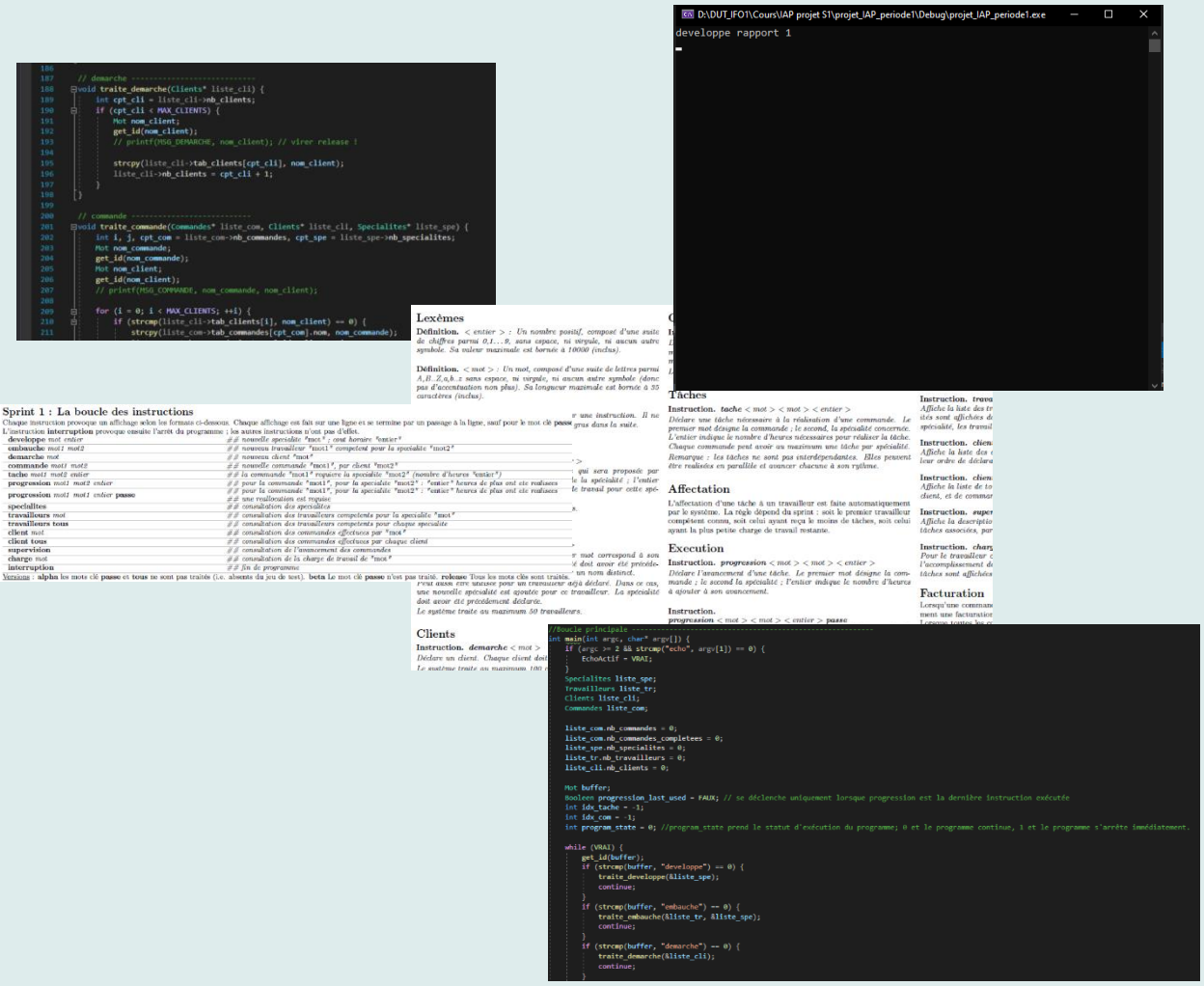
Université de Paris

IAP

*S1 période A*
*2020-2021*

# Rapport de projet sur la conception d'un système de gestion informatisée de commandes




*Anthony ZAKANI*
*et Mehdi ZAOUI*
Groupe 111

# TABLE DES MATIERES


**I. Présentation du problème**
Page 3

**II. Preuves de validation**
Captures d'écran du site diffchecker.com pour chaque sprint réalisé

**III. Bilan du projet**
Page 9

**IV. Code source des versions release du projet**
Sprint 0 : p. 11
Sprint 1r : p. 12
Sprint 2r : p. 15
Sprint 3r : p. 22
Sprint 4-5r : p. 30
Sprint 6r : p. 43


*À travers ce rapport, nous vous présenterons les difficultés et défis qui ont rythmé la réalisation des différents sprints de notre programme.*

*À l'aide du langage C, nous avons dû améliorer, par itérations, un programme de traitement de commandes.*

# *Présentation du problème*

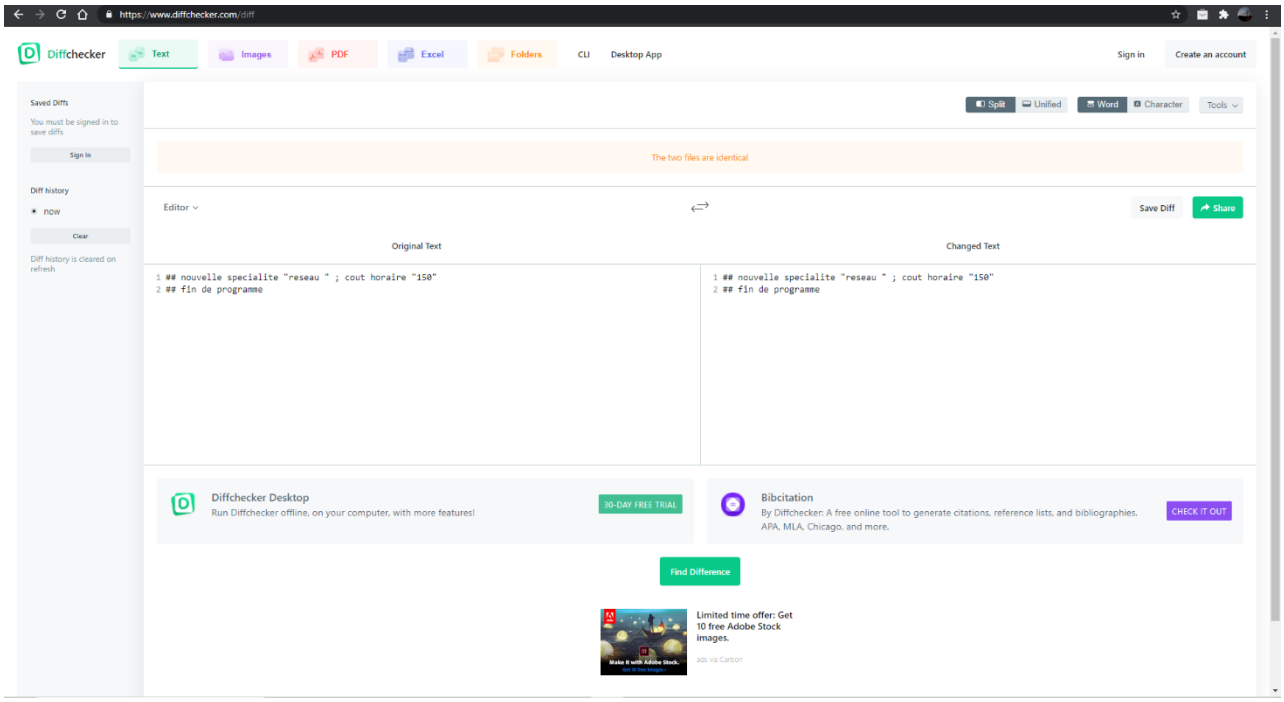
Le principe du projet d'IAP qui nous a été assigné était de développer un système de gestion d'une entreprise réalisant des commandes, en permettant la création de spécialités de tâches, de travailleurs compétents pour lesdites spécialités, de clients commandant des produits, qui pour être réalisés nécessitent telle ou telle tâche, nécessitant elle-même telle ou telle spécialité pour être réalisée.

Toute la difficulté de ce projet était donc d'adopter une approche nous permettant de comprendre ce qu'il faudra coder avant de se lancer. D'une certaine manière, nous devions être l'entreprise recevant une commande pour un logiciel de traitement de commandes.
Dès le sprint 0, nous devions donc, avant de passer aux suivants, comprendre comment le programme fonctionnait, puis une fois ceci fait, nous devions planifier quelles fonctions devraient être codées, quelle était la syntaxe, la méthode utilisée par le développeur du sprint 0 pour pouvoir appliquer un formatage similaire (qui serait compréhensible et efficace) dès le sprint 1 alpha.

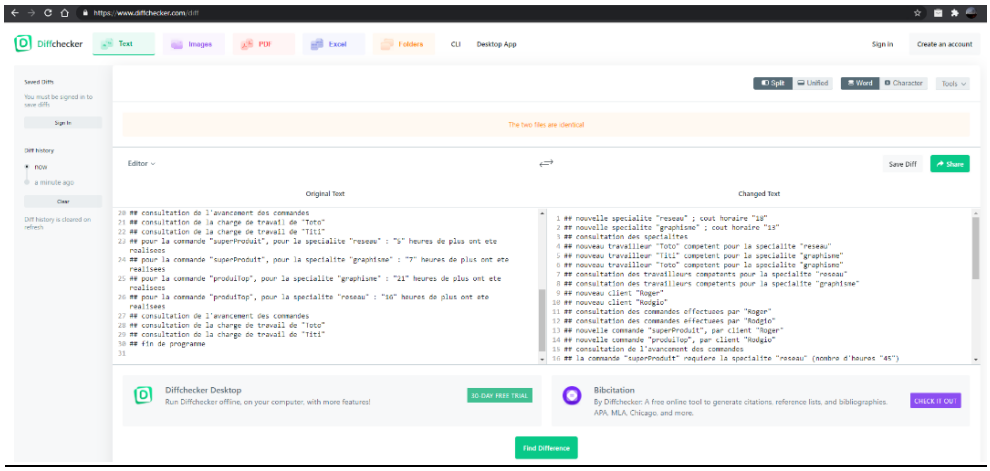
Ce projet était donc une manière de se mettre à la place, pour la première fois, d'une entreprise réalisant des solutions logicielles : nous devions répondre à un cahier des charges clair qui nous demandait de coder à chaque étape de nouvelles fonctionnalités de notre logiciel en respectant les demandes de notre client.
Nous avions également une certaine latitude par rapport à la manière d'implémenter ces changements, même si l'apport du guide technique dans les premières étapes était essentiel à notre réflexion.

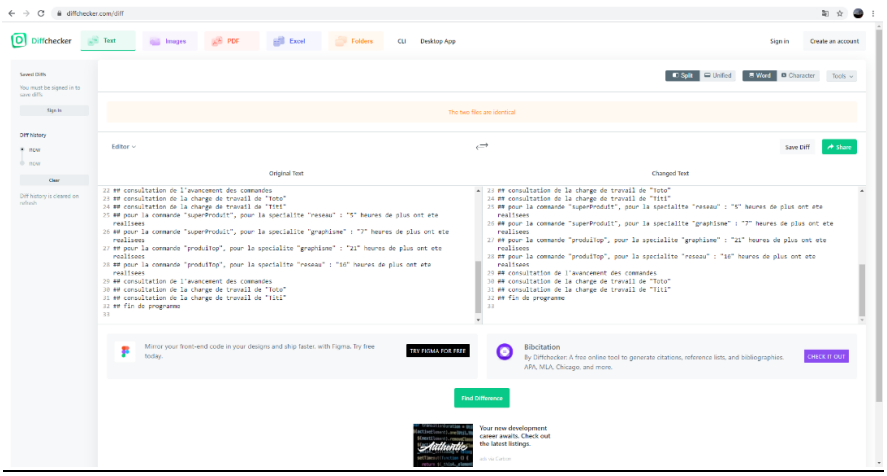
La méthode de développement utilisée (dite « agile ») ici fonctionnait sur un principe simple : préparer les bases du programme (sprint 0, 1), ensuite implémenter les fonctionnalités préparées (sprint 2 et 3), et enfin automatiser le traitement des données issues de ses fonctionnalités (sprint 4, 5 et 6).
Ainsi, au cours du projet, nous devions développer le fonctionnement « en boucle » du programme en mettant en place les différentes instructions, puis implémenter le stockage des données entrées pour chaque instruction avant enfin de coder la réaction du programme à chaque donnée rentrée ou modifiée.

# *Preuves de validation*

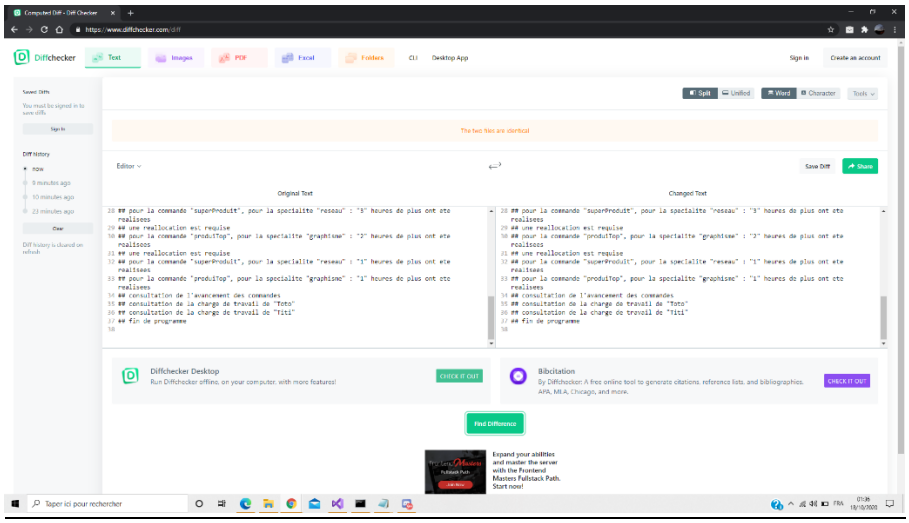


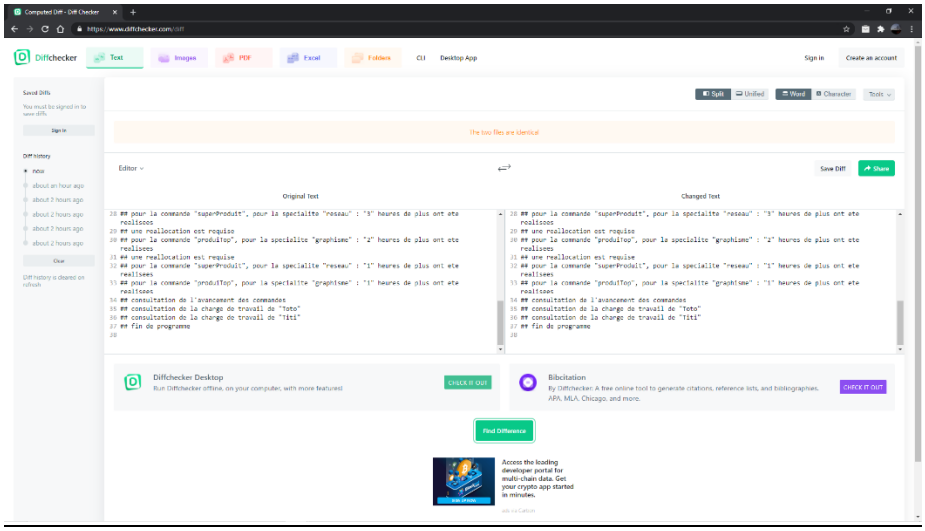

## Rendu du sprint 0



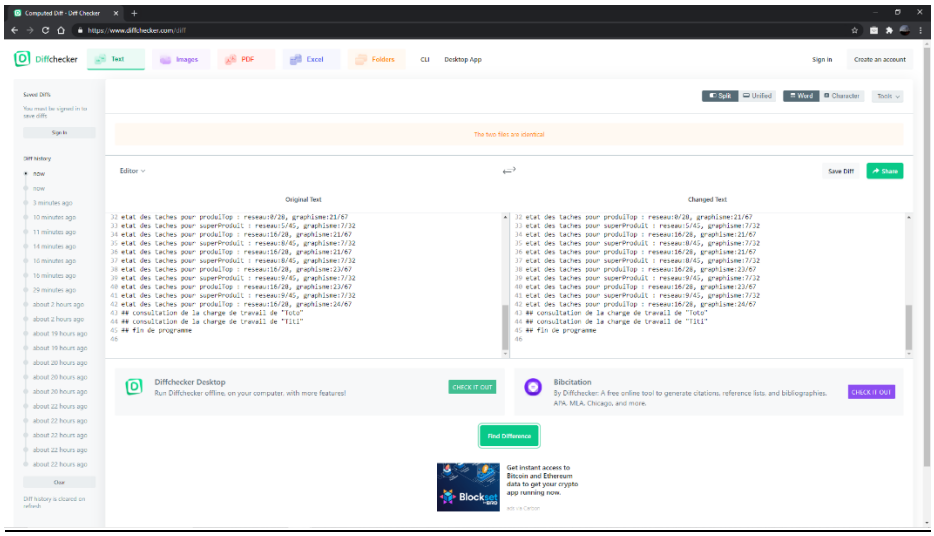




## Rendus du sprint 1

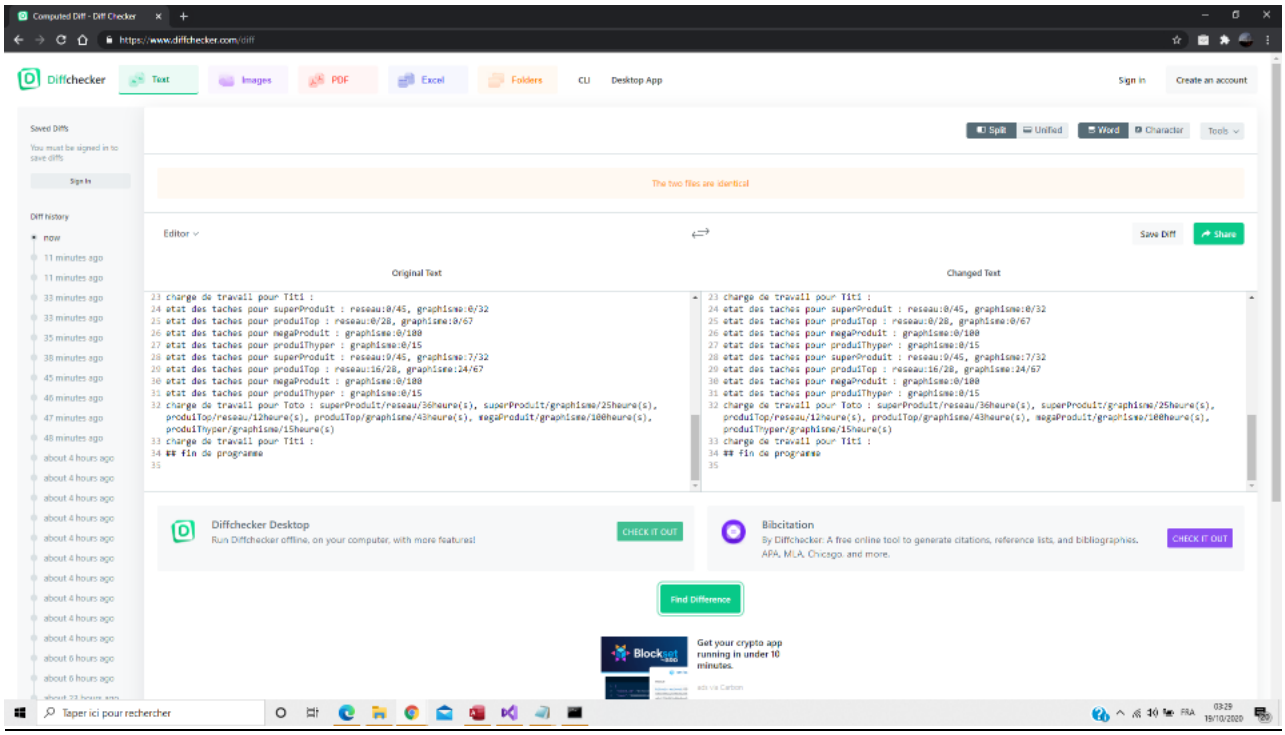




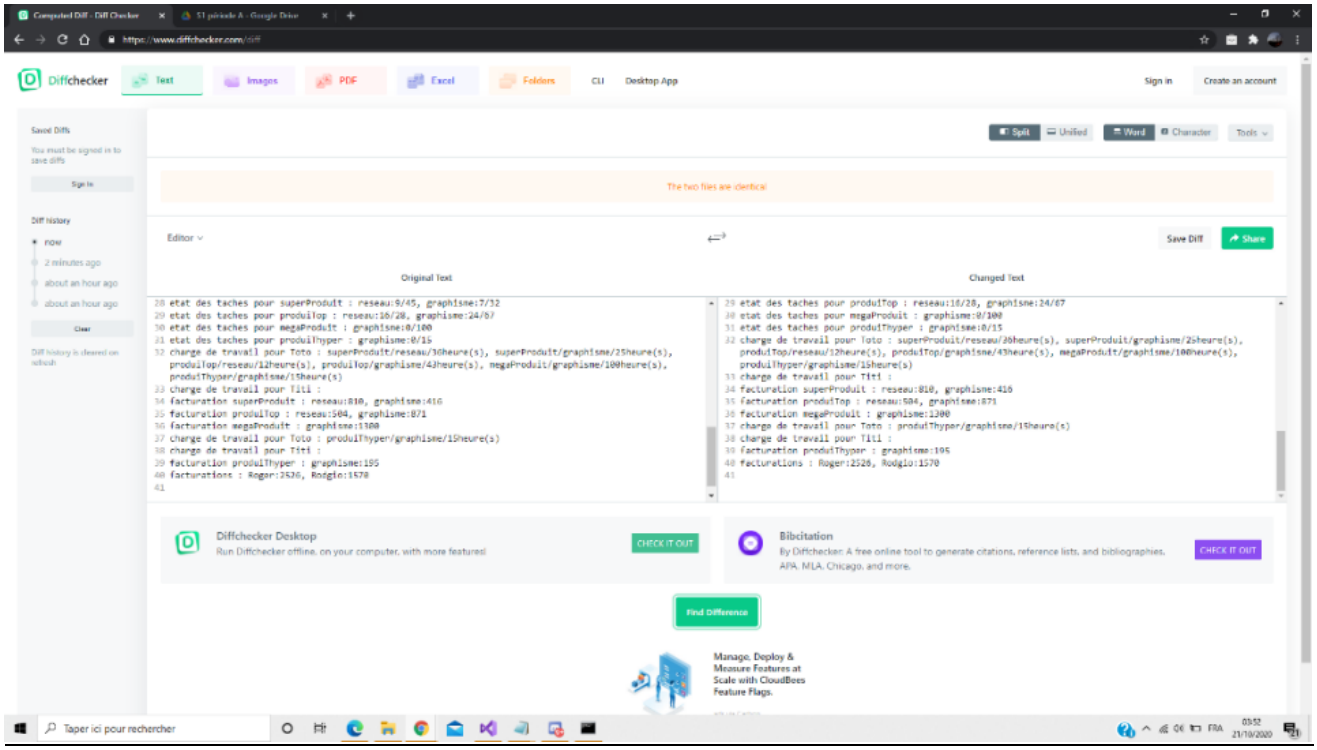
## Rendus du sprint 2



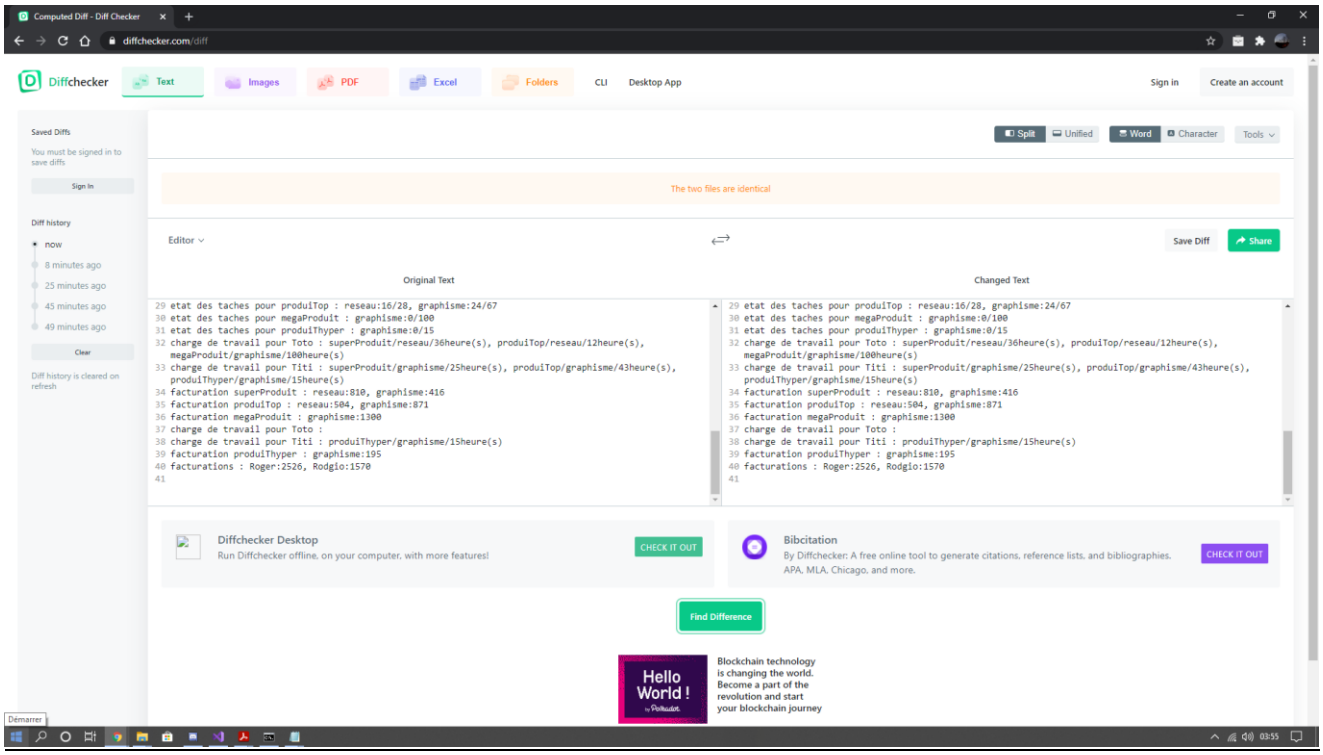
## Rendu du sprint 3



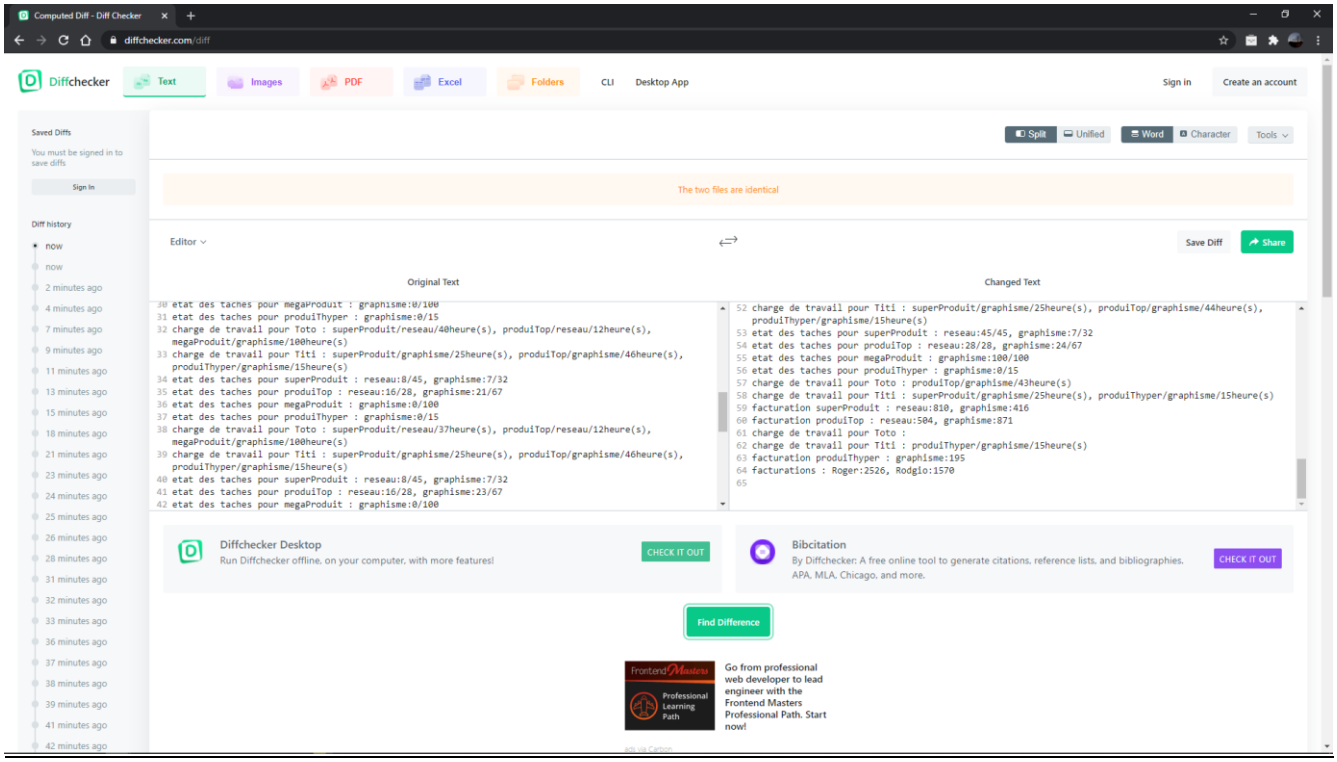

## Rendu du sprint 4 alpha




## Rendu du sprint 5 alpha




## Rendu du sprint 5 release




## Rendu du sprint 6 release

# Bilan du projet

Au cours de la réalisation de ce projet, nous avons pu apprendre une méthodologie de développement : la méthode par « sprint ». L'usage de cette méthode nous a permis de rendre le développement du logiciel plus segmenté, nous permettant donc de nous concentrer sur une fonctionnalité ou un aspect du programme en particulier.
Nous étions donc beaucoup moins éparpillés que si nous avions eu à coder tout d'un coup.

Cela a été pour nous une première dans plein de domaines différents (Mehdi faisait de la programmation pour la première fois tandis que j'avais déjà programmé auparavant mais jamais en utilisant la méthode par sprint).
Ce qui était difficile pour Mehdi était d'apprendre à programmer, pour commencer, car il n'a pas eu la chance d'avoir une section spécialisée dans la programmation au lycée, comparé à d'autres membres de la promo qui eux avaient déjà eu l'occasion de programmer et donc avaient des bases pour le projet, il a trouvé que les TD/TP l'ont aidé mais malgré cela il manquait d'expérience dans le domaine.
Mehdi avait beaucoup de mal avec les fonctions, le stockage des spécialités par exemple, et nous nous sommes aidés sur ces points. Son logiciel Visual Studio plantait aussi assez souvent. Nous avons travaillé ensemble afin de comprendre le but du projet, les connaissances que nous allions mobiliser ainsi que les choses qui pourraient nous bloquer.
Dès les premiers sprints, nous avons dû comprendre le fonctionnement des guides techniques fournis, ce qui a pris beaucoup de temps. Il a notamment fallu comprendre à quoi devait servir les structures qui nous étaient fournies, comment les utiliser (ce qui a généré beaucoup d'essais, de tests et de débogages) avant finalement de demander de l'aide à nos professeurs, qui nous ont mis sur la voie. Nous avons beaucoup appris sur le fonctionnement des structures, ce qui était stocké ou pas, l'usage des pointeurs, la définition de fonctions et le prototypage de celles-ci, entre autres.

D'autres aspects du logiciel nous aussi amené à changer d'approche, notamment au moment du sprint 4r et du sprint 6r: il nous a fallu nous y reprendre plusieurs fois pour faire un algorithme efficace qui décernerait la tâche au travailleur correspondant exactement à ce qui était attendu (au point de tester des centaines de fois pendant plusieurs jours nos conditions, l'implémentation de chaque tâche, les attributs de chaque travailleur). C'était l'une des phases les plus dures du projet, qui nous a pris le plus de temps. Notre progression sur ce projet a été très hachée : nous avons complété les trois premiers sprints en une dizaine de jours à un rythme plutôt lent, pour comprendre ce que nous devions faire, avant de terminer les trois autres sur une bonne semaine. Nous étions donc soulagés de voir que le sprint 6 release avait bien été validé lors de la recette (malgré une petite frayeur avec des virgules manquantes, autre difficulté de notre projet).

En conclusion, travailler sur ce projet était une première expérience particulière pour nous deux : nous avons dû nous adapter, comprendre et appliquer les connaissances acquises au cours de cette période pour pouvoir, à la manière d'une entreprise, répondre à la commande qui nous était faite. Nous sommes donc assez satisfaits du résultat obtenu.

# *Code source des versions release du projet*

# SPRINT 0

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

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

// Messages emis par les instructions ----------------------------------------
#define MSG_DEVELOPPE "## nouvelle specialite \"%s \" ; cout horaire \"%d\"\n"
#define MSG_INTERRUPTION "## fin de programme\n"

// Lexemes -------------------------------------------------------------------
#define LGMOT 35
#define NBCHIFFREMAX 5
typedef char Mot[LGMOT + 1];
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);
}

// Instructions --------------------------------------------------------------
// developpe -----------------------------
void traite_developpe() {
   Mot nom_specialite;
   get_id(nom_specialite);
   int cout_horaire = get_int();
   printf(MSG_DEVELOPPE, nom_specialite, cout_horaire);
}
// interruption ------------------------
void traite_interruption() {
   printf(MSG_INTERRUPTION);
}

//Boucle principale ---------------------------------------------------------
int main(int argc, char* argv[]) {
   if (argc >= 2 && strcmp("echo", argv[1]) == 0) {
      EchoActif = VRAI;
   }
   Mot buffer;
   while (VRAI) {
      get_id(buffer);
      if (strcmp(buffer, "developpe") == 0) {
         traite_developpe();
         continue;
      }
      if (strcmp(buffer, "interruption") == 0) {
         traite_interruption();
         break;
      }
      printf("!!! instruction inconnue >%s< !!!\n", buffer);
   }
   return 0;
}
```

# SPRINT 1 RELEASE

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

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

// Messages emis par les instructions ------------------------------------
--
#define MSG_DEVELOPPE "## nouvelle specialite \"%s\" ; cout horaire \"%d\"\n"
#define MSG_EMBAUCHE "## nouveau travailleur \"%s\" competent pour la
specialite \"%s\"\n"
#define MSG_DEMARCHE "## nouveau client \"%s\"\n"
#define MSG_COMMANDE "## nouvelle commande \"%s\", par client \"%s\"\n"
#define MSG_TACHE "## la commande \"%s\" requiere la specialite \"%s\"
(nombre d'heures \"%d\")\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"
#define MSG_SPECIALITES "## consultation des specialites\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_CLIENT "## consultation des commandes effectuees par \"%s\"\n"
#define MSG_CLIENT_TOUS "## consultation des commandes effectuees par chaque
client\n"
#define MSG_SUPERVISION "## consultation de l'avancement des commandes\n"
#define MSG_CHARGE "## consultation de la charge de travail de \"%s\"\n"
#define MSG_INTERRUPTION "## fin de programme\n"

// Lexemes ---------------------------------------------------------------
--
#define LGMOT 35
#define NBCHIFFREMAX 5
typedef char Mot[LGMOT + 1];
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);
}

// Instructions ----------------------------------------------------------
--
// developpe ---------------------------
void traite_developpe() {
   Mot nom_specialite;
   get_id(nom_specialite);
   int cout_horaire = get_int();
   printf(MSG_DEVELOPPE, nom_specialite, cout_horaire);
}

// embauche ----------------------------
void traite_embauche() {
```

```
    Mot nom_travailleur;
    get_id(nom_travailleur);
    Mot nom_specialite;
    get_id(nom_specialite);
    printf(MSG_EMBAUCHE, nom_travailleur, nom_specialite);
}

// demarche ----------------------------
void traite_demarche() {
    Mot nom_client;
    get_id(nom_client);
    printf(MSG_DEMARCHE, nom_client);
}

// commande ----------------------------
void traite_commande() {
    Mot nom_commande;
    get_id(nom_commande);
    Mot nom_client;
    get_id(nom_client);
    printf(MSG_COMMANDE, nom_commande, nom_client);
}
// tache -------------------------------
void traite_tache() {
    Mot nom_commande;
    get_id(nom_commande);
    Mot nom_specialite;
    get_id(nom_specialite);
    int nombre_heures = get_int();
    printf(MSG_TACHE, nom_commande, nom_specialite, nombre_heures);
}
// progression -------------------------
void traite_progression() {
    Mot nom_commande;
    get_id(nom_commande);
    Mot nom_specialite;
    get_id(nom_specialite);
    int    nombre_heures_sup = get_int();
    printf(MSG_PROGRESSION, nom_commande, nom_specialite, nombre_heures_sup);
}
void passe() {
    printf(MSG_PASSE); // chaque instruction est execut?e en ligne, l'une
apr?s l'autre; donc si on dit progression
}                  // et que tous les arguments pour progression sont pass?s,
on passe ? l'instruction suviante
// specialites -------------------------
void traite_specialites() {
    printf(MSG_SPECIALITES);
}
// travailleurs ------------------------
void traite_travailleurs() {
    Mot nom_specialite;
    get_id(nom_specialite);
    if (strcmp(nom_specialite, "tous") == 0)
        printf(MSG_TRAVAILLEURS_TOUS);
    else
        printf(MSG_TRAVAILLEURS, nom_specialite);
}
// 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);
}
// supervision -------------------------
void traite_supervision() {
   printf(MSG_SUPERVISION);
}
// charge ------------------------------
void traite_charge() {
   Mot nom_travailleur;
   get_id(nom_travailleur);
   printf(MSG_CHARGE, nom_travailleur);
}
// interruption ------------------------
void traite_interruption() {
   printf(MSG_INTERRUPTION);
}

//Boucle principale -------------------------------------------------------
-
int main(int argc, char* argv[]) {
   if (argc >= 2 && strcmp("echo", argv[1]) == 0) {
      EchoActif = VRAI;
   }
   Mot buffer;
   while (VRAI) {
      get_id(buffer);
      if (strcmp(buffer, "developpe") == 0) {
         traite_developpe();
         continue;
      }
      if (strcmp(buffer, "embauche") == 0) {
         traite_embauche();
         continue;
      }
      if (strcmp(buffer, "demarche") == 0) {
         traite_demarche();
         continue;
      }
      if (strcmp(buffer, "commande") == 0) {
         traite_commande();
         continue;
      }
      if (strcmp(buffer, "tache") == 0) {
         traite_tache();
         continue;
      }
      if (strcmp(buffer, "progression") == 0) {
         traite_progression();
         continue;
      }
      if (strcmp(buffer, "passe") == 0) {
         passe();
         continue;
      }
      if (strcmp(buffer, "specialites") == 0) {
         traite_specialites();
         continue;
      }
      if (strcmp(buffer, "travailleurs") == 0) {
         traite_travailleurs();
         continue;
      }
```

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

# SPRINT 2 RELEASE

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

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


// Messages emis par les instructions ----------------------------------------
--
#define MSG_DEVELOPPE "## nouvelle specialite \"%s\" ; cout horaire \"%d\"\n"
#define MSG_EMBAUCHE "## nouveau travailleur \"%s\" competent pour la
specialite \"%s\"\n"
#define MSG_DEMARCHE "## nouveau client \"%s\"\n"
#define MSG_COMMANDE "## nouvelle commande \"%s\", par client \"%s\"\n"
#define MSG_TACHE "## la commande \"%s\" requiere la specialite \"%s\"
(nombre d'heures \"%d\")\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"

#define MSG_SPECIALITES "specialites traitees :"

#define MSG_TRAVAILLEURS "la specialite %s peut etre prise en charge par :"
#define MSG_TRAVAILLEURS_TOUS "## consultation des travailleurs competents
pour chaque specialite\n"

#define MSG_CLIENT "le client %s a commande : "
#define MSG_CLIENT_TOUS "## consultation des commandes effectuees par chaque
client\n"

#define MSG_SUPERVISION "## consultation de l'avancement des commandes\n"
#define MSG_CHARGE "## consultation de la charge de travail de \"%s\"\n"
#define MSG_INTERRUPTION "## fin de programme\n"

// Lexemes -------------------------------------------------------------------
```

```
--
#define LGMOT 35
#define NBCHIFFREMAX 5
typedef char Mot[LGMOT + 1];
void get_id(Mot id) {
   scanf("%s", id);
   if (EchoActif) printf(">>echo %s\n", id);
}
int get_int() {
   char buffer[NBCHIFFREMAX + 1];
   // buffer[NBCHIFFREMAX] = '\0'; Apparemment ?a ?vite le warning, ? voir
en pratique ! C'est ce qui a l'air de corrompre le buffer en cas de dep.
   scanf("%s", buffer);
   if (EchoActif) printf(">>echo %s\n", buffer);
   return atoi(buffer);
}

//Donn?es ------------------------------------------------------------------
--
// specialites -------------------------------------------------------------
-
#define MAX_SPECIALITES 10
typedef struct {
   Mot nom;
   int cout_horaire;
} Specialite;

typedef struct {
   Specialite tab_specialites[MAX_SPECIALITES];
   unsigned int nb_specialites;
} Specialites;

// travailleurs ------------------------------------------------------------
-
#define MAX_TRAVAILLEURS 50
typedef struct {
   Mot nom;
   Booleen tags_competences[MAX_SPECIALITES]; //chaque rang du tableau de
bool?en correspond a l'?tat Vrai/faux pour une spe donn?e !
} Travailleur;

typedef struct {
   Travailleur tab_travailleurs[MAX_TRAVAILLEURS];
   unsigned int nb_travailleurs;
} Travailleurs;

// client ------------------------------------------------------------------
--
#define MAX_CLIENTS 10
typedef struct {
   Mot tab_clients[MAX_CLIENTS];
   unsigned int nb_clients;
} Clients;

void liste_travailleurs(Specialites* liste_spe, Travailleurs* liste_tr, Mot
nom_specialite);
void liste_commandes(Clients* liste_cli, Mot nom_client);
unsigned int initialise_embauche(Travailleurs* liste_tr, Mot
nom_travailleur);

// Instructions ------------------------------------------------------------
--
// developpe ---------------------------
void traite_developpe(Specialites* liste_spe) {
```

```
   int cpt_sp = liste_spe->nb_specialites;
   if (cpt_sp < MAX_SPECIALITES) {
      Mot nom_specialite;
      get_id(nom_specialite);
      int cout_horaire = get_int();
      // printf(MSG_DEVELOPPE, nom_specialite, cout_horaire); // retirer  la
release

      liste_spe->tab_specialites[cpt_sp].cout_horaire = cout_horaire;
      strcpy(liste_spe->tab_specialites[cpt_sp].nom, nom_specialite);
      liste_spe->nb_specialites = cpt_sp + 1;
   }
   /*else
      printf("Vous ne pouvez plus creer de nouvelles fonctions.\n"); SI ON
LAISSE CA AFFICHE*/

}
// embauche ----------------------------
void traite_embauche(Travailleurs* liste_tr, Specialites* liste_spe) {
   int cpt_tr = liste_tr->nb_travailleurs;
   unsigned int i, worker_id;
   if (cpt_tr < MAX_TRAVAILLEURS) {
      Mot nom_travailleur;
      get_id(nom_travailleur);
      Mot nom_specialite;
      get_id(nom_specialite);
      // printf(MSG_EMBAUCHE, nom_travailleur, nom_specialite); // retirer 
la release

      Mot temp_mot;
      for (i = 0; i < liste_spe->nb_specialites; ++i) {
         strcpy(temp_mot, liste_spe->tab_specialites[i].nom);
         if (strcmp(nom_specialite, temp_mot) == 0) {
            worker_id = initialise_embauche(liste_tr, nom_travailleur); //
worker_id est soit l'id de l'employ d j d tect , soit -1 pour employ 
non d tect
            if (worker_id == -1) { // si l'employ  n'existe pas...
               liste_tr->tab_travailleurs[cpt_tr].tags_competences[i] = VRAI;
//... alors on mets sa comp tence sur le prochain champ vide, qui lui sera
d di
            }
            else {
               liste_tr->tab_travailleurs[worker_id].tags_competences[i] =
VRAI; // sinon on l'assigne  son champ.
            }
            // printf("La spe %s a ete attribuee au travailleur %s",
nom_specialite, nom_travailleur); //virer release !
            break;
         }
         /*else
            printf("La spe entree n'a pas ete reconnue, le travailleur n'a
pas ete enregistre\n"); // virer release !*/
      }

   }
   /*else
       printf("Vous ne pouvez plus ajouter de nouveau travailleur."); //
virer release !*/
}

// demarche ----------------------------
void traite_demarche(Clients* liste_cli) {
   int cpt_cli = liste_cli->nb_clients;
```

```
    if (cpt_cli < MAX_CLIENTS) {
        Mot nom_client;
        get_id(nom_client);
        // printf(MSG_DEMARCHE, nom_client); // virer release !

        strcpy(liste_cli->tab_clients[cpt_cli], nom_client);
        liste_cli->nb_clients = cpt_cli + 1;
    }
}

// commande ----------------------------
void traite_commande() {
    Mot nom_commande;
    get_id(nom_commande);
    Mot nom_client;
    get_id(nom_client);
    printf(MSG_COMMANDE, nom_commande, nom_client);
}
// tache -------------------------------
void traite_tache() {
    Mot nom_commande;
    get_id(nom_commande);
    Mot nom_specialite;
    get_id(nom_specialite);
    int nombre_heures = get_int();
    printf(MSG_TACHE, nom_commande, nom_specialite, nombre_heures);
}
// progression -------------------------
void traite_progression() {
    Mot nom_commande;
    get_id(nom_commande);
    Mot nom_specialite;
    get_id(nom_specialite);
    int    nombre_heures_sup = get_int();
    printf(MSG_PROGRESSION, nom_commande, nom_specialite, nombre_heures_sup);
}
// specialites -------------------------
void traite_specialites(Specialites* liste_spe) {
    printf(MSG_SPECIALITES);
    int i, cpt_nb_spe = liste_spe->nb_specialites;
    for (i = 0; i < cpt_nb_spe; ++i) {
        if (i < cpt_nb_spe - 1)
            printf(" %s/%d,", liste_spe->tab_specialites[i].nom, liste_spe-
>tab_specialites[i].cout_horaire);
        else
            printf(" %s/%d", liste_spe->tab_specialites[i].nom, liste_spe-
>tab_specialites[i].cout_horaire);
    }
    printf("\n");
}
// travailleurs ------------------------
void traite_travailleurs(Travailleurs* liste_tr, Specialites* liste_spe) {
    Mot nom_specialite;
    get_id(nom_specialite);
    int i, cpt_nb_spe = liste_spe->nb_specialites;
    if (strcmp(nom_specialite, "tous") == 0) { // si le mot cl? est "tous"...
        for (i = 0; i < cpt_nb_spe; ++i) { // on affiche pour chaque sp?
possible ses travailleurs
            strcpy(nom_specialite, liste_spe->tab_specialites[i].nom);
            liste_travailleurs(liste_spe, liste_tr, nom_specialite);
        }
    }
    else {
        liste_travailleurs(liste_spe, liste_tr, nom_specialite);
```

```
    }
}

void liste_travailleurs(Specialites* liste_spe, Travailleurs* liste_tr, Mot
nom_specialite) { // cette fonction liste les travailleurs selon leur sp?
    int i, j, k, set = 0, nb_pr_spe = 0, cpt_nb_spe = liste_spe-
>nb_specialites, cpt_nb_tr = liste_tr->nb_travailleurs;
    printf(MSG_TRAVAILLEURS, nom_specialite);
    for (i = 0; i < cpt_nb_spe; ++i) { // on fouille le tableau de
sp?cialit?s
        if (strcmp(nom_specialite, liste_spe->tab_specialites[i].nom) == 0) {
// si on trouve notre sp?...
            for (j = 0; j < cpt_nb_tr; ++j) { // on fouille celui des
travailleurs
                if (liste_tr->tab_travailleurs[j].tags_competences[i] == VRAI) {
// si l'un des travailleurs a la sp?
                    if (set == 0) { // (si on a d?j? v?rifi? le nombre de
travailleurs lors de cet usage de la fct)
                        for (k = 0; k < MAX_TRAVAILLEURS; ++k) { // sinon, v?rifie
combien de travailleurs ont la sp?
                            if (liste_tr->tab_travailleurs[k].tags_competences[i] ==
VRAI) {
                                ++nb_pr_spe;
                            }
                        }
                        set = 1;
                    }
                    if (j < nb_pr_spe - 1) // enfin on affiche le nom selon le
rang du travailleur avec la sp? par rapport au prochain avec la sp?
                        printf(" %s,", liste_tr->tab_travailleurs[j].nom);
                    else
                        printf(" %s", liste_tr->tab_travailleurs[j].nom);
                }
            }
            printf("\n");
        }
    }
}

// client ------------------------------
void traite_client(Clients* liste_cli) {
    Mot nom_client;
    get_id(nom_client);
    int i, cpt_nb_cli = liste_cli->nb_clients;
    if (strcmp(nom_client, "tous") == 0) { // si le mot cl? est "tous"...
        for (i = 0; i < cpt_nb_cli; ++i) { // on affiche pour chaque client
possible ses commandes
            strcpy(nom_client, liste_cli->tab_clients[i]);
            liste_commandes(liste_cli, nom_client);
        }
    }
    else {
        liste_commandes(liste_cli, nom_client);
    }
}

void liste_commandes(Clients* liste_cli, Mot nom_client) { // pour ce sprint,
affiche juste la liste des clients
    printf(MSG_CLIENT, nom_client);
    printf("\n");
}
// initialise embauche
int initialise_embauche(Travailleurs* liste_tr, Mot nom_travailleur) {
```

```
    unsigned int cpt_tr = liste_tr->nb_travailleurs;
    unsigned int i, j, no_corresp = 0;
    int known_worker = 0;
    for (i = 0; i < MAX_TRAVAILLEURS; ++i) {
        if (strcmp(nom_travailleur, liste_tr->tab_travailleurs[i].nom) == 0)
{ // si le travailleur existe, on d?clare qu'il y a correspondance et donc on
r?cup?re l'id
            no_corresp += 1;
            known_worker = i; // recup?re le num?ro du travailleur d?j?
connu pr changer sa comp?tence
            break;
        }
    }
    if (no_corresp == 0) { // si il n'y aucune correspondance, on initialise
un nouvel employ?
        strcpy(liste_tr->tab_travailleurs[cpt_tr].nom, nom_travailleur);
        liste_tr->nb_travailleurs = cpt_tr + 1;
        for (j = 0; j < MAX_SPECIALITES; ++j) {
            liste_tr->tab_travailleurs[cpt_tr].tags_competences[j] = FAUX;
        }
        known_worker = -1;
    }
    return known_worker; // la fonction retourne l'id de l'employ? ou alors
le fait qu'il n'y a pas d'employ? concern?
}

// supervision -------------------------
void traite_supervision() {
   printf(MSG_SUPERVISION);
}
// charge ------------------------------
void traite_charge() {
   Mot nom_travailleur;
   get_id(nom_travailleur);
   printf(MSG_CHARGE, nom_travailleur);
}
// interruption ------------------------
void traite_interruption() {
   printf(MSG_INTERRUPTION);
}
// passe -------------------------------
void passe() {
    printf(MSG_PASSE); // chaque instruction est execut?e en ligne, l'une
apr?s l'autre; donc si on dit progression
}                  // et que tous les arguments pour progression sont pass?s,
on passe ? l'instruction suviante

//Boucle principale -------------------------------------------------------
-
int main(int argc, char* argv[]) {
   if (argc >= 2 && strcmp("echo", argv[1]) == 0) {
      EchoActif = VRAI;
   }
   Specialites liste_spe;
   Travailleurs liste_tr;
   Clients liste_cli;
   liste_spe.nb_specialites = 0;
   liste_tr.nb_travailleurs = 0;
   liste_cli.nb_clients = 0;
   Mot buffer;

   while (VRAI) {
```

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

# SPRINT 3 RELEASE

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

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


// Messages emis par les instructions ----------------------------------------
--
#define MSG_DEVELOPPE "## nouvelle specialite \"%s\" ; cout horaire \"%d\"\n"
#define MSG_EMBAUCHE "## nouveau travailleur \"%s\" competent pour la
specialite \"%s\"\n"
#define MSG_DEMARCHE "## nouveau client \"%s\"\n"
#define MSG_COMMANDE "## nouvelle commande \"%s\", par client \"%s\"\n"
#define MSG_TACHE "## la commande \"%s\" requiere la specialite \"%s\"
(nombre d'heures \"%d\")\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"
#define MSG_SPECIALITES "specialites traitees :"
#define MSG_TRAVAILLEURS "la specialite %s peut etre prise en charge par :"
// #define MSG_TRAVAILLEURS_TOUS "## consultation des travailleurs competents
pour chaque specialite\n"
#define MSG_CLIENT "le client %s a commande : "
// #define MSG_CLIENT_TOUS "## consultation des commandes effectuees par
chaque client\n"
#define MSG_SUPERVISION "etat des taches pour %s : "
#define MSG_CHARGE "## consultation de la charge de travail de \"%s\"\n"
#define MSG_INTERRUPTION "## fin de programme\n"

// Lexemes -------------------------------------------------------------------
--
#define LGMOT 35
#define NBCHIFFREMAX 5
typedef char Mot[LGMOT + 1];
void get_id(Mot id) {
   scanf("%s", id);
   if (EchoActif) printf(">>echo %s\n", id);
}
int get_int() {
   char buffer[NBCHIFFREMAX + 1];
   // buffer[NBCHIFFREMAX] = '\0'; Apparemment ça évite le warning, à voir en
pratique ! C'est ce qui a l'air de corrompre le buffer en cas de dep.
   scanf("%s", buffer);
   if (EchoActif) printf(">>echo %s\n", buffer);
   return atoi(buffer);
}

//Données -------------------------------------------------------------------
--
// specialites ---------------------------------------------------------------
-
#define MAX_SPECIALITES 10
typedef struct {
   Mot nom;
   int cout_horaire;
```

```
} Specialite;

typedef struct {
   Specialite tab_specialites[MAX_SPECIALITES];
   unsigned int nb_specialites;
} Specialites;

// travailleurs ------------------------------------------------------------
-
#define MAX_TRAVAILLEURS 50
typedef struct {
   Mot nom;
   Booleen tags_competences[MAX_SPECIALITES]; //chaque rang du tableau de
booléen correspond a l'état Vrai/faux pour une spe donnée !
} Travailleur;

typedef struct {
   Travailleur tab_travailleurs[MAX_TRAVAILLEURS];
   unsigned int nb_travailleurs;
} Travailleurs;

// client ------------------------------------------------------------------
--
#define MAX_CLIENTS 10
typedef struct {
   Mot tab_clients[MAX_CLIENTS];
   unsigned int nb_clients;
} Clients;

// commandes ---------------------
#define MAX_COMMANDES 500
typedef struct {
   unsigned int nb_heures_requises;
   unsigned int nb_heures_effectuees;
}Tache;

typedef struct {
   Mot nom;
   unsigned int idx_client;
   Tache taches_par_specialite[MAX_SPECIALITES]; // nb_heures_requises==0 <=>
pas de tache pour cette specialite
}Commande;

typedef struct {
   Commande tab_commandes[MAX_COMMANDES];
   unsigned int nb_commandes;
}Commandes;

void liste_travailleurs(Specialites* liste_spe, Travailleurs* liste_tr, Mot
nom_specialite);
void liste_commandes(Clients* liste_cli, Commandes* liste_com, Mot
nom_client);
int initialise_embauche(Travailleurs* liste_tr, Mot nom_travailleur);

// Instructions ------------------------------------------------------------
--
// developpe --------------------------
void traite_developpe(Specialites* liste_spe) {
   int cpt_sp = liste_spe->nb_specialites;
   if (cpt_sp < MAX_SPECIALITES) {
      Mot nom_specialite;
      get_id(nom_specialite);
      int cout_horaire = get_int();
      // printf(MSG_DEVELOPPE, nom_specialite, cout_horaire); // retirer à la
```

```
release

      liste_spe->tab_specialites[cpt_sp].cout_horaire = cout_horaire;
      strcpy(liste_spe->tab_specialites[cpt_sp].nom, nom_specialite);
      liste_spe->nb_specialites = cpt_sp + 1;
   }
   /*else
      printf("Vous ne pouvez plus creer de nouvelles fonctions.\n");*/

}
// embauche ----------------------------
void traite_embauche(Travailleurs* liste_tr, Specialites* liste_spe) {
   int cpt_tr = liste_tr->nb_travailleurs;
   unsigned int i, worker_id;
   if (cpt_tr < MAX_TRAVAILLEURS) {
      Mot nom_travailleur;
      get_id(nom_travailleur);
      Mot nom_specialite;
      get_id(nom_specialite);
      // printf(MSG_EMBAUCHE, nom_travailleur, nom_specialite); // retirer à
la release

      Mot temp_mot;
      for (i = 0; i < liste_spe->nb_specialites; ++i) {
         strcpy(temp_mot, liste_spe->tab_specialites[i].nom);
         if (strcmp(nom_specialite, temp_mot) == 0) {
            worker_id = initialise_embauche(liste_tr, nom_travailleur); //
worker_id est soit l'id de l'employé déjà détecté, soit -1 pour employé non
détecté
            if (worker_id == -1) { // si l'employé n'existe pas...
               liste_tr->tab_travailleurs[cpt_tr].tags_competences[i] = VRAI;
//... alors on mets sa compétence sur le prochain champ vide, qui lui sera
dédié
            }
            else {
               liste_tr->tab_travailleurs[worker_id].tags_competences[i] =
VRAI; // sinon on l'assigne à son champ.
            }
            // printf("La spe %s a ete attribuee au travailleur %s",
nom_specialite, nom_travailleur); //virer release !
            break;
         }
         /*else
            printf("La spe entree n'a pas ete reconnue, le travailleur n'a
pas ete enregistre\n"); // virer release !*/
      }

   }
   /*else
       printf("Vous ne pouvez plus ajouter de nouveau travailleur."); //
virer release !*/
}

int initialise_embauche(Travailleurs* liste_tr, Mot nom_travailleur) {
   unsigned int cpt_tr = liste_tr->nb_travailleurs;
   unsigned int i, j, no_corresp = 0;
   int known_worker = 0;
   for (i = 0; i < MAX_TRAVAILLEURS; ++i) {
      if (strcmp(nom_travailleur, liste_tr->tab_travailleurs[i].nom) == 0) {
// si le travailleur existe, on déclare qu'il y a correspondance et donc on
récupère l'id
         no_corresp += 1;
         known_worker = i; // recupère le numéro du travailleur déjà connu pr
changer sa compétence
```

```
            break;
         }
      }
      if (no_corresp == 0) { // si il n'y aucune correspondance, on initialise
un nouvel employé
         strcpy(liste_tr->tab_travailleurs[cpt_tr].nom, nom_travailleur);
         liste_tr->nb_travailleurs = cpt_tr + 1;
         for (j = 0; j < MAX_SPECIALITES; ++j) {
            liste_tr->tab_travailleurs[cpt_tr].tags_competences[j] = FAUX;
         }
         known_worker = -1;
      }
      return known_worker; // la fonction retourne l'id de l'employé ou alors le
fait qu'il n'y a pas d'employé concerné
}

// demarche ----------------------------
void traite_demarche(Clients* liste_cli) {
   int cpt_cli = liste_cli->nb_clients;
   if (cpt_cli < MAX_CLIENTS) {
      Mot nom_client;
      get_id(nom_client);
      // printf(MSG_DEMARCHE, nom_client); // virer release !

      strcpy(liste_cli->tab_clients[cpt_cli], nom_client);
      liste_cli->nb_clients = cpt_cli + 1;
   }
}

// commande ----------------------------
void traite_commande(Commandes* liste_com, Clients* liste_cli, Specialites*
liste_spe) {
   int i, j, cpt_com = liste_com->nb_commandes, cpt_spe = liste_spe-
>nb_specialites;
   Mot nom_commande;
   get_id(nom_commande);
   Mot nom_client;
   get_id(nom_client);
   // printf(MSG_COMMANDE, nom_commande, nom_client);

   for (i = 0; i < MAX_CLIENTS; ++i) {
      if (strcmp(liste_cli->tab_clients[i], nom_client) == 0) {
         strcpy(liste_com->tab_commandes[cpt_com].nom, nom_commande);
         liste_com->tab_commandes[cpt_com].idx_client = i;
         for (j = 0; j < MAX_SPECIALITES; ++j) {
            liste_com-
>tab_commandes[cpt_com].taches_par_specialite[j].nb_heures_requises = 0;
            liste_com-
>tab_commandes[cpt_com].taches_par_specialite[j].nb_heures_effectuees = 0;
         }
         liste_com->nb_commandes += 1;
      }
   }
}
// tache ----------------------------
void traite_tache(Commandes* liste_com, Specialites* liste_spe) {
   unsigned int i, j, cpt_spe = liste_spe->nb_specialites, cpt_com =
liste_com->nb_commandes;
   Mot nom_commande;
   get_id(nom_commande);
   Mot nom_specialite;
   get_id(nom_specialite);
   int nombre_heures = get_int();
   // printf(MSG_TACHE, nom_commande, nom_specialite, nombre_heures);
```

```
   for (i = 0; i < cpt_com; ++i) { // on fouille le tableau de commandes
      if (strcmp(nom_commande, liste_com->tab_commandes[i].nom) == 0) { // si
dedans une commande a le nom de commande déclaré
         for (j = 0; j < cpt_spe; ++j) { // on fouille le tableau de spé
            if (strcmp(nom_specialite, liste_spe->tab_specialites[j].nom) ==
0) { // si la spé existe
               liste_com-
>tab_commandes[i].taches_par_specialite[j].nb_heures_requises =
nombre_heures; // alors à l'index de la spé, on déclare tant d'heures
requises pour la tache dans la commande
            }
         }

      }
   }
}
// progression -------------------------
void traite_progression(Commandes* liste_com, Specialites* liste_spe) {
   int i, j, cpt_com = liste_com->nb_commandes, cpt_spe = liste_spe-
>nb_specialites;
   Mot nom_commande;
   get_id(nom_commande);
   Mot nom_specialite;
   get_id(nom_specialite);
   int    nombre_heures_faites = get_int();
   // printf(MSG_PROGRESSION, nom_commande, nom_specialite,
nombre_heures_faites);

   for (i = 0; i < cpt_com; ++i) { // on fouille le tableau de commandes
      if (strcmp(nom_commande, liste_com->tab_commandes[i].nom) == 0) { // si
dedans une commande a le nom de commande déclaré
         for (j = 0; j < cpt_spe; ++j) { // on fouille le tableau de spé
            if (strcmp(nom_specialite, liste_spe->tab_specialites[j].nom) ==
0) { // si la spé existe
               liste_com-
>tab_commandes[i].taches_par_specialite[j].nb_heures_effectuees +=
nombre_heures_faites; // alors à l'index de la spé, on déclare tant d'heures
sup pour la tache dans la commande
            }
         }

      }
   }
}
void passe() {
   //printf(MSG_PASSE);
   /* chaque instruction est executée en ligne, l'une après l'autre; donc si
on dit progression
    et que tous les arguments pour progression sont passés, on passe à
l'instruction suviante */
}
// specialites ------------------------
void traite_specialites(Specialites* liste_spe) {
   printf(MSG_SPECIALITES);
   int i, cpt_nb_spe = liste_spe->nb_specialites;
   for (i = 0; i < cpt_nb_spe; ++i) {
      if (i < cpt_nb_spe - 1)
         printf(" %s/%d,", liste_spe->tab_specialites[i].nom, liste_spe-
>tab_specialites[i].cout_horaire);
      else
         printf(" %s/%d", liste_spe->tab_specialites[i].nom, liste_spe-
>tab_specialites[i].cout_horaire);
   }
```

```
   printf("\n");
}
// travailleurs ------------------------
void traite_travailleurs(Travailleurs* liste_tr, Specialites* liste_spe) {
   Mot nom_specialite;
   get_id(nom_specialite);
   int i, cpt_nb_spe = liste_spe->nb_specialites;
   if (strcmp(nom_specialite, "tous") == 0) { // si le mot clé est "tous"...
      for (i = 0; i < cpt_nb_spe; ++i) { // on affiche pour chaque spé
possible ses travailleurs
         strcpy(nom_specialite, liste_spe->tab_specialites[i].nom);
         liste_travailleurs(liste_spe, liste_tr, nom_specialite);
      }
   }
   else {
      liste_travailleurs(liste_spe, liste_tr, nom_specialite);
   }
}

void liste_travailleurs(Specialites* liste_spe, Travailleurs* liste_tr, Mot
nom_specialite) { // cette fonction liste les travailleurs selon leur spé
   int i, j, k, set = 0, nb_pr_spe = 0, cpt_nb_spe = liste_spe-
>nb_specialites, cpt_nb_tr = liste_tr->nb_travailleurs;
   printf(MSG_TRAVAILLEURS, nom_specialite);
   for (i = 0; i < cpt_nb_spe; ++i) { // on fouille le tableau de spécialités
      if (strcmp(nom_specialite, liste_spe->tab_specialites[i].nom) == 0) {
// si on trouve notre spé...
         for (j = 0; j < cpt_nb_tr; ++j) { // on fouille celui des
travailleurs
            if (liste_tr->tab_travailleurs[j].tags_competences[i] == VRAI) {
// si l'un des travailleurs a la spé
               if (set == 0) { // (si on a déjà vérifié le nombre de
travailleurs lors de cet usage de la fct)
                  for (k = 0; k < MAX_TRAVAILLEURS; ++k) { // sinon, vérifie
combien de travailleurs ont la spé
                     if (liste_tr->tab_travailleurs[k].tags_competences[i] ==
VRAI) {
                        ++nb_pr_spe;
                     }
                  }
                  set = 1;
               }
               if (j < nb_pr_spe - 1) // enfin on affiche le nom selon le
rang du travailleur avec la spé par rapport au prochain avec la spé
                  printf(" %s,", liste_tr->tab_travailleurs[j].nom);
               else
                  printf(" %s", liste_tr->tab_travailleurs[j].nom);
            }
         }
         printf("\n");
      }
   }
}

// client ------------------------------
void traite_client(Clients* liste_cli, Commandes* liste_com) {
   Mot nom_client;
   get_id(nom_client);
   int i, cpt_nb_cli = liste_cli->nb_clients;
   if (strcmp(nom_client, "tous") == 0) { // si le mot clé est "tous"...
      for (i = 0; i < cpt_nb_cli; ++i) { // on affiche pour chaque client
possible ses commandes
         strcpy(nom_client, liste_cli->tab_clients[i]);
         liste_commandes(liste_cli, liste_com, nom_client);
```

```
        }
    }
    else {
        liste_commandes(liste_cli, liste_com, nom_client);
    }
}

void liste_commandes(Clients* liste_cli, Commandes* liste_com, Mot
nom_client) { // pour ce sprint, affiche juste la liste des clients
    unsigned int i, j, set = 0, com_pr_cli = 0, cpt_com = liste_com-
>nb_commandes;
    printf(MSG_CLIENT, nom_client);
    for (i = 0; i < MAX_CLIENTS; ++i) { // on parcours le tableau des clients
        if (strcmp(liste_cli->tab_clients[i], nom_client) == 0) { // on
retrouve le notre
            if (set ==  0) { // si on ne sait pas combien il a de commandes
                for (j = 0; j < cpt_com; ++j) { //... on parcours le tableau des
commandes...
                    if (liste_com->tab_commandes[j].idx_client == i) { // (si l'id
client commande = id client)
                        ++com_pr_cli; // pour les compter !
                    }
                }
                set = 1;
            }
            for (j = 0; j < cpt_com; ++j) { // on parcours le tableau client
                if (liste_com->tab_commandes[j].idx_client == i){ // si une
commande est assignée au client
                    if (j < com_pr_cli - 1) {
                        printf("%s, ", liste_com->tab_commandes[j].nom);
                    }
                    else {
                        printf("%s", liste_com->tab_commandes[j].nom);
                    }
                }
            }
        }
    }
    printf("\n");
}

// supervision -------------------------
void traite_supervision(Commandes* liste_com, Specialites* liste_spe) {
    int i, j, k, nb_h_r_verif = 0, set = 0, tache_pr_spe = 0, nb_h_e = -1,
nb_h_r = -1, cpt_com = liste_com->nb_commandes, cpt_spe = liste_spe-
>nb_specialites;
    int tache_avant = 0;
    Mot nom_specialite;
    for (i = 0; i < cpt_com; ++i) { // on consulte la liste des commandes
        printf(MSG_SUPERVISION, liste_com->tab_commandes[i].nom);
        for (j = 0; j < cpt_spe; ++j) { // on consulte la liste des spé
            nb_h_e = liste_com-
>tab_commandes[i].taches_par_specialite[j].nb_heures_effectuees; // on
récupère le nb heures effectuees pour cette spé dans la commande
            nb_h_r = liste_com-
>tab_commandes[i].taches_par_specialite[j].nb_heures_requises; // on recupère
le nb heures requises pour cette spé dans la commande
            strcpy(nom_specialite, liste_spe->tab_specialites[j].nom); //
nom_specialite devient celui de la spe
            if (set == 0) { // si on a pas déjà compté le nb de spé dans la
commande
                for (k = 0; k < MAX_SPECIALITES; ++k) {
                    nb_h_r_verif = liste_com-
>tab_commandes[i].taches_par_specialite[k].nb_heures_requises;
```

```
                if (nb_h_r_verif > 0 && nb_h_r_verif < 500) {
                    ++tache_pr_spe;
                }
            }
            set = 1;
        }
        if (j == tache_pr_spe - 1 && tache_pr_spe > 0 && nb_h_r > 0) // si
la spé vérifiée a pour id le nb de spé possible - 1 ET que il y a plus de
zéro spé possible ET que le nb heures requises est sup à zero
            printf("%s:%d/%d", nom_specialite, nb_h_e, nb_h_r); // alors
on écrit la spé sans virgule
        else { // sinon
            if (j < tache_pr_spe && tache_pr_spe > 0 && nb_h_r > 0)
            printf("%s:%d/%d, ", nom_specialite, nb_h_e, nb_h_r); // avec
virgule (ou on écrit pas)
        }
    }
    printf("\n");
  }

}
// charge -----------------------------
void traite_charge() {
  Mot nom_travailleur;
  get_id(nom_travailleur);
  printf(MSG_CHARGE, nom_travailleur);
}
// interruption ------------------------
void traite_interruption() {
  printf(MSG_INTERRUPTION);
}

//Boucle principale ------------------------------------------------------
-
int main(int argc, char* argv[]) {
  if (argc >= 2 && strcmp("echo", argv[1]) == 0) {
    EchoActif = VRAI;
  }
  Specialites liste_spe;
  Travailleurs liste_tr;
  Clients liste_cli;
  Commandes liste_com;

  liste_com.nb_commandes = 0;
  liste_spe.nb_specialites = 0;
  liste_tr.nb_travailleurs = 0;
  liste_cli.nb_clients = 0;

  Mot buffer;

  while (VRAI) {
    get_id(buffer);
    if (strcmp(buffer, "developpe") == 0) {
      traite_developpe(&liste_spe);
      continue;
    }
    if (strcmp(buffer, "embauche") == 0) {
      traite_embauche(&liste_tr, &liste_spe);
      continue;
    }
    if (strcmp(buffer, "demarche") == 0) {
      traite_demarche(&liste_cli);
      continue;
    }
```

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

# SPRINT 4-5 RELEASE

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

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


// Messages emis par les instructions ----------------------------------------
--
#define MSG_DEVELOPPE "## nouvelle specialite \"%s\" ; cout horaire \"%d\"\n"
#define MSG_EMBAUCHE "## nouveau travailleur \"%s\" competent pour la
specialite \"%s\"\n"
```

```c
#define MSG_DEMARCHE "## nouveau client \"%s\"\n"
#define MSG_COMMANDE "## nouvelle commande \"%s\", par client \"%s\"\n"
#define MSG_TACHE "## la commande \"%s\" requiere la specialite \"%s\"
(nombre d'heures \"%d\")\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"
#define MSG_SPECIALITES "specialites traitees :"
#define MSG_TRAVAILLEURS "la specialite %s peut etre prise en charge par :"
// #define MSG_TRAVAILLEURS_TOUS "## consultation des travailleurs competents
pour chaque specialite\n"
#define MSG_CLIENT "le client %s a commande : "
// #define MSG_CLIENT_TOUS "## consultation des commandes effectuees par
chaque client\n"
#define MSG_FACTURATION "facturation %s : "
#define MSG_FACTURATION_FIN "facturations : "
#define MSG_SUPERVISION "etat des taches pour %s : "
#define MSG_CHARGE "charge de travail pour %s : "
#define MSG_INTERRUPTION "## fin de programme\n"

// Lexemes ----------------------------------------------------------------
--
#define LGMOT 35
#define NBCHIFFREMAX 5
typedef char Mot[LGMOT + 1];
void get_id(Mot id) {
   scanf("%s", id);
   if (EchoActif) printf(">>echo %s\n", id);
}
int get_int() {
   char buffer[NBCHIFFREMAX + 1];
   // buffer[NBCHIFFREMAX] = '\0'; Apparemment ça évite le warning, à voir en
pratique ! C'est ce qui a l'air de corrompre le buffer en cas de dep.
   scanf("%s", buffer);
   if (EchoActif) printf(">>echo %s\n", buffer);
   return atoi(buffer);
}

//Données ----------------------------------------------------------------
--
// specialites ------------------------------------------------------------
-
#define MAX_SPECIALITES 10
typedef struct {
   Mot nom;
   int cout_horaire;
} Specialite;

typedef struct {
   Specialite tab_specialites[MAX_SPECIALITES];
   unsigned int nb_specialites;
} Specialites;

// travailleurs -----------------------------------------------------------
-
#define MAX_TRAVAILLEURS 50
#define MAX_COMMANDES 500
typedef struct {
   Mot nom;
   Booleen tags_competences[MAX_SPECIALITES]; //chaque rang du tableau de
booléen correspond a l'état Vrai/faux pour une spe donnée !
   long nb_heures_pr_completion;
   unsigned int nb_taches_actives;
} Travailleur;
```

```c
typedef struct {
   Travailleur tab_travailleurs[MAX_TRAVAILLEURS];
   unsigned int nb_travailleurs;
} Travailleurs;

// client ------------------------------------------------------------------
--
#define MAX_CLIENTS 10
typedef struct {
   Mot tab_clients[MAX_CLIENTS];
   unsigned int nb_clients;
} Clients;

// commandes ----------------------
typedef struct {
   unsigned int nb_heures_requises;
   unsigned int nb_heures_effectuees;
   unsigned int idx_travailleur; //chaque travailleur peut etre assigne a une
tache d'une commande
}Tache;

typedef struct {
   Mot nom;
   unsigned int idx_client;
   unsigned int specialites_actives;
   Booleen completion_commande; //VRAI : Terminée, FAUX : en cours
   Tache taches_par_specialite[MAX_SPECIALITES]; // nb_heures_requises==0 <=>
pas de tache pour cette specialite
}Commande;

typedef struct {
   Commande tab_commandes[MAX_COMMANDES];
   unsigned int nb_commandes;
   unsigned int nb_commandes_completees;
}Commandes;

void liste_travailleurs(Specialites* liste_spe, Travailleurs* liste_tr, Mot
nom_specialite);
void liste_commandes(Clients* liste_cli, Commandes* liste_com, Mot
nom_client);
int initialise_embauche(Travailleurs* liste_tr, Mot nom_travailleur);
int facturation(Commandes* liste_com, Specialites* liste_spe, Clients*
liste_cli, Mot nom_commande);

// Instructions ------------------------------------------------------------
--
// developpe ----------------------------
void traite_developpe(Specialites* liste_spe) {
   int cpt_sp = liste_spe->nb_specialites;
   if (cpt_sp < MAX_SPECIALITES) {
      Mot nom_specialite;
      get_id(nom_specialite);
      int cout_horaire = get_int();
      // printf(MSG_DEVELOPPE, nom_specialite, cout_horaire); // retirer à la
release

      liste_spe->tab_specialites[cpt_sp].cout_horaire = cout_horaire;
      strcpy(liste_spe->tab_specialites[cpt_sp].nom, nom_specialite);
      liste_spe->nb_specialites = cpt_sp + 1;
   }
   /*else
      printf("Vous ne pouvez plus creer de nouvelles fonctions.\n");*/
```

```
}
// embauche ----------------------------
void traite_embauche(Travailleurs* liste_tr, Specialites* liste_spe) {
   int cpt_tr = liste_tr->nb_travailleurs;
   unsigned int i, worker_id;
   if (cpt_tr < MAX_TRAVAILLEURS) {
      Mot nom_travailleur;
      get_id(nom_travailleur);
      Mot nom_specialite;
      get_id(nom_specialite);
      // printf(MSG_EMBAUCHE, nom_travailleur, nom_specialite); // retirer à
la release

      Mot temp_mot;
      for (i = 0; i < liste_spe->nb_specialites; ++i) {
         strcpy(temp_mot, liste_spe->tab_specialites[i].nom);
         if (strcmp(nom_specialite, temp_mot) == 0) {
            worker_id = initialise_embauche(liste_tr, nom_travailleur); //
worker_id est soit l'id de l'employé déjà détecté, soit -1 pour employé non
détecté
            if (worker_id == -1) { // si l'employé n'existe pas...
               liste_tr->tab_travailleurs[cpt_tr].tags_competences[i] = VRAI;
//... alors on mets sa compétence sur le prochain champ vide, qui lui sera
dédié
            }
            else {
               liste_tr->tab_travailleurs[worker_id].tags_competences[i] =
VRAI; // sinon on l'assigne à son champ.
            }
            // printf("La spe %s a ete attribuee au travailleur %s",
nom_specialite, nom_travailleur); //virer release !
            break;
         }
         /*else
            printf("La spe entree n'a pas ete reconnue, le travailleur n'a
pas ete enregistre\n"); // virer release !*/
      }

   }
   /*else
       printf("Vous ne pouvez plus ajouter de nouveau travailleur."); //
virer release !*/
}

int initialise_embauche(Travailleurs* liste_tr, Mot nom_travailleur) {
   unsigned int cpt_tr = liste_tr->nb_travailleurs;
   unsigned int i, j, no_corresp = 0;
   int known_worker = 0;
   for (i = 0; i < MAX_TRAVAILLEURS; ++i) {
      if (strcmp(nom_travailleur, liste_tr->tab_travailleurs[i].nom) == 0) {
// si le travailleur existe, on déclare qu'il y a correspondance et donc on
récupère l'id
         no_corresp += 1;
         known_worker = i; // recupère le numéro du travailleur déjà connu pr
changer sa compétence
         break;
      }
   }
   if (no_corresp == 0) { // si il n'y aucune correspondance, on initialise
un nouvel employé
      strcpy(liste_tr->tab_travailleurs[cpt_tr].nom, nom_travailleur);
      liste_tr->tab_travailleurs[cpt_tr].nb_heures_pr_completion = 0;
      liste_tr->tab_travailleurs[cpt_tr].nb_taches_actives = 0;
      if (cpt_tr == 0) {
```

```
            for (i = 0; i < MAX_TRAVAILLEURS; ++i) {
               liste_tr->tab_travailleurs[i].nb_heures_pr_completion = 0;
            }
         }
         liste_tr->nb_travailleurs = cpt_tr + 1;
         for (j = 0; j < MAX_SPECIALITES; ++j) {
            liste_tr->tab_travailleurs[cpt_tr].tags_competences[j] = FAUX;
         }
         known_worker = -1;
      }
      return known_worker; // la fonction retourne l'id de l'employé ou alors le
fait qu'il n'y a pas d'employé concerné
}

// demarche -----------------------------
void traite_demarche(Clients* liste_cli) {
   int cpt_cli = liste_cli->nb_clients;
   if (cpt_cli < MAX_CLIENTS) {
      Mot nom_client;
      get_id(nom_client);
      // printf(MSG_DEMARCHE, nom_client); // virer release !

      strcpy(liste_cli->tab_clients[cpt_cli], nom_client);
      liste_cli->nb_clients = cpt_cli + 1;
   }
}

// commande -----------------------------
void traite_commande(Commandes* liste_com, Clients* liste_cli, Specialites*
liste_spe) {
   int i, j, cpt_com = liste_com->nb_commandes, cpt_spe = liste_spe-
>nb_specialites;
   Mot nom_commande;
   get_id(nom_commande);
   Mot nom_client;
   get_id(nom_client);
   // printf(MSG_COMMANDE, nom_commande, nom_client);

   for (i = 0; i < MAX_CLIENTS; ++i) {
      if (strcmp(liste_cli->tab_clients[i], nom_client) == 0) {
         strcpy(liste_com->tab_commandes[cpt_com].nom, nom_commande);
         liste_com->tab_commandes[cpt_com].idx_client = i;
         for (j = 0; j < MAX_SPECIALITES; ++j) {
            liste_com-
>tab_commandes[cpt_com].taches_par_specialite[j].nb_heures_requises = 0;
            liste_com-
>tab_commandes[cpt_com].taches_par_specialite[j].nb_heures_effectuees = 0;
            liste_com-
>tab_commandes[cpt_com].taches_par_specialite[j].idx_travailleur = -1; // -1
voulant dire que la tache n'a pas été assignée
            liste_com->tab_commandes[cpt_com].completion_commande = FAUX;
            liste_com->tab_commandes[cpt_com].specialites_actives = 0;
         }
         liste_com->nb_commandes += 1;
      }
   }
}
// tache -----------------------------
void traite_tache(Commandes* liste_com, Specialites* liste_spe, Travailleurs*
liste_tr) {
   unsigned int i, j, k, id_tr_rech = 0, cpt_spe = liste_spe->nb_specialites,
cpt_com = liste_com->nb_commandes, cpt_tr = liste_tr->nb_travailleurs;
   Mot nom_commande;
   get_id(nom_commande);
```

```
    Mot nom_specialite;
    get_id(nom_specialite);
    int nombre_heures = get_int();
    // printf(MSG_TACHE, nom_commande, nom_specialite, nombre_heures);

    for (i = 0; i < cpt_com; ++i) { // on fouille le tableau de commandes
        if (strcmp(nom_commande, liste_com->tab_commandes[i].nom) == 0) { // si
dedans une commande a le nom de commande déclaré
            for (j = 0; j < cpt_spe; ++j) { // on fouille le tableau de spé
                if (strcmp(nom_specialite, liste_spe->tab_specialites[j].nom) ==
0) { // si la spé existe
                    liste_com-
>tab_commandes[i].taches_par_specialite[j].nb_heures_requises =
nombre_heures; // alors à l'index de la spé, on déclare tant d'heures
requises pour la tache dans la commande
                    liste_com->tab_commandes[i].specialites_actives += 1;
                    for (k = 0; k < cpt_tr; ++k) { // on parcours le tableau des
travailleurs
                        if (k == 0 && liste_tr-
>tab_travailleurs[k].tags_competences[j] == VRAI) // si c'est la première
valeur du tab et qu'elle a la propriété, elle devient auto le plus petit.
                            id_tr_rech = k;
                        else if (k > 0 && liste_tr-
>tab_travailleurs[k].nb_heures_pr_completion < liste_tr->tab_travailleurs[k -
1].nb_heures_pr_completion
                            && liste_tr->tab_travailleurs[k].tags_competences[j] ==
VRAI) // sinon si c'est une autre valeur, que la précédente est plus grande
qu'elle et qu'elle a le tag, alors elle devient la plus petite
                            id_tr_rech = k;
                    }
                    if (liste_tr->tab_travailleurs[id_tr_rech].tags_competences[j]
== VRAI) { // si la spé correspond avec celle de l'employé
                        liste_com-
>tab_commandes[i].taches_par_specialite[j].idx_travailleur = id_tr_rech; //
alors on dis que l'id du travailleur attribué est le rang de la recherche
                        liste_tr-
>tab_travailleurs[id_tr_rech].nb_heures_pr_completion += nombre_heures; // on
lui ajoute plus d'heures à faire pour arriver à zéro !
                        //break; // et on sort
                    }
                }
            }

        }
    }
}
// progression -------------------------
int traite_progression(Commandes* liste_com, Specialites* liste_spe, Clients*
liste_cli, Travailleurs* liste_tr) {
    int i, j, k, cpt_com = liste_com->nb_commandes, cpt_spe = liste_spe-
>nb_specialites, cpt_tr = liste_tr->nb_travailleurs;
    int program_state = 0; //recupère l'état du programme après traitement de
la progression
    Mot nom_commande;
    get_id(nom_commande);
    Mot nom_specialite;
    get_id(nom_specialite);
    int    nombre_heures_faites = get_int();
    // printf(MSG_PROGRESSION, nom_commande, nom_specialite,
nombre_heures_faites);

    for (i = 0; i < cpt_com; ++i) { // on fouille le tableau de commandes
        if (strcmp(nom_commande, liste_com->tab_commandes[i].nom) == 0) { // si
dedans une commande a le nom de commande déclaré
```

```
            for (j = 0; j < cpt_spe; ++j) { // on fouille le tableau de spé
               if (strcmp(nom_specialite, liste_spe->tab_specialites[j].nom) ==
0) { // si la spé existe
                  liste_com-
>tab_commandes[i].taches_par_specialite[j].nb_heures_effectuees +=
nombre_heures_faites; // alors à l'index de la spé, on déclare tant d'heures
effectuees pour la tache dans la commande
                  for (k = 0; k < cpt_tr; ++k) { // on parcours le tableau des
travailleurs pour lui retirer les heures effectuees
                     if (liste_com-
>tab_commandes[i].taches_par_specialite[j].idx_travailleur == k) {
                        liste_tr->tab_travailleurs[k].nb_heures_pr_completion -=
nombre_heures_faites;
                     }
                  }
                  if (liste_com-
>tab_commandes[i].taches_par_specialite[j].nb_heures_effectuees == liste_com-
>tab_commandes[i].taches_par_specialite[j].nb_heures_requises &&
                     liste_com-
>tab_commandes[i].taches_par_specialite[j].nb_heures_effectuees != 0) { // à
ajouter et si TOUTES les spés de la liste de spé de la commande sont vraies
                     program_state = facturation(liste_com, liste_spe,
liste_cli, nom_commande);
                  }
               }
            }
         }
      }
      return program_state;
}

int facturation(Commandes* liste_com, Specialites* liste_spe, Clients*
liste_cli, Mot nom_commande) {
   // la fonction facturation va émettre une facturation à l'aide de l'état
des taches
   int i, j, k = 0, l = 0, cpt_com = liste_com->nb_commandes, cpt_spe =
liste_spe->nb_specialites, cpt_cli = liste_cli->nb_clients;
   long nb_heures_rest = -1, cout_total = 0, cout_client = 0;
   Mot nom_client;
   Booleen fin = FAUX;
   for (i = 0; i < cpt_com; ++i) { // on parcours le tableau des commandes
      if (strcmp(nom_commande, liste_com->tab_commandes[i].nom) == 0) { // si
la commande existe
         for (j = 0; j < cpt_spe; ++j) { // on parcours les spé de la
commande
            cout_total = liste_spe->tab_specialites[j].cout_horaire *
liste_com->tab_commandes[i].taches_par_specialite[j].nb_heures_effectuees;
            nb_heures_rest = liste_com-
>tab_commandes[i].taches_par_specialite[j].nb_heures_effectuees - liste_com-
>tab_commandes[i].taches_par_specialite[j].nb_heures_requises;
            if (nb_heures_rest == 0 && cout_total > 0) {
               ++l;
            }
         }
         for (j = 0; j < cpt_spe; ++j) { // on parcours les spé de la
commande
            cout_total = liste_spe->tab_specialites[j].cout_horaire *
liste_com->tab_commandes[i].taches_par_specialite[j].nb_heures_effectuees;
            nb_heures_rest = liste_com-
>tab_commandes[i].taches_par_specialite[j].nb_heures_effectuees - liste_com-
>tab_commandes[i].taches_par_specialite[j].nb_heures_requises;
            if (nb_heures_rest == 0 && cout_total > 0 && l == liste_com-
>tab_commandes[i].specialites_actives) {
               if (k == 0) { // si c'est le premier client
```

```
                printf(MSG_FACTURATION, nom_commande);
                printf("%s:%d", liste_spe->tab_specialites[j].nom,
cout_total);
                ++k;
            }
            else if (k == 1) { // sinon si c'est les autres, ils mettent
leur petite virgule avant d'entrer en scène
                printf(", ");
                printf("%s:%d", liste_spe->tab_specialites[j].nom,
cout_total);
            }
         }
      }
      k = 0;
      if (l == liste_com->tab_commandes[i].specialites_actives) { // si l
(l'entier incrémenté lorsqu'une spé d'une commande est complète) est égal au
nombre de spé d'une commande, alors la commande est complète.
         printf("\n");
         liste_com->tab_commandes[i].completion_commande = VRAI;
         ++liste_com->nb_commandes_completees;
      }
   }

}
if (liste_com->nb_commandes_completees == liste_com->nb_commandes &&
liste_com->nb_commandes > 0) { // si à l'issue de la dernière facturation,
toutes les commandes ont été facturées...
   printf(MSG_FACTURATION_FIN);
   for (k = 0; k < cpt_cli; ++k) { // on parcours le tableau des clients
      strcpy(nom_client, liste_cli->tab_clients[k]); // on copie le nom du
client
      cout_client = 0;
      for (i = 0; i < cpt_com; ++i) { // on parcours le tableau des
commandes
         if (liste_com->tab_commandes[i].idx_client == k) { // si le
client concerné par la commande est bien désigné par son index dans le
tableau des clients
            for (j = 0; j < cpt_spe; ++j) { // on parcours les spé de la
commande
               cout_total = liste_spe->tab_specialites[j].cout_horaire *
liste_com->tab_commandes[i].taches_par_specialite[j].nb_heures_effectuees; //
le cout de chaque spé = cout horaire * temps effectué
               cout_client += cout_total; // on l'ajoute au cout déjà
calculé
            }
         }
      }
      if (k == 0)
         printf("%s:%d", nom_client, cout_client);
      if (k > 0) {
         printf(", ");
         printf("%s:%d", nom_client, cout_client);
      }
   }
   printf("\n");
   fin = VRAI;
}
if (fin == VRAI)
   return 1;
if (fin == FAUX)
   return 0;
return 0;
}
```

```
void passe() {
   //printf(MSG_PASSE);
   /* chaque instruction est executée en ligne, l'une après l'autre; donc si
on dit progression
    et que tous les arguments pour progression sont passés, on passe à
l'instruction suviante */
}
// specialites ------------------------
void traite_specialites(Specialites* liste_spe) {
   printf(MSG_SPECIALITES);
   int i, cpt_nb_spe = liste_spe->nb_specialites;
   for (i = 0; i < cpt_nb_spe; ++i) { // on parcours le tableau des
spécialités
      if (i < cpt_nb_spe - 1) // si c'est pas l'avant dernière
         printf(" %s/%d,", liste_spe->tab_specialites[i].nom, liste_spe-
>tab_specialites[i].cout_horaire); // avec virgule
      else
         printf(" %s/%d", liste_spe->tab_specialites[i].nom, liste_spe-
>tab_specialites[i].cout_horaire); // sinon sans
   }
   printf("\n");
}
// travailleurs ------------------------
void traite_travailleurs(Travailleurs* liste_tr, Specialites* liste_spe) {
   Mot nom_specialite;
   get_id(nom_specialite);
   int i, cpt_nb_spe = liste_spe->nb_specialites;
   if (strcmp(nom_specialite, "tous") == 0) { // si le mot clé est "tous"...
      for (i = 0; i < cpt_nb_spe; ++i) { // on affiche pour chaque spé
possible ses travailleurs
         strcpy(nom_specialite, liste_spe->tab_specialites[i].nom);
         liste_travailleurs(liste_spe, liste_tr, nom_specialite);
      }
   }
   else {
      liste_travailleurs(liste_spe, liste_tr, nom_specialite);
   }
}

void liste_travailleurs(Specialites* liste_spe, Travailleurs* liste_tr, Mot
nom_specialite) { // cette fonction liste les travailleurs selon leur spé
   int i, j, k, set = 0, nb_pr_spe = 0, cpt_nb_spe = liste_spe-
>nb_specialites, cpt_nb_tr = liste_tr->nb_travailleurs;
   printf(MSG_TRAVAILLEURS, nom_specialite);
   for (i = 0; i < cpt_nb_spe; ++i) { // on fouille le tableau de spécialités
      if (strcmp(nom_specialite, liste_spe->tab_specialites[i].nom) == 0) {
// si on trouve notre spé...
         for (j = 0; j < cpt_nb_tr; ++j) { // on fouille celui des
travailleurs
            if (liste_tr->tab_travailleurs[j].tags_competences[i] == VRAI) {
// si l'un des travailleurs a la spé
               if (set == 0) { // (si on a déjà vérifié le nombre de
travailleurs lors de cet usage de la fct)
                  for (k = 0; k < MAX_TRAVAILLEURS; ++k) { // sinon, vérifie
combien de travailleurs ont la spé
                     if (liste_tr->tab_travailleurs[k].tags_competences[i] ==
VRAI) {
                        ++nb_pr_spe;
                     }
                  }
                  set = 1;
               }
```

```
                if (j < nb_pr_spe - 1) // enfin on affiche le nom selon le
rang du travailleur avec la spé par rapport au prochain avec la spé
                    printf(" %s,", liste_tr->tab_travailleurs[j].nom);
                else
                    printf(" %s", liste_tr->tab_travailleurs[j].nom);
            }
         }
         printf("\n");
      }
   }
}

// client -----------------------------
void traite_client(Clients* liste_cli, Commandes* liste_com) {
   Mot nom_client;
   get_id(nom_client);
   int i, cpt_nb_cli = liste_cli->nb_clients;
   if (strcmp(nom_client, "tous") == 0) { // si le mot clé est "tous"...
      for (i = 0; i < cpt_nb_cli; ++i) { // on affiche pour chaque client
possible ses commandes
         strcpy(nom_client, liste_cli->tab_clients[i]);
         liste_commandes(liste_cli, liste_com, nom_client);
      }
   }
   else {
      liste_commandes(liste_cli, liste_com, nom_client);
   }
}

void liste_commandes(Clients* liste_cli, Commandes* liste_com, Mot
nom_client) { // pour ce sprint, affiche juste la liste des clients
   unsigned int i, j, set = 0, cpt_cli = liste_cli->nb_clients, com_pr_cli =
0, cpt_com = liste_com->nb_commandes;
   printf(MSG_CLIENT, nom_client);
   for (i = 0; i < cpt_cli; ++i) { // on parcours le tableau des clients
      if (strcmp(liste_cli->tab_clients[i], nom_client) == 0) { // on
retrouve le notre
         if (set == 0) { // si on ne sait pas combien il a de commandes
            for (j = 0; j < cpt_com; ++j) { //... on parcours le tableau des
commandes...
               if (liste_com->tab_commandes[j].idx_client == i) { // (si l'id
client commande = id client)
                  ++com_pr_cli; // pour les compter !
               }
            }
            set = 1;
         }
         for (j = 0; j < cpt_com; ++j) { // on parcours le tableau client
            if (liste_com->tab_commandes[j].idx_client == i) { // si une
commande est assignée au client
               if (j < com_pr_cli) {
                  printf("%s, ", liste_com->tab_commandes[j].nom);
               }
               else {
                  printf("%s", liste_com->tab_commandes[j].nom);
               }
            }
         }
      }
   }
   printf("\n");
}

// supervision ------------------------
```

```
void traite_supervision(Commandes* liste_com, Specialites* liste_spe) {
   int i, j, k, nb_h_r_verif = 0, set = 0, tache_pr_spe = 0, nb_h_e = -1,
nb_h_r = -1, cpt_com = liste_com->nb_commandes, cpt_spe = liste_spe-
>nb_specialites;
   int tache_avant = 0;
   Mot nom_specialite;
   for (i = 0; i < cpt_com; ++i) { // on consulte la liste des commandes
      printf(MSG_SUPERVISION, liste_com->tab_commandes[i].nom);
      for (j = 0; j < cpt_spe; ++j) { // on consulte la liste des spé
         nb_h_e = liste_com-
>tab_commandes[i].taches_par_specialite[j].nb_heures_effectuees; // on
récupère le nb heures effectuees pour cette spé dans la commande
         nb_h_r = liste_com-
>tab_commandes[i].taches_par_specialite[j].nb_heures_requises; // on recupère
le nb heures requises pour cette spé dans la commande
         strcpy(nom_specialite, liste_spe->tab_specialites[j].nom); //
nom_specialite devient celui de la spe
         if (set == 0) { // si on a pas déjà compté le nb de spé dans la
commande
            for (k = 0; k < MAX_SPECIALITES; ++k) {
               nb_h_r_verif = liste_com-
>tab_commandes[i].taches_par_specialite[k].nb_heures_requises;
               if (nb_h_r_verif > 0 && nb_h_r_verif < 500) {
                  ++tache_pr_spe;
               }
            }
            set = 1;
         }
         if (j == tache_pr_spe - 1 && tache_pr_spe > 0 && nb_h_r > 0) // si
la spé vérifiée a pour id le nb de spé possible - 1 ET que il y a plus de
zéro spé possible ET que le nb heures requises est sup à zero
            printf("%s:%d/%d", nom_specialite, nb_h_e, nb_h_r); // alors on
écrit la spé sans virgule
         else { // sinon
            if (j < tache_pr_spe && tache_pr_spe > 0 && nb_h_r > 0)
               printf("%s:%d/%d, ", nom_specialite, nb_h_e, nb_h_r); // avec
virgule (ou on écrit pas)
         }
      }
      printf("\n");
   }

}
// charge ------------------------------
void traite_charge(Travailleurs* liste_tr, Specialites* liste_spe, Commandes*
liste_com) {
   int i, j, k, affiche = 0, nb_heures_rest = 0, cpt_tr = liste_tr-
>nb_travailleurs, cpt_com = liste_com->nb_commandes, cpt_spe = liste_spe-
>nb_specialites;
   Mot nom_travailleur;
   get_id(nom_travailleur);

   for (k = 0; k < cpt_tr; ++k) { // on parcours le tableau des travailleurs
      if (strcmp(nom_travailleur, liste_tr->tab_travailleurs[k].nom) == 0) {
         printf(MSG_CHARGE, nom_travailleur);
         for (j = 0; j < cpt_com; ++j) { // on parcours le tableau des
commandes déclarées
            for (i = 0; i < cpt_spe; ++i) { // on parcours le tableau des spe
déclarées
               if (liste_com-
>tab_commandes[j].taches_par_specialite[i].idx_travailleur == k) { // si l'id
du travailleur correspond à celui de la spe de la tache de la commande
                  nb_heures_rest = liste_com-
>tab_commandes[j].taches_par_specialite[i].nb_heures_requises - liste_com-
```

```
>tab_commandes[j].taches_par_specialite[i].nb_heures_effectuees;
                if (nb_heures_rest == 0) // si une tache est déjà remplie
                    continue; // on passe à la prochaine itération de la
boucle (donc rien ne s'affiche)
                if (affiche == 0) { // si c'est la première fois qu'on
affiche, alors on mets sans virgule
                    printf("%s/%s/%dheure(s)", liste_com-
>tab_commandes[j].nom, liste_spe->tab_specialites[i].nom, nb_heures_rest); //
sans virgule
                    affiche += 1;
                    continue;
                }
                else if (affiche > 0) // sinon les valeurs suivantes
préparent une virgule avant de s'afficher
                    printf(", %s/%s/%dheure(s)", liste_com-
>tab_commandes[j].nom, liste_spe->tab_specialites[i].nom, nb_heures_rest); //
sinon sans virgule
            }
        }
    }
    printf("\n");
  }
}
// interruption ------------------------
void traite_interruption() {
  printf(MSG_INTERRUPTION);
}

//Boucle principale ---------------------------------------------------------
-
int main(int argc, char* argv[]) {
  if (argc >= 2 && strcmp("echo", argv[1]) == 0) {
    EchoActif = VRAI;
  }
  Specialites liste_spe;
  Travailleurs liste_tr;
  Clients liste_cli;
  Commandes liste_com;

  liste_com.nb_commandes = 0;
  liste_com.nb_commandes_completees = 0;
  liste_spe.nb_specialites = 0;
  liste_tr.nb_travailleurs = 0;
  liste_cli.nb_clients = 0;

  Mot buffer;
  int program_state = 0; //program_state prend le statut d'exécution du
programme; 0 et le programme continue, 1 et le programme s'arrête
immédiatement.

  while (VRAI) {
    get_id(buffer);
    if (strcmp(buffer, "developpe") == 0) {
      traite_developpe(&liste_spe);
      continue;
    }
    if (strcmp(buffer, "embauche") == 0) {
      traite_embauche(&liste_tr, &liste_spe);
      continue;
    }
    if (strcmp(buffer, "demarche") == 0) {
      traite_demarche(&liste_cli);
      continue;
```

```
        }
        if (strcmp(buffer, "commande") == 0) {
            traite_commande(&liste_com, &liste_cli, &liste_spe);
            continue;
        }
        if (strcmp(buffer, "tache") == 0) {
            traite_tache(&liste_com, &liste_spe, &liste_tr);
            continue;
        }
        if (strcmp(buffer, "progression") == 0) {
            program_state = traite_progression(&liste_com, &liste_spe,
&liste_cli, &liste_tr);
            if (program_state == 0) {
                continue;
            }
            if (program_state == 1) {
                break;
            }
        }
        if (strcmp(buffer, "passe") == 0) {
            passe();
            continue;
        }
        if (strcmp(buffer, "specialites") == 0) {
            traite_specialites(&liste_spe);
            continue;
        }
        if (strcmp(buffer, "travailleurs") == 0) {
            traite_travailleurs(&liste_tr, &liste_spe);
            continue;
        }
        if (strcmp(buffer, "client") == 0) {
            traite_client(&liste_cli, &liste_com);
            continue;
        }
        if (strcmp(buffer, "supervision") == 0) {
            traite_supervision(&liste_com, &liste_spe);
            continue;
        }
        if (strcmp(buffer, "charge") == 0) {
            traite_charge(&liste_tr, &liste_spe, &liste_com);
            continue;
        }
        if (strcmp(buffer, "interruption") == 0) {
            traite_interruption();
            break;
        }
        printf("!!! instruction inconnue >%s< !!!\n", buffer);
    }
    return 0;
}
```

# SPRINT 6 RELEASE

```
/*
IAP Projet periode A : Conception d'un système de traitement de commandes
Auteurs : Mehdi ZAOUI et Anthony ZAKANI
   dans le cadre d'une formation de l'Université de Paris
Sur la base d'un code source issu de l'équipe pédagogique de l'IUT de Paris-
Rives de Seine, faisant partie de l'Université de Paris
Dernière mise à jour : lundi 2 novembre 2020 à 14h19
*/
```

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

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


// Messages emis par les instructions ------------------------------------
--
#define MSG_DEVELOPPE "## nouvelle specialite \"%s\" ; cout horaire \"%d\"\n"
#define MSG_EMBAUCHE "## nouveau travailleur \"%s\" competent pour la
specialite \"%s\"\n"
#define MSG_DEMARCHE "## nouveau client \"%s\"\n"
#define MSG_COMMANDE "## nouvelle commande \"%s\", par client \"%s\"\n"
#define MSG_TACHE "## la commande \"%s\" requiere la specialite \"%s\"
(nombre d'heures \"%d\")\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"
#define MSG_SPECIALITES "specialites traitees :"
#define MSG_TRAVAILLEURS "la specialite %s peut etre prise en charge par :"
// #define MSG_TRAVAILLEURS_TOUS "## consultation des travailleurs competents
pour chaque specialite\n"
#define MSG_CLIENT "le client %s a commande : "
// #define MSG_CLIENT_TOUS "## consultation des commandes effectuees par
chaque client\n"
#define MSG_FACTURATION "facturation %s : "
#define MSG_FACTURATION_FIN "facturations : "
#define MSG_SUPERVISION "etat des taches pour %s : "
#define MSG_CHARGE "charge de travail pour %s : "
#define MSG_INTERRUPTION "## fin de programme\n"

// Lexemes ---------------------------------------------------------------
--
#define LGMOT 35
#define NBCHIFFREMAX 5
typedef char Mot[LGMOT + 1];
void get_id(Mot id) {
   scanf("%s", id);
   if (EchoActif) printf(">>echo %s\n", id);
}
int get_int() {
   char buffer[NBCHIFFREMAX + 1];
   // buffer[NBCHIFFREMAX] = '\0'; Apparemment ça évite le warning, à voir en
pratique ! C'est ce qui a l'air de corrompre le buffer en cas de dep.
   scanf("%s", buffer);
   if (EchoActif) printf(">>echo %s\n", buffer);
   return atoi(buffer);
}

//Données ---------------------------------------------------------------
--
// specialites -----------------------------------------------------------
-
#define MAX_SPECIALITES 10
typedef struct {
   Mot nom;
   int cout_horaire;
} Specialite;
```

```c
typedef struct {
   Specialite tab_specialites[MAX_SPECIALITES];
   unsigned int nb_specialites;
} Specialites;

// travailleurs ------------------------------------------------------------
-
#define MAX_TRAVAILLEURS 50
#define MAX_COMMANDES 500
typedef struct {
   Mot nom;
   Booleen tags_competences[MAX_SPECIALITES]; //chaque rang du tableau de
booléen correspond a l'état Vrai/faux pour une spe donnée !
   long nb_heures_pr_completion;
   unsigned int nb_taches_actives;
} Travailleur;

typedef struct {
   Travailleur tab_travailleurs[MAX_TRAVAILLEURS];
   unsigned int nb_travailleurs;
} Travailleurs;

// client ------------------------------------------------------------------
--
#define MAX_CLIENTS 10
typedef struct {
   Mot tab_clients[MAX_CLIENTS];
   unsigned int nb_clients;
} Clients;

// commandes ----------------------
typedef struct {
   unsigned int nb_heures_requises;
   unsigned int nb_heures_effectuees;
   unsigned int idx_travailleur; //chaque travailleur peut etre assigne a une
tache d'une commande
}Tache;

typedef struct {
   Mot nom;
   unsigned int idx_client;
   unsigned int specialites_actives;
   Booleen completion_commande; //VRAI : Terminée, FAUX : en cours
   Tache taches_par_specialite[MAX_SPECIALITES]; // nb_heures_requises==0 <=>
pas de tache pour cette specialite
}Commande;

typedef struct {
   Commande tab_commandes[MAX_COMMANDES];
   unsigned int nb_commandes;
   unsigned int nb_commandes_completees;
}Commandes;

// Prototypes --------------------------------------------------------------
---
// instructions ----------------------
void traite_developpe(Specialites* liste_spe);
void traite_embauche(Travailleurs* liste_tr, Specialites* liste_spe);
int initialise_embauche(Travailleurs* liste_tr, Mot nom_travailleur);
void traite_demarche(Clients* liste_cli);
void traite_commande(Commandes* liste_com, Clients* liste_cli, Specialites*
liste_spe);
```

```
void traite_tache(Commandes* liste_com, Specialites* liste_spe, Travailleurs*
liste_tr);
int traite_progression(Commandes* liste_com, Specialites* liste_spe, Clients*
liste_cli, Travailleurs* liste_tr, int* idx_tache, int* idx_com);
void traite_supervision(Commandes* liste_com, Specialites* liste_spe);
void traite_charge(Travailleurs* liste_tr, Specialites* liste_spe, Commandes*
liste_com);

// fonctions subsidiaires --------------
void liste_travailleurs(Specialites* liste_spe, Travailleurs* liste_tr, Mot
nom_specialite);
void liste_commandes(Clients* liste_cli, Commandes* liste_com, Mot
nom_client);
int initialise_embauche(Travailleurs* liste_tr, Mot nom_travailleur);
int facturation(Commandes* liste_com, Specialites* liste_spe, Clients*
liste_cli, Mot nom_commande);

// Instructions ------------------------------------------------------------
--
// developpe ----------------------------
void traite_developpe(Specialites* liste_spe) {
   /*
   [brief] La fonction traite_developpe a pour but d'enregistrer la
spécialité entrée par l'utilisateur dans la liste
   de spécialités possibles pour le fonctionnement du système
   liste_spe [in-out] : on recupère la structure liste_spe de type
Specialites, contenant le tableau des spécialités ainsi qu'un compteur de
spécialités
                  Cette structure sera celle qu'on modifiera au cours du
traitement de la spécialité entrée par l'utilisateur.
   */
   int cpt_sp = liste_spe->nb_specialites;
   if (cpt_sp < MAX_SPECIALITES) {
      Mot nom_specialite;
      get_id(nom_specialite);
      int cout_horaire = get_int();
      // printf(MSG_DEVELOPPE, nom_specialite, cout_horaire); // retirer à la
release

      liste_spe->tab_specialites[cpt_sp].cout_horaire = cout_horaire;
      strcpy(liste_spe->tab_specialites[cpt_sp].nom, nom_specialite);
      liste_spe->nb_specialites = cpt_sp + 1;
   }
   /*else
      printf("Vous ne pouvez plus creer de nouvelles fonctions.\n");*/

}
// embauche ----------------------------
void traite_embauche(Travailleurs* liste_tr, Specialites* liste_spe) {
   /*
   [brief] La fonction traite_embauche a pour but d'enregistrer le
travailleur entré par l'utilisateur dans la liste
   de travailleurs possibles pour le fonctionnement du système ET en lui
assignant la spécialité spécifée par l'utilisateur, sous réserve qu'elle
appartienne
   à la liste des spécialites déjà déclarées.
      - La vérification d'existence de spécialité se fait dans cette fonction
tandis que l'enregistrement du travailleur fait l'objet d'une autre fonction
détaillée ci-dessous : initialise_embauche().
   liste_tr [in-out] : on recupère la structure liste_tr de type
Travailleurs, contenant le tableau des travailleurs ainsi qu'un compteur du
nombre de travailleurs
                  Cette structure sera celle qu'on modifiera au cours du
traitement du travailleur entré par l'utilisateur.
```

```
                On l'utilisera notamment pour ajouter un nouveau
travailleur à l'index des travailleurs, en lui assignant, au rang de la
compétence ajoutée, un tag_competence VRAI;
                De plus, cette fonction, de par son fonctionnement, ajoute
automatiquement à un travailleur la spécialité spécifiée, sous réserve que le
travailleur et la spécialité existent déjà.
   liste_spe [in] : on recupère la structure liste_spe de type Specialites,
contenant le tableau des spécialités ainsi qu'un compteur de spécialités.
                On l'utilisera cette fois uniquement pour vérifier
l'existence de la spécialité spécifiée par l'utilisateur, ainsi que pour
récupérer l'index de celle-ci pour l'assigner au travailleur enregistré.
   */
   int cpt_tr = liste_tr->nb_travailleurs;
   unsigned int i, worker_id;
   if (cpt_tr < MAX_TRAVAILLEURS) {
      Mot nom_travailleur;
      get_id(nom_travailleur);
      Mot nom_specialite;
      get_id(nom_specialite);
      // printf(MSG_EMBAUCHE, nom_travailleur, nom_specialite); // retirer à
la release

      Mot temp_mot;
      for (i = 0; i < liste_spe->nb_specialites; ++i) {
         strcpy(temp_mot, liste_spe->tab_specialites[i].nom);
         if (strcmp(nom_specialite, temp_mot) == 0) {
            worker_id = initialise_embauche(liste_tr, nom_travailleur); //
worker_id est soit l'id de l'employé déjà détecté, soit -1 pour employé non
détecté
            if (worker_id == -1) { // si l'employé n'existe pas...
               liste_tr->tab_travailleurs[cpt_tr].tags_competences[i] = VRAI;
//... alors on mets sa compétence sur le prochain champ vide, qui lui sera
dédié
            }
            else {
               liste_tr->tab_travailleurs[worker_id].tags_competences[i] =
VRAI; // sinon on l'assigne à son champ.
            }
            // printf("La spe %s a ete attribuee au travailleur %s",
nom_specialite, nom_travailleur); //virer release !
            break;
         }
         /*else
            printf("La spe entree n'a pas ete reconnue, le travailleur n'a
pas ete enregistre\n"); // virer release !*/
      }

   }
   /*else
       printf("Vous ne pouvez plus ajouter de nouveau travailleur."); //
virer release !*/
}

int initialise_embauche(Travailleurs* liste_tr, Mot nom_travailleur) {
   /*
   [brief] La fonction initialise_embauche a pour but d'enregistrer le
travailleur nom_travailleur entré par l'utilisateur dans la liste
   de travaillleurs puis de retourner à la fonction traite_embauche l'état
d'enregistrement dudit travailleur.
   liste_tr [in-out] : on recupère la structure liste_tr de type
Travailleurs, contenant le tableau des travailleurs ainsi qu'un compteur du
nombre de travailleurs
                Cette structure sera celle qu'on modifiera au cours du
traitement du travailleur entré par l'utilisateur.
```

```
                On l'utilisera ici seulement pour ajouter un nouveau
travailleur à l'index des travailleurs (ou alors pour vérifier si un
travailleur du même nom existe déjà)
   nom_travailleur [in] : le Mot nom_travailleur provient de la fonction
traite_embauche, il s'agit du nom de travailleur entré par l'utilisateur lors
de l'appel de l'instruction
                    embauche. Il sera utilisé pour vérifier la présence d'un
travailleur avec le même nom dans la liste des travailleurs liste_tr et pour
en enregistrer un nouveau
                    si aucun travaillerur n'existe pour ce nom.
   known-worker [out] : cet entier known-worker ("travailleur connu" en
français) prend deux valeurs possibles : soit, le travailleur existe déjà et
known-worker prend la valeur de son index dans la liste_tr des travailleurs,
                                                              soit, le
travailleur n'existe pas et known-worker prend la valeur -1, qui montre qu'un
ajout de travailleur a été effectué et que
                                                              l'enregistrement
a été effectué
   */
   unsigned int cpt_tr = liste_tr->nb_travailleurs;
   unsigned int i, j, no_corresp = 0;
   int known_worker = 0;
   for (i = 0; i < MAX_TRAVAILLEURS; ++i) {
      if (strcmp(nom_travailleur, liste_tr->tab_travailleurs[i].nom) == 0) {
// si le travailleur existe, on déclare qu'il y a correspondance et donc on
récupère l'id
         no_corresp += 1;
         known_worker = i; // recupère le numéro du travailleur déjà connu pr
changer sa compétence
         break;
      }
   }
   if (no_corresp == 0) { // si il n'y aucune correspondance, on initialise
un nouvel employé
      strcpy(liste_tr->tab_travailleurs[cpt_tr].nom, nom_travailleur);
      liste_tr->tab_travailleurs[cpt_tr].nb_heures_pr_completion = 0;
      liste_tr->tab_travailleurs[cpt_tr].nb_taches_actives = 0;
      if (cpt_tr == 0) {
         for (i = 0; i < MAX_TRAVAILLEURS; ++i) {
            liste_tr->tab_travailleurs[i].nb_heures_pr_completion = 0;
         }
      }
      liste_tr->nb_travailleurs = cpt_tr + 1;
      for (j = 0; j < MAX_SPECIALITES; ++j) {
         liste_tr->tab_travailleurs[cpt_tr].tags_competences[j] = FAUX;
      }
      known_worker = -1;
   }
   return known_worker; // la fonction retourne l'id de l'employé ou alors le
fait qu'il n'y a pas d'employé concerné
}

// demarche ----------------------------
void traite_demarche(Clients* liste_cli) {
   /*
   [brief] La fonction traite_demarche a pour but d'enregistrer le client
entrée par l'utilisateur dans la liste
   de clients possibles pour le fonctionnement du système
   liste_cli [in-out] : on recupère la structure liste_cli de type Clients,
contenant le tableau des clients ainsi qu'un compteur de clients
                  Cette structure sera celle qu'on modifiera au cours du
traitement du client entré par l'utilisateur.
   */
   int cpt_cli = liste_cli->nb_clients;
```

```
    if (cpt_cli < MAX_CLIENTS) {
        Mot nom_client;
        get_id(nom_client);
        // printf(MSG_DEMARCHE, nom_client); // virer release !

        strcpy(liste_cli->tab_clients[cpt_cli], nom_client);
        liste_cli->nb_clients = cpt_cli + 1;
    }
}

// commande ----------------------------
void traite_commande(Commandes* liste_com, Clients* liste_cli, Specialites*
liste_spe) {
    /*
    [brief] La fonction traite_commande a pour but d'enregistrer la commande
entrée par l'utilisateur dans la liste
    de commandes possibles pour le fonