisateur attendue : travailleurs <Mot specialite || "tous">
* Affiche la liste des travailleurs compétents pour la spécialité donnée en argument,par
ordre de déclaration des travailleurs.
* Si le nom de la spécialité vaut "tous", affiche la liste des travailleurs pour toutes les
spécialités.
*/
void traite_travailleurs(const Stockage* store) {
    Mot specialite;
    get_id(&specialite);
    if (strcmp(specialite, "tous") == 0) {
        for (int specialitesI = 0; specialitesI < store->specialites.inserted; +
+specialitesI) {
            printf(MSG_TRAVAILLEURS, store->specialites.table[specialitesI].nom);
            int passedCheck = 0;
            for (int travailleursI = 0; travailleursI < store->travailleurs.inserted; +
+travailleursI) {
                if (store->travailleurs.table[travailleursI].tag_specialite[specialitesI] ==
VRAI) {
                    if (passedCheck == 0)
                        printf("%s", store->travailleurs.table[travailleursI].nom);
                    else
                        printf(", %s", store->travailleurs.table[travailleursI].nom);
                    passedCheck++;
                }
            }
            printf("\n");
        }
    }
    else {
        printf(MSG_TRAVAILLEURS, specialite);
```

```
        int indexSpe = getIndex_spe(&store->specialites, specialite);
        int passedCheck = 0;
        for (int travailleursI = 0; travailleursI < store->travailleurs.inserted; +
+travailleursI) {
            if (store->travailleurs.table[travailleursI].tag_specialite[indexSpe] == VRAI) {
                if (passedCheck == 0)
                    printf("%s", store->travailleurs.table[travailleursI].nom);
                else
                    printf(", %s", store->travailleurs.table[travailleursI].nom);
                passedCheck++;
            }
        }
        printf("\n");
    }
}

/*
* traite_demarche() || CF. prototype pour la doc de cette fonction
*
* demarche <Mot nom_client>
* Déclare un nouveau client qui aura pour nom le premier argument
*/
void traite_demarche(Stockage* store) {
    Mot nom_client;
    get_id(&nom_client);
    if (store->clients.inserted < CLIENTS_SIZE) {
        strcpy(store->clients.table[store->clients.inserted].nom, nom_client);
        store->clients.inserted++;
    }
}

/*
* traite_client() || CF. prototype pour la doc de cette fonction
*
* client <Mot nom_client>
* Affiche les commandes du client ayant pour nom le premier argument
*/
void traite_client(const Stockage* store) {
    Mot nom_client;
    get_id(&nom_client);
    if (strcmp(nom_client, "tous") == 0) {
        for (int clientsI = 0; clientsI < store->clients.inserted; ++clientsI) {
            printf(MSG_CLIENT, store->clients.table[clientsI].nom);
            int passedCheck = 0;
            for (int commandesI = 0; commandesI < store->commandes.inserted; ++commandesI) {
                if (strcmp(store->commandes.table[commandesI].nom_client, store-
>clients.table[clientsI].nom) == 0) {
                    if (passedCheck == 0)
                        printf("%s", store->commandes.table[commandesI].produit);
                    else
                        printf(", %s", store->commandes.table[commandesI].produit);
                    passedCheck++;
                }
            }
            printf("\n");
        }
    }
    else {
        printf(MSG_CLIENT, nom_client);
        int passedCheck = 0;
        for (int i = 0; i < store->commandes.inserted; ++i) {
            if (strcmp(store->commandes.table[i].nom_client, nom_client) == 0) {
                if (passedCheck == 0)
                    printf("%s", store->commandes.table[i].produit);
                else
                    printf(", %s", store->commandes.table[i].produit);
                passedCheck++;
            }
```

```
        }
        printf("\n");
    }
}

/*
* traite_commande() || CF. prototype pour la doc de cette fonction
*
* commande <Mot produit> <Mot nom_client>
* Ajoute une commande au programme, qui aura pour nom le premier argument.
* Le second spécifie le nom du client qui l'a commandé
*/
void traite_commande(Stockage* store) {
    Mot produit, nom_client;
    get_id(&produit);
    get_id(&nom_client);
    const unsigned int i = store->commandes.inserted++; // stockage puis incrémentation
    strcpy(store->commandes.table[i].nom_client, nom_client);
    strcpy(store->commandes.table[i].produit, produit);
    store->commandes.table[i].complete = FAUX;
    //initialisation de la liste de tâches
    for (unsigned int speNbr = 0; speNbr < MAX_SPECIALITES; ++speNbr)
    {
        store->commandes.table[i].liste_taches[speNbr].nb_heures_requises = 0;
        store->commandes.table[i].liste_taches[speNbr].nb_heures_effectuees = 0;
    }
}

/*
* traite_supervision() || CF. prototype pour la doc de cette fonction
*
* Affiche l'avancement actuel des commandes sous le format suivant:
*  "etat des taches pour <nom_produit> : [<liste>, ..., <liste>]"
*     avec <liste> valant "<specialite>:<heures effectuées>/<heures nécessaires>"
*/
void traite_supervision(const Stockage* store) {
    for (int i_cmd = 0; i_cmd < store->commandes.inserted; ++i_cmd)
    {
        printf("etat des taches pour %s : ", store->commandes.table[i_cmd].produit);
        Booleen isFirstTime = VRAI;
        for (int i_spe = 0; i_spe < store->specialites.inserted; ++i_spe)
        {
            const Tache tacheCourante = store->commandes.table[i_cmd].liste_taches[i_spe];
            if (tacheCourante.nb_heures_requises) { // valeur si vide : 0 = faux
                if (isFirstTime) {
                    isFirstTime = FAUX;
                }
                else {
                    printf(", ");
                }
                printf("%s:", store->specialites.table[i_spe].nom);
                printf("%d/%d", tacheCourante.nb_heures_effectuees,
tacheCourante.nb_heures_requises);
            }
        }
        printf("\n");
    }
}
```

```
/**
 * Détermine le travailleur le plus adapté pour la réalisation d'une tâche.
 * Une tâche étant déterminée par une spécialité, il attends donc en ENTREE
 * l'identifiant d'une spécialitée attendue, et afin de faire ses opérations
 * le pointeur du tableau de travailleurs
 * En SORTIE, il retourne l'index du travailleur à prendre en charge.
**/
char determiner_travailleur_pour(const Stockage* store, int id_spe) {
    unsigned int retval = TRAVAILLEURS_SIZE;
    int totalWorker[TRAVAILLEURS_SIZE];
    for (int i = 0; i < TRAVAILLEURS_SIZE; ++i) totalWorker[i] = 0;
    for (int id_worker = 0; id_worker < store->travailleurs.inserted; ++id_worker) {
        for (int i_cmd = 0; i_cmd < store->commandes.inserted; ++i_cmd) {
            for (int i_spe = 0; i_spe < store->specialites.inserted; ++i_spe) {
                if (store->commandes.table[i_cmd].en_charge_tache[i_spe] == id_worker) {
                    totalWorker[id_worker] += store-
>commandes.table[i_cmd].liste_taches[i_spe].nb_heures_requises - store-
>commandes.table[i_cmd].liste_taches[i_spe].nb_heures_effectuees;
                }
            }
        }
    }
    int lowestHours = -1;
    for (int id_worker = 0; id_worker < store->travailleurs.inserted; ++id_worker) {
        if (store->travailleurs.table[id_worker].tag_specialite[id_spe] == VRAI) {
            if (lowestHours < 0) {
                lowestHours = totalWorker[id_worker];
                retval = id_worker;
            }
            if (EchoActif)
                printf(">>> >>> %s %d: %d (%d)\n", store->travailleurs.table[id_worker].nom,
id_worker, totalWorker[id_worker], lowestHours);
            if (lowestHours > totalWorker[id_worker]) {
                retval = id_worker;
                lowestHours = totalWorker[id_worker];
            }
        }
    }
    return retval;
}
/*
* traite_tache() || CF. prototype pour la doc de cette fonction
*
* tache <Mot commande> <Mot specialite> <int heures>
* Ajoute une specialité ainsi que le nombre d'heures requises à une commande
*/
void traite_tache(Stockage* store) {
    Mot commande, specialite;
    get_id(&commande);
    get_id(&specialite);
    int heures = get_int();
    // recherche de la commande concernée puis de la spécialité concernée
    const unsigned int cmd_i = getIndex_cmd(&store->commandes, commande);
    const unsigned int id_spe = getIndex_spe(&store->specialites, specialite);
    // initialisation de la tâche pour la commande en question
    store->commandes.table[cmd_i].liste_taches[id_spe].nb_heures_requises = heures;
    // Assignation du travailleur à la tache
    const unsigned int id_worker = determiner_travailleur_pour(store, id_spe);
    if (id_worker < TRAVAILLEURS_SIZE) {
        store->commandes.table[cmd_i].en_charge_tache[id_spe] = id_worker;
        // store->travailleurs.table[id_worker].heuresRealises += heures;
    }
    else if (EchoActif) {
        printf("$ Erreur : aucun travailleur trouvé pour traiter la spécialité demandée.\n");
    }
    // Il sera nécessaire de stoker le nombre de [tâches||d'heures] au bout d'une
    // affectation de tâche afin de trouver par la suite le meilleur travailleur
    // pour la prochaine tâche
```

```
    // pour la beta : set_total_tasks_for(Travailleur* travailleur);
    // pour la release : set_total_hours_for(Travailleur* travailleur);
}

/*
* traite_charge() || CF. prototype pour la doc de cette fonction
*
* charge <Mot nom_travailleur>
*/
void traite_charge(const Stockage* store) {
    Mot nom_travailleur;
    get_id(&nom_travailleur);
    // printf(MSG_CHARGE, nom_travailleur);
    // superProduit/reseau/45heure(s)
    unsigned int travailleursId = getIndex_trv(&store->travailleurs, nom_travailleur);
    printf("charge de travail pour %s : ", nom_travailleur);
    Booleen isFirstTime = VRAI;
    for (int commandesI = 0; commandesI < store->commandes.inserted; commandesI++) {
        for (int specialitesI = 0; specialitesI < SPECIALITE_SIZE; specialitesI++) {
            const Tache tache = store-
>commandes.table[commandesI].liste_taches[specialitesI];
            if (store->commandes.table[commandesI].en_charge_tache[specialitesI] ==
travailleursId
                && tache.nb_heures_requises != tache.nb_heures_effectuees) {
                if (isFirstTime) {
                    isFirstTime = FAUX;
                }
                else {
                    printf(", ");
                }
                printf("%s/%s/%dheure(s)", store->commandes.table[commandesI].produit, store-
>specialites.table[specialitesI].nom, tache.nb_heures_requises - tache.nb_heures_effectuees);
            }
        }
    }
    printf("\n");
}

/*
* verif_facturation() || CF. prototype pour la doc de cette fonction
*/
Booleen verif_facturation(Stockage* store, const unsigned int cmd) {
    int compteur = 0;
    for (int spec = 0; spec < SPECIALITE_SIZE; ++spec) {
        Tache tacheCourante = store->commandes.table[cmd].liste_taches[spec];
        if (tacheCourante.nb_heures_effectuees >= tacheCourante.nb_heures_requises) {
            compteur++;
        }
    }
    if (compteur >= SPECIALITE_SIZE) {
        return VRAI;
    }
    else {
        return FAUX;
    }
}

/*
* verif_facturationsGlobales() || CF. prototype pour la doc de cette fonction
*/
Booleen verif_facturationsGlobales(const Stockage* store) {
    int compteurGlobal = 0;
    for (int cmd = 0; cmd < store->commandes.inserted; ++cmd) {
        if (store->commandes.table[cmd].complete == VRAI) {
            compteurGlobal++;
        }
    }
    if (compteurGlobal == store->commandes.inserted) {
```

```
        return VRAI;
    }
    else {
        return FAUX;
    }
}

/*
* traite_progression() || CF. prototype pour la doc de cette fonction
*
* entrée utilisateur attendue tache : <Mot produit> <Mot specialite> <int heures_travaillees>
* Donné le nom d'une commande, et le nom de la spécialité à mettre à jour, modifie son nombre
d'heures travaillées
*/
Booleen traite_progression(Stockage* store) {
    Mot commande, specialite;
    get_id(&commande);
    get_id(&specialite);
    int heures_travaillees = get_int();
    const unsigned int cmd_i = getIndex_cmd(&store->commandes, commande);
    const unsigned int id_spe = getIndex_spe(&store->specialites, specialite);
    store->commandes.table[cmd_i].liste_taches[id_spe].nb_heures_effectuees +=
heures_travaillees;
    store->lastCommande = cmd_i;
    store->lastSpecialite = id_spe;
    store->lastProgression = heures_travaillees;
    Booleen checkFacturation = verif_facturation(store, cmd_i);
    if (checkFacturation && store->commandes.table[cmd_i].complete == FAUX) {
        printf("facturation %s : ", commande);
        Booleen firstDone = VRAI;
        for (int specCommande = 0; specCommande < SPECIALITE_SIZE; ++specCommande) {
            if (store->commandes.table[cmd_i].liste_taches[specCommande].nb_heures_requises !
= 0) {
                if (firstDone) {
                    firstDone = FAUX;
                }
                else {
                    printf(", ");
                }
                Gros_entier cout_commande = store-
>commandes.table[cmd_i].liste_taches[specCommande].nb_heures_effectuees * store-
>specialites.table[specCommande].cout_horaire;
                printf("%s:%llu", store->specialites.table[specCommande].nom, cout_commande);
                store->commandes.table[cmd_i].complete = VRAI;
            }
        }
        printf("\n");
    }
    Booleen checkFacturationsGlobales = verif_facturationsGlobales(store);
    if (checkFacturationsGlobales) {
        printf("facturations : ");
        Booleen firstDone = VRAI;
        for (int client = 0; client < store->clients.inserted; ++client) {
            Gros_entier cout_total_client = 0;
            for (int commande = 0; commande < store->commandes.inserted; ++commande) {
                if (strcmp(store->commandes.table[commande].nom_client, store-
>clients.table[client].nom) == 0) {
                    for (int spec = 0; spec < SPECIALITE_SIZE; ++spec) {
                        Tache tacheCourante = store-
>commandes.table[commande].liste_taches[spec];
                        cout_total_client += tacheCourante.nb_heures_effectuees * store-
>specialites.table[spec].cout_horaire;
                    }
                }
            }
            if (firstDone) {
                firstDone = FAUX;
            }
```

```
            else {
                printf(", ");
            }
            printf("%s:%llu", store->clients.table[client].nom, cout_total_client);
        }
        printf("\n");
        return FAUX;
    }
    return VRAI;
}

/*
* traite_passe() || CF. prototype pour la doc de cette fonction
*
* entrée utilisateur attendue : passe
* réaffecte un nouveau travailleur pour la dernière tâche passée en argument de l'instruction
progression
*/
void traite_passe(Stockage* store) {
    if (store->lastCommande && store->lastSpecialite) {
        if (EchoActif)
            printf(">>> commande: %d / specialite: %d / worker actuel: %d \n", store-
>lastCommande, store->lastSpecialite, store->commandes.table[store-
>lastCommande].en_charge_tache[store->lastSpecialite]);
        const unsigned int id_worker = determiner_travailleur_pour(store, store-
>lastSpecialite);
        store->commandes.table[store->lastCommande].en_charge_tache[store->lastSpecialite] =
id_worker;
        if (EchoActif)
            printf(">>> nouveau worker: %d - %d\n", id_worker, store->commandes.table[store-
>lastCommande].en_charge_tache[store->lastSpecialite]);
        store->lastCommande = 0;
        store->lastSpecialite = 0;
    }
}
```

```c
/*
    sprint.c
    ------------------------------------------------------------ 80 chars ->
    @author JiveOff (Antoine Banha - antoine@jiveoff.fr)
    @author ShinProg (Logan Tann - logan@kagescan.fr)
*/

#pragma warning(disable:4996)
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <locale.h>
#include <assert.h>

// Messages emis par les instructions

#define MSG_SPECIALITES "specialites traitees : "
#define MSG_TRAVAILLEURS "la specialite %s peut etre prise en charge par : "
#define MSG_CLIENT "le client %s a commande : "
#define MSG_CHARGE  "charge de travail pour %s : "
#define MSG_SUPERVISION "etat des taches pour %s : "
#define MSG_FACTURATION "facturation %s : "
#define MSG_FACTURATION_SEULE "facturations : "

// Lexemes

#define LGMOT 35
#define NBCHIFFREMAX 5
#define MAX_COMMANDES 500
#define MAX_SPECIALITES 10
typedef char Mot[LGMOT + 1]; // Définition du type Mot
typedef unsigned long long Gros_entier;

typedef enum { FAUX = 0, VRAI = 1 } Booleen;
Booleen EchoActif = FAUX;

void get_id(Mot* id) {
    scanf("%s", id);
    if (EchoActif) printf(">>echo %s\n", id);
}

int get_int() {
    char buffer[NBCHIFFREMAX + 1];
    scanf("%s", buffer);
    if (EchoActif) printf(">>echo %s\n", buffer);
    return atoi(buffer);
}

// Déclarations des constantes

const enum {
    SPECIALITE_SIZE = 10,
    TRAVAILLEURS_SIZE = 50,
    CLIENTS_SIZE = 100,
    COMMANDES_SIZE = 500
};

// Specialité

typedef struct {
    Mot nom;
    int cout_horaire;
} Specialite;
```

```c
typedef struct {
    Specialite table[SPECIALITE_SIZE];
    int inserted;
} Specialites;

// Travailleurs

typedef struct {
    Mot nom;
    Booleen tag_specialite[SPECIALITE_SIZE];
    int heuresRealises;
} Travailleur;
typedef struct {
    Travailleur table[TRAVAILLEURS_SIZE];
    int inserted;
} Travailleurs;

// Client

typedef struct {
    Mot nom;
} Client;
typedef struct {
    Client table[CLIENTS_SIZE];
    int inserted;
} Clients;

// Commande et taches

typedef  struct {
    unsigned int nb_heures_requises;
    unsigned int nb_heures_effectuees;
} Tache;
typedef struct {
    Mot produit;
    Mot nom_client;
    Tache liste_taches[SPECIALITE_SIZE];
    Booleen complete;
    // L'identifiant du travailleur en charge de la tache
    unsigned int en_charge_tache[SPECIALITE_SIZE];
} Commande;
typedef struct {
    Commande table[COMMANDES_SIZE];
    int inserted;
} Commandes;

// Stockage global

typedef struct {
    Specialites specialites;
    Travailleurs travailleurs;
    Clients clients;
    Commandes commandes;
    // pour la commande progression passe
    int lastCommande;
    int lastSpecialite;
    int lastProgression;
} Stockage;
```

```c
// Declaration des prototypes de fonction ----------------------------------//
/*
    INFORMATION : Dans les prototypes : documentation des fonctions en elles-
    même; Dans la définition, il s'agit d'un commentaire libre qui explique
    la commande/l'entrée de l'utilisateur
*/
// fonctions helpers
/**
* [brief] : Permet de récupérer une entrée écrite par l'utilisateur.
* id [out]: Pointeur de type Mot qui permettra la modification de la variable
*           originale.
* [pre] : (l'entrée de l'utilisateur doit être valide)
**/
void get_id(Mot* id);

/**
* [brief] : A le même effet que get_id, mais retourne l'entrée en tant que int
*           plutôt que affecter dans l'argument via pointeur.
* [pre] : (l'entrée de l'utilisateur doit être un entier naturel)
* return: l'entier qui a été entré par l'utilisateur.
**/
int get_int();

/**
* [brief] : trouve et retourne l'entier n tel que Commandes->table[n].produit
*           (in 1) est égal à nom_produit (in 2)
* commandes [in] : pointeur constant d'une struct de type Commandes. Spécifie
*                  la struct dans laquelle on va chercher.
* nom_produit [in]: de type Mot. Spécifie le nom du produit à rechercher dans la
*                   struct donnée en tant que premier argument.
* return: l'indice du produit concerné dans le tableau de la structure Commandes
**/
int getIndex_cmd(const Commandes* commandes, Mot nom_produit);

/**
* [brief]:Trouve et retourne l'entier n tel que Travailleurs->table[n].nom (in1)
*         est égale nom
* travailleurs [in] : Pointeur constant d'une struct de type Travailleurs.
*                     Spécifie la struct dans laquelle on va chercher.
* nom [in] : De type Mot. Spécifie le nom du travailleur à rechercher dans la
*            struct donnée en tant que premier argument.
* return: L'indice du travailleur concerné dans le tableau de la structure
*         Travailleurs.
**/
int getIndex_trv(const Travailleurs* travailleurs, Mot nom);

/**
* [brief] : Trouve et retourne l'entier n tel que Specialites->table[n].nom
*           (in1) est égal à nom (in2)
* specialites [in]: Pointeur constant d'une struct de type Specialites.
*                   Spécifie la struct dans laquelle on va chercher.
* nom [in] : De type Mot. Spécifie le nom de la spécialité à rechercher dans la
*            struct donnée en tant que premier argument.
* return : L'indice de la spécialité concerné dans le tableau de la structure
*          Specialites.
**/
int getIndex_spe(const Specialites* specialites, Mot nom);
```

```c
/**
* [brief] : Une spécialité sera créée à l'aide de l'entrée utilisateur demandée.
* store [in-out] : Pointeur constant d'une struct de type Stockage. Spécifie la
*                  struct dans laquelle on va chercher et éditer.
* [pre] : store->specialites.inserted doit être initialisé par 0. Une
*         vérification est également faite si cette valeur ne dépasse pas le
*         nombre max de spécialités possibles.
**/
void traite_developpe(Stockage* store);

/**
* [brief] : La liste des spécialités sera affichée à l'utilisateur.
* store [in] : Pointeur constant d'une struct de type Stockage.
*              Spécifie la struct dans laquelle on va chercher.
**/
void traite_specialites(const Stockage* store);

/**
* [brief] : Un nouveau travailleur sera crée à l'aide de l'entrée utilisateur.
* store [in-out] : pointeur constant d'une struct de type Stockage. Spécifie la
*                  struct dans laquelle on va chercher et éditer.
* [pre] : store->travailleurs.inserted doit être initialisé par 0. Une
*         vérification est également faite si cette valeur ne dépasse pas le
*         nombre max de travailleurs possibles.
**/
void traite_embauche(Stockage* store);

/**
* [brief] : La liste des travailleurs sera affichée à l'utilisateur selon la
*           spécialité mentionnée ou "tous".
* store [in] : pointeur constant d'une struct de type Stockage. Spécifie la
*              struct dans laquelle on va chercher.
* [pre] : store->travailleurs.inserted et store->specialites.inserted doivent
*         être initialisés à 0 au préalable. Une vérification est également
*         faite si cette valeur ne dépasse pas le nombre max de travailleurs
*         possibles.
*
**/
void traite_travailleurs(const Stockage* store);

/**
* [brief] : Un nouveau client sera crée à l'aide de l'entrée utilisateur.
* store [in-out] : pointeur constant d'une struct de type Stockage. Spécifie la
*                  struct dans laquelle on va chercher et éditer.
* [pre] : store->client.inserted doit être initialisé à 0 au préalable. Une
*         vérification est également faite si cette valeur ne dépasse pas le
*         nombre max de travailleurs possibles.
**/
void traite_demarche(Stockage* store);

/**
* [brief] : La liste des commandes d'un client donnée par l'utilisateur sera
*           affichée à l'utilisateur.
*           L'utilisateur peut aussi mentionner "tous" si il veut voir toutes
*           les commandes de chaque client.
* store [in] : pointeur constant d'une struct de type Stockage. Spécifie la
*              struct dans laquelle on va chercher.
* [pre] : store->clients.inserted et store->commandes.inserted doivent
*         être initialisés à 0 au préalable. Une vérification est également
*         faite si cette valeur ne dépasse pas le nombre max de travailleurs
*         possibles.
**/
void traite_client(const Stockage* store);
```

```c
/**
* [brief] : Une nouvelle commande sera créée à l'aide de l'entrée utilisateur.
* store [in-out] : pointeur constant d'une struct de type Stockage. Spécifie la
*                  struct dans laquelle on va chercher et éditer.
**/
void traite_commande(Stockage* store);

/**
* [brief] : Affiche l'avancement actuel des commandes.
* store [in] : pointeur constant d'une struct de type Stockage. Spécifie la
*              struct dans laquelle on va chercher.
**/
void traite_supervision(const Stockage* store);

/**
* [brief] : Determine le travailleur le plus adéquat pour une spécialité et le
*           renvoie sous forme de char car le MAX_TRAVAILLEURS < CHAR_MAX.
* store [in] : pointeur constant d'une struct de type Stockage. Spécifie la
*              struct dans laquelle on va chercher.
* id_spe [in] : indice de la spécialité.
* return: indice du travailleur determiné.
**/
char determiner_travailleur_pour(const Stockage* store, int id_spe);

/**
* [brief] : Ajoute une specialité ainsi que le nombre d'heures requises à une
*           commande.
* store [in-out] : pointeur constant d'une struct de type Stockage. Spécifie la
*                  struct dans laquelle on va chercher et éditer.
**/
void traite_tache(Stockage* store);

/**
* [brief] Affiche la charge de travail d'un travailleur.
* store [in] : Pointeur constant d'une struct de type Stockage.
*              Spécifie la struct dans laquelle on va chercher.
**/
void traite_charge(const Stockage* store);

/**
* [brief]: Vérifie si toutes les tâches de la commande spécifiée dans l'argument
*          cmd sont complétées.
* store [in] : pointeur constant d'une struct de type Stockage. Spécifie la
*              struct dans laquelle on va chercher.
* cmd [in] : indice de la commande
* return : booleen VRAI ou FAUX indiquant si les tâches de la commande spécifiée
*          sont bien completées.
**/
Booleen verif_facturation(const Stockage* store, unsigned int cmd);

/**
* [brief] Vérifie si absolutement toutes les commandes sont complétées.
* store [in] :  pointeur constant d'une struct de type Stockage. Spécifie la
*                struct dans laquelle on va chercher.
* return: booleen VRAI ou FAUX indiquant si toutes les tâches de toutes les
*          commande sont bien completées.
**/
Booleen verif_facturationsGlobales(const Stockage* store);
```

```c
/**
* [brief] Indique une progression pour une tâche donnée par l'entrée utilisateur
* store [in] :  pointeur constant d'une struct de type Stockage. Spécifie la
*                struct dans laquelle on va chercher et éditer.
* return: booleen VRAI ou FAUX indiquant si toutes les tâches de toutes les
*         commande sont bien completées en utilisant les fonctions de vérifica-
*         tions au dessus.
**/
Booleen traite_progression(Stockage* store);

/**
* [brief] : Réaffecte un nouveau travailleur pour la dernière tâche passée dans
*           l'instruction progression.
* store [in] : pointeur constant d'une struct de type Stockage. Spécifie la
*              struct dans laquelle on va chercher et éditer.
* [pre] : traitre_progression() doit avoir été appelé au préalable afin d'ini-
*         tialiser les variatbles store->lastCommande et store->lastSpecialite
**/
void traite_passe(Stockage* store);

// Porte d'entrée du programme

int main(int argc, char* argv[]) {
    if (argc >= 2 && strcmp("echo", argv[1]) == 0) {
        EchoActif = VRAI;
    }
    setlocale(LC_ALL, "fr-FR");
    Stockage globalStore;
    globalStore.specialites.inserted = 0;
    globalStore.travailleurs.inserted = 0;
    globalStore.clients.inserted = 0;
    globalStore.commandes.inserted = 0;
    globalStore.lastCommande = -1;
    globalStore.lastSpecialite = -1;
    globalStore.lastProgression = -1;
    Mot buffer;
    while (VRAI) {
        get_id(&buffer);
        if (strcmp(buffer, "developpe") == 0) {
            traite_developpe(&globalStore);
            continue;
        }
        if (strcmp(buffer, "specialites") == 0) {
            traite_specialites(&globalStore);
            continue;
        }
        if (strcmp(buffer, "embauche") == 0) {
            traite_embauche(&globalStore);
            continue;
        }
        if (strcmp(buffer, "travailleurs") == 0) {
            traite_travailleurs(&globalStore);
            continue;
        }
        if (strcmp(buffer, "demarche") == 0) {
            traite_demarche(&globalStore);
            continue;
        }
        if (strcmp(buffer, "client") == 0) {
            traite_client(&globalStore);
            continue;
        }
        if (strcmp(buffer, "commande") == 0) {
            traite_commande(&globalStore);
            continue;
        }
        if (strcmp(buffer, "supervision") == 0) {
            traite_supervision(&globalStore);
```

```
            continue;
        }
        if (strcmp(buffer, "tache") == 0) {
            traite_tache(&globalStore);
            continue;
        }
        if (strcmp(buffer, "charge") == 0) {
            traite_charge(&globalStore);
            continue;
        }
        if (strcmp(buffer, "progression") == 0) {
            if (traite_progression(&globalStore)) {
                continue;
            }
            else {
                break;
            }
        }
        if (strcmp(buffer, "passe") == 0) {
            traite_passe(&globalStore);
            continue;
        }
        if (strcmp(buffer, "interruption") == 0) {
            break;
        }
        printf("!!! instruction inconnue >%s< !!!\n", buffer);
    }
    return 0;
}

// Helpers

int getIndex_cmd(const Commandes* commandes, Mot nom_produit) {
    // dans le cas où commandes->inserted vaut 0, la boucle n'est pas executée.
    // Il ne s'agit donc pas d'une précondition
    for (int i = 0; i < commandes->inserted; i++) {
        if (strcmp(commandes->table[i].produit, nom_produit) == 0) {
            return i;
        }
    }
    return -1; //fallback
}

int getIndex_trv(const Travailleurs* travailleurs, Mot nom) {
    // dans le cas où travailleurs->inserted == 0, la boucle n'est pas executée.
    // Il ne s'agit donc pas d'une précondition
    for (int i = 0; i < travailleurs->inserted; i++) {
        if (strcmp(travailleurs->table[i].nom, nom) == 0) {
            return i;
        }
    }
    return -1; //fallback
}

int getIndex_spe(const Specialites* specialites, Mot nom) {
    // dans le cas où specialites->inserted == 0, la boucle n'est pas executée.
    // Il ne s'agit donc pas d'une précondition
    for (int i = 0; i < specialites->inserted; i++) {
        if (strcmp(specialites->table[i].nom, nom) == 0) {
            return i;
        }
    }
    return -1; //fallback
}
```

```c
// Commandes

/*
* traite_developpe() || CF. prototype pour la doc de cette fonction
*
* Entrée utilisateur attendue: developpe <Mot nom_specialite> <int cout_horaire>
* où nom_specialite est le nom de la spécialité à créer
* et cout_horaire le prix de facturation d'une heure de travail pour cette spé
*/
void traite_developpe(Stockage* store) {
    Mot nom_specialite;
    get_id(&nom_specialite);
    int cout_horaire = get_int();
    assert(store->specialites.inserted >= 0);
    assert(store->specialites.inserted < SPECIALITE_SIZE);
    int i = store->specialites.inserted++;
    strcpy(store->specialites.table[i].nom, nom_specialite);
    store->specialites.table[i].cout_horaire = cout_horaire;
}

/*
* traite_specialites()|| CF. prototype pour la doc de cette fonction
*
* Entrée utilisateur attendue : specialites
* Cette instruction affiche les specialités enregistrées dans le programme dans
* l'ordre de leur déclaration, ainsi que le coût associé
*/
void traite_specialites(const Stockage* store) {
    printf(MSG_SPECIALITES);
    for (int i = 0; i < store->specialites.inserted; ++i) {
        const Specialite* specialite = &store->specialites.table[i];
        if (i == 0)
            printf("%s/%d", specialite->nom, specialite->cout_horaire);
        else
            printf(", %s/%d", specialite->nom, specialite->cout_horaire);
    }
    printf("\n");
}

/*
* traite_embauche()|| CF. prototype pour la doc de cette fonction
*
* Entrée utilisateur attendue : embauche <Mot travailleur> <Mot specialite>
* Recherche le travailleur qui a pour nom le premier argument (ci celui-ci
* n'existe pas, alors il sera créé),
* et lui assigne la spécialité donnée en tant que second argument
*/
void traite_embauche(Stockage* store) {
    Mot travailleur, specialite;
    get_id(&travailleur);
    get_id(&specialite);
    assert(store->travailleurs.inserted >= 0);
    assert(store->travailleurs.inserted < TRAVAILLEURS_SIZE);
    int id_trv = getIndex_trv(&store->travailleurs, travailleur);
    int id_spe = getIndex_spe(&store->specialites, specialite);
    if (id_trv >= 0) {
        store->travailleurs.table[id_trv].tag_specialite[id_spe] = VRAI;
    }
    else { // si le travailleur existe pas encore, on le crée.
        int new_id_trv = store->travailleurs.inserted++;
        Travailleur* new_trv = &store->travailleurs.table[new_id_trv];
        strcpy(new_trv->nom, travailleur); // enregistre son joli nom
        for (unsigned int i = 0; i < SPECIALITE_SIZE; ++i) {
            new_trv->tag_specialite[i] = (i == id_spe) ? VRAI : FAUX;
            // Necessité d'initialiser chaque cases pour éviter des bugs.
        }
        new_trv->heuresRealises = 0;
    }
```

```
}

/*
* traite_travailleurs() || CF. prototype pour la doc de cette fonction
*
* Entrée utilisateur attendue : travailleurs <Mot specialite || "tous">
* Affiche la liste des travailleurs compétents pour la spécialité donnée en
* argument, par ordre de déclaration des travailleurs.
* Si le nom de la spécialité vaut "tous", affiche la liste des travailleurs
* pour toutes les spécialités.
*/
void traite_travailleurs(const Stockage* store) {
    Mot specialite;
    get_id(&specialite);
    assert(store->specialites.inserted >= 0);
    assert(store->specialites.inserted < SPECIALITE_SIZE);
    assert(store->travailleurs.inserted >= 0);
    assert(store->travailleurs.inserted < TRAVAILLEURS_SIZE);
    int insertedSpe = store->specialites.inserted;
    int insertedTra = store->travailleurs.inserted;
    if (strcmp(specialite, "tous") == 0) {
        // la même chose que dans le else sauf qu'on loope sur toutes les spés
        // au lieu d'une seule
        for (int id_spe = 0; id_spe < insertedSpe; ++id_spe) {
            // la spé %s pourra être pise en charge par :
            printf(MSG_TRAVAILLEURS, store->specialites.table[id_spe].nom);
            int passedCheck = 0;
            for (int id_tra = 0; id_tra < insertedTra; ++id_tra) {
                const Travailleur* ce_trv = &store->travailleurs.table[id_tra];
                if (ce_trv->tag_specialite[id_spe] == VRAI) {
                    if (passedCheck == 0)
                        printf("%s", ce_trv->nom);
                    else
                        printf(", %s", ce_trv->nom);
                    passedCheck++;
                }
            }
            printf("\n");
        }
    }
    else {
        printf(MSG_TRAVAILLEURS, specialite);
        int indexSpe = getIndex_spe(&store->specialites, specialite);
        int passedCheck = 0;
        for (int id_tra = 0; id_tra < insertedTra; ++id_tra) {
            const Travailleur* ce_trv = &store->travailleurs.table[id_tra];
            if (ce_trv->tag_specialite[indexSpe] == VRAI) {
                if (passedCheck == 0)
                    printf("%s", ce_trv->nom);
                else
                    printf(", %s", ce_trv->nom);
                passedCheck++;
            }
        }
        printf("\n");
    }
}

/*
* traite_demarche() || CF. prototype pour la doc de cette fonction
*
* demarche <Mot nom_client>
* Déclare un nouveau client qui aura pour nom le premier argument
*/
void traite_demarche(Stockage* store) {
    Mot nom_client;
    get_id(&nom_client);
    assert(store->clients.inserted >= 0);
```

```
    assert(store->clients.inserted < CLIENTS_SIZE);
    strcpy(store->clients.table[store->clients.inserted].nom, nom_client);
    store->clients.inserted++;
}

/*
* traite_client() || CF. prototype pour la doc de cette fonction
*
* client <Mot nom_client>
* Affiche les commandes du client ayant pour nom le premier argument
*/
void traite_client(const Stockage* store) {
    Mot nom_client;
    get_id(&nom_client);
    assert(store->clients.inserted >= 0);
    assert(store->clients.inserted < CLIENTS_SIZE);
    assert(store->commandes.inserted >= 0);
    assert(store->commandes.inserted < MAX_COMMANDES);
    int nb_clients = store->clients.inserted;
    int nb_commandes = store->commandes.inserted;
    if (strcmp(nom_client, "tous") == 0) {
        // fais la mm chose que le else, mais sur tous les clients au lieu d'un
        for (int clientsI = 0; clientsI < nb_clients; ++clientsI) {
            const Client* client = &store->clients.table[clientsI];
            printf(MSG_CLIENT, client->nom);
            int passedCheck = 0;
            for (int indexCmd = 0; indexCmd < nb_commandes; ++indexCmd) {
                const Commande* commande = &store->commandes.table[indexCmd];
                if (strcmp(commande->nom_client, client->nom) == 0) {
                    if (passedCheck == 0)
                        printf("%s", commande->produit);
                    else
                        printf(", %s", commande->produit);
                    passedCheck++;
                }
            }
            printf("\n");
        }
    }
    else {
        printf(MSG_CLIENT, nom_client);
        int passedCheck = 0;
        // pour chaque commande ...
        for (int i = 0; i < nb_commandes; ++i) {
            const Commande* commande = &store->commandes.table[i];
            // ... si le client actuel est en charge de celle ci, on l'affiche
            if (strcmp(commande->nom_client, nom_client) == 0) {
                if (passedCheck == 0)
                    printf("%s", commande->produit);
                else
                    printf(", %s", commande->produit);
                passedCheck++;
            }
        }
        printf("\n");
    }
}

/*
* traite_commande() || CF. prototype pour la doc de cette fonction
*
* Entrée utilisateur : commande <Mot produit> <Mot nom_client>
* Ajoute une commande au programme, qui aura pour nom le premier argument.
* Le second spécifie le nom du client qui l'a commandé.
*/
void traite_commande(Stockage* store) {
    Mot produit, nom_client;
    get_id(&produit);
```

```
    get_id(&nom_client);
    assert(store->commandes.inserted >= 0);
    assert(store->commandes.inserted < MAX_COMMANDES);
    // initialisation de la nouvelle commande
    const unsigned int i = store->commandes.inserted++;
    strcpy(store->commandes.table[i].nom_client, nom_client);
    strcpy(store->commandes.table[i].produit, produit);
    store->commandes.table[i].complete = FAUX;
    //initialisation de la liste de tâches
    for (unsigned int speNbr = 0; speNbr < MAX_SPECIALITES; ++speNbr)
    {
        store->commandes.table[i].liste_taches[speNbr].nb_heures_requises = 0;
        store->commandes.table[i].liste_taches[speNbr].nb_heures_effectuees = 0;
    }
}

/*
* traite_supervision() || CF. prototype pour la doc de cette fonction
*
* Affiche l'avancement actuel des commandes sous le format suivant:
*    "etat des taches pour <nom_produit> : [<liste>, ..., <liste>]"
*    avec <liste> valant "<specialite>:<heures effectuées>/<heures nécessaires>"
*/
void traite_supervision(const Stockage* store) {
    for (int i_cmd = 0; i_cmd < store->commandes.inserted; ++i_cmd)
    {
        printf(MSG_SUPERVISION, store->commandes.table[i_cmd].produit);
        Booleen isFirstTime = VRAI;
        for (int i_spe = 0; i_spe < store->specialites.inserted; ++i_spe)
        {
            const Commande* commande = &store->commandes.table[i_cmd];
            const Tache* tacheCourante = &commande->liste_taches[i_spe];
            if (tacheCourante->nb_heures_requises) { //valeur si vide : 0 = faux
                if (isFirstTime) {
                    isFirstTime = FAUX;
                }
                else {
                    printf(", ");
                }
                printf("%s:", store->specialites.table[i_spe].nom);
                printf("%d/", tacheCourante->nb_heures_effectuees);
                printf("%d", tacheCourante->nb_heures_requises);
            }
        }
        printf("\n");
    }
}

/**
 * Détermine le travailleur le plus adapté pour la réalisation d'une tâche.
 * Une tâche étant déterminée par une spécialité, il attends donc en ENTREE
 * l'identifiant d'une spécialitée attendue, et afin de faire ses opérations
 * le pointeur du tableau de travailleurs
 * En SORTIE, il retourne l'index du travailleur à prendre en charge.
**/
char determiner_travailleur_pour(const Stockage* store, int id_spe) {
    unsigned int retval = TRAVAILLEURS_SIZE;
    // initialise le tableau qui va contenir la charge de chaque travailleur
    int totalWorker[TRAVAILLEURS_SIZE];
    for (int i = 0; i < TRAVAILLEURS_SIZE; ++i) totalWorker[i] = 0;
    // boucle sur chaque travailleur dont sa charge sera calculée
    for (int id_trv = 0; id_trv < store->travailleurs.inserted; ++id_trv) {
        // pour chaque commande ... :
        for (int i_cmd = 0; i_cmd < store->commandes.inserted; ++i_cmd) {
            const Commande* curr_cmd = &store->commandes.table[i_cmd];
            // regarde si le travailleur actuel est sur la spé en question ... :
            for (int i_spe = 0; i_spe < store->specialites.inserted; ++i_spe) {
                if (curr_cmd->en_charge_tache[i_spe] == id_trv) {
```

```
                    // ... et ajoute un peu de charge de travail selon les stats
                    // de la spé
                    totalWorker[id_trv] += (
                            curr_cmd->liste_taches[i_spe].nb_heures_requises
                            - curr_cmd->liste_taches[i_spe].nb_heures_effectuees
                    );
                }
            }
        }
    }
    // Une fois la charge de chaque travailleur calculée, on fait une recherche
    // du travailleur ayant la plus petite charge
    int lowestHours = -1;
    for (int id_trv = 0; id_trv < store->travailleurs.inserted; ++id_trv) {
        if (store->travailleurs.table[id_trv].tag_specialite[id_spe] == VRAI) {
            if (lowestHours < 0) {
                lowestHours = totalWorker[id_trv];
                retval = id_trv;
            }
            if (EchoActif) {
                printf(">>> travailleur trouvé %s %d: %d (%d)\n",
                       store->travailleurs.table[id_trv].nom,
                       id_trv,
                       totalWorker[id_trv],
                       lowestHours);
            }
            if (lowestHours > totalWorker[id_trv]) {
                retval = id_trv;
                lowestHours = totalWorker[id_trv];
            }
        }
    }
    return retval;
}

/*
* traite_tache() || CF. prototype pour la doc de cette fonction
*
* tache <Mot commande> <Mot specialite> <int heures>
* Ajoute une specialité ainsi que le nombre d'heures requises à une commande
*/
void traite_tache(Stockage* store) {
    Mot commande, specialite;
    get_id(&commande);
    get_id(&specialite);
    int heures = get_int();
    // recherche de la commande concernée puis de la spécialité concernée
    const int cmd_i = getIndex_cmd(&store->commandes, commande);
    const int id_spe = getIndex_spe(&store->specialites, specialite);
    // La commande et la specialité doivent exister, nous vérifions si la valeur
    // retournée est au dessus de -1, car -1 = inexistance de la spécialité ou
    // de la commande
    assert(cmd_i > -1);
    assert(id_spe > -1);
    // initialisation de la tâche pour la commande en question
    Commande* commandeCourante = &store->commandes.table[cmd_i];
    Tache* tacheCourante = &commandeCourante->liste_taches[id_spe];
    tacheCourante->nb_heures_requises = heures;
    // Assignation du travailleur à la tache
    const unsigned int id_worker = determiner_travailleur_pour(store, id_spe);
    if (id_worker < TRAVAILLEURS_SIZE) {
        commandeCourante->en_charge_tache[id_spe] = id_worker;
    }
    else if (EchoActif) {
        printf("!!! aucun travailleur trouvé pour traiter la spé demandée.\n");
    }
}
```

```c
/*
* traite_charge() || CF. prototype pour la doc de cette fonction
*
* charge <Mot nom_travailleur>
* Donné le nom du travailleur, donne sa charge de travail
*/
void traite_charge(const Stockage* store) {
    Mot nom_trv;
    get_id(&nom_trv);
    int nb_commandes = store->commandes.inserted;
    const int id_travailleur = getIndex_trv(&store->travailleurs, nom_trv);
    // La commande et la specialité doivent exister, nous vérifions si la valeur
    // retournée est au dessus de -1, car -1 = inexistance de la spécialité
    // ou de la commande
    assert(id_travailleur > -1);
    printf(MSG_CHARGE, nom_trv);
    Booleen isFirstTime = VRAI;
    for (int indexCmd = 0; indexCmd < nb_commandes; indexCmd++) {
        for (int indexSpe = 0; indexSpe < SPECIALITE_SIZE; indexSpe++) {
            const Commande* commande = &store->commandes.table[indexCmd];
            const Tache* tache = &commande->liste_taches[indexSpe];
            if (commande->en_charge_tache[indexSpe] == id_travailleur
                && tache->nb_heures_requises != tache->nb_heures_effectuees) {
                if (isFirstTime) {
                    isFirstTime = FAUX;
                }
                else {
                    printf(", ");
                }
                printf("%s/%s/%dheure(s)",
                       commande->produit,
                       store->specialites.table[indexSpe].nom,
                       tache->nb_heures_requises - tache->nb_heures_effectuees
                );
            }
        }
    }
    printf("\n");
}

/*
* verif_facturation() || CF. prototype pour la doc de cette fonction
*/
Booleen verif_facturation(const Stockage* store, const unsigned int cmd) {
    int compteur = 0;
    for (int spec = 0; spec < SPECIALITE_SIZE; ++spec) {
        const Tache* tache = &store->commandes.table[cmd].liste_taches[spec];
        if (tache->nb_heures_effectuees >= tache->nb_heures_requises) {
            compteur++;
        }
    }
    if (compteur >= SPECIALITE_SIZE) {
        return VRAI;
    }
    else {
        return FAUX;
    }
}

/*
* verif_facturationsGlobales() || CF. prototype pour la doc de cette fonction
*/
Booleen verif_facturationsGlobales(const Stockage* store) {
    int compteurGlobal = 0;
    for (int cmd = 0; cmd < store->commandes.inserted; ++cmd) {
        if (store->commandes.table[cmd].complete == VRAI) {
            compteurGlobal++;
        }
```

```
    }
    if (compteurGlobal == store->commandes.inserted) {
        return VRAI;
    }
    else {
        return FAUX;
    }
}

/*
* traite_progression() || CF. prototype pour la doc de cette fonction
*
* Entrée utilisateur attendue tache :
*   progression <Mot produit> <Mot specialite> <int heures_travaillees>
* Donné le nom d'une commande, et le nom de la spécialité à mettre à jour,
* modifie son nombre d'heures travaillées
*/
Booleen traite_progression(Stockage* store) {
    Mot commande, specialite;
    get_id(&commande);
    get_id(&specialite);
    int heures_travaillees = get_int();
    // fais la correspondance entre entrée et donnée stockée, puis mets à jour
    // la progression des taches
    const unsigned int cmd_i = getIndex_cmd(&store->commandes, commande);
    const unsigned int id_spe = getIndex_spe(&store->specialites, specialite);
    Commande* current_cmd = &store->commandes.table[cmd_i];
    current_cmd->liste_taches[id_spe].nb_heures_effectuees +=heures_travaillees;
    // stockage d'un historique court pour la commande passe
    store->lastCommande = cmd_i;
    store->lastSpecialite = id_spe;
    store->lastProgression = heures_travaillees;
    // vérification de la facturation, et traitement conséquent au besoin
    Booleen checkFacturation = verif_facturation(store, cmd_i);
    if (checkFacturation && current_cmd->complete == FAUX) {
        printf(MSG_FACTURATION, commande);
        Booleen firstDone = VRAI;
        // pour chaque spé, si celle-ci est définie (nb_heures_requises != 0)...
        for (int id_spe = 0; id_spe < SPECIALITE_SIZE; ++id_spe) {
            if (current_cmd->liste_taches[id_spe].nb_heures_requises != 0) {
                if (firstDone) {
                    firstDone = FAUX;
                }
                else {
                    printf(", ");
                }
                // ... calcul du cout de la commande et affichage
                Gros_entier cout_commande = (
                        current_cmd->liste_taches[id_spe].nb_heures_effectuees
                        * store->specialites.table[id_spe].cout_horaire
                );
                printf("%s:%llu",
                       store->specialites.table[id_spe].nom,
                       cout_commande
                );
                current_cmd->complete = VRAI;
            }
        }
        printf("\n");
    }
    // idem, mais pour la facturation globale
    Booleen checkFacturationsGlobales = verif_facturationsGlobales(store);
    if (checkFacturationsGlobales) {
        printf(MSG_FACTURATION_SEULE);
        Booleen firstDone = VRAI;
        // pour chaque commande, regarder chaque client ...
        for (int client = 0; client < store->clients.inserted; ++client) {
            Gros_entier cout_total_client = 0;
```

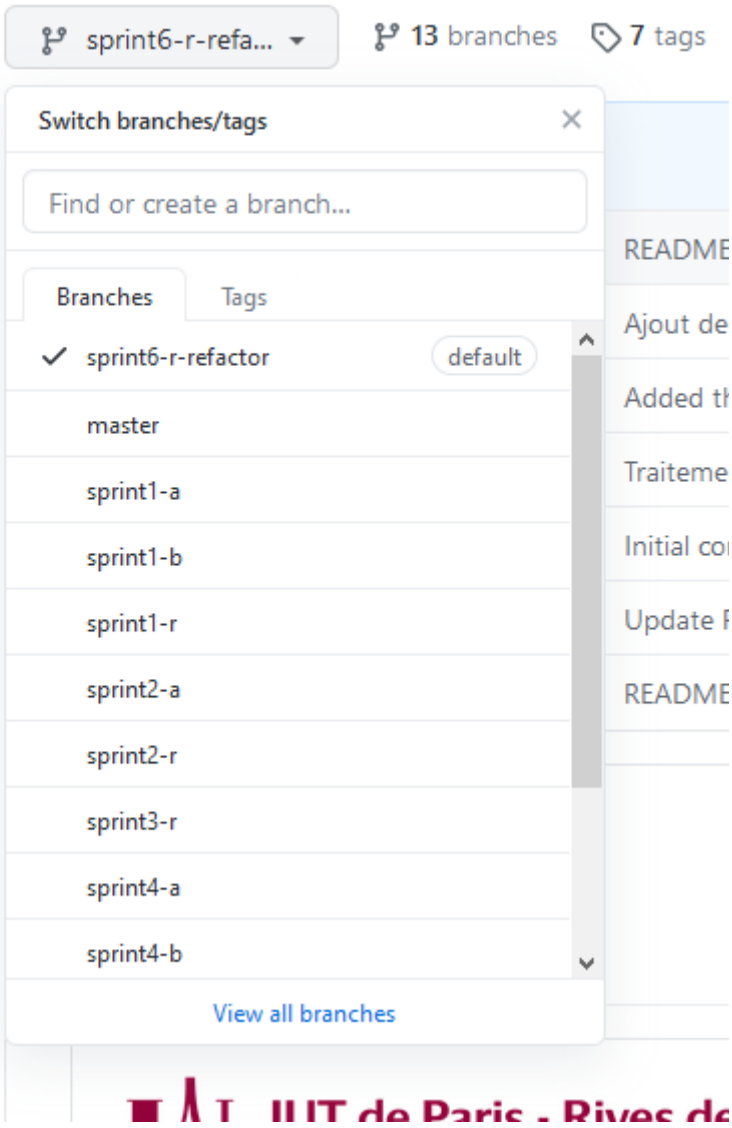
```
            Mot* nom_de_ce_client = &store->clients.table[client].nom;
            for (int id_cmd = 0; id_cmd < store->commandes.inserted; ++id_cmd) {
                Commande* current_cmd = &store->commandes.table[id_cmd];
                // ... et si celui-ci est bien en charge de cette commande,
                // regarder chaque tache ...
                if (strcmp(current_cmd->nom_client, *nom_de_ce_client) == 0) {
                    for (int spec = 0; spec < SPECIALITE_SIZE; ++spec) {
                        Tache tacheCourante = current_cmd->liste_taches[spec];
                        // puis mettre à jour le coût total du client en fon-
                        // ction de son nombre d'heures effectuées et du coût
                        // horairede la spé
                        cout_total_client += (
                                tacheCourante.nb_heures_effectuees
                                * store->specialites.table[spec].cout_horaire
                        );
                    }
                }
            }
            // enfin, une fois le coût total du client calculé, faire l'affi-
            // chage.
            if (firstDone) {
                firstDone = FAUX;
            }
            else {
                printf(", ");
            }
            printf("%s:%llu", nom_de_ce_client, cout_total_client);
        }
        printf("\n");
        return FAUX;
    }
    return VRAI;
}

/*
* traite_passe() || CF. prototype pour la doc de cette fonction
*
* entrée utilisateur attendue : passe
* réaffecte un nouveau travailleur pour la dernière tâche passée en argument de
* l'instruction progression
*/
void traite_passe(Stockage* store) {
    assert(store->lastCommande && store->lastSpecialite);
    const unsigned int id_worker = determiner_travailleur_pour(
            store,
            store->lastSpecialite
    );
    // et en conséquence, change l'id du travailleur par celui nouvellement
    // trouvé dans la dernière spécialitée utilisée avec la commande traite
    store->commandes.table[store->lastCommande]
            .en_charge_tache[store->lastSpecialite] = id_worker;
    store->lastCommande = 0;
    store->lastSpecialite = 0;
}
```

Liens pratiques

Lien vers notre GitHub de projet :

https://github.com/LoganTann/IAP-PJ1

Le GitHub du projet contient des releases .exe du projet ainsi que l'entièreté du code source réparti sur plusieurs branches distinctes.



Fin du rapport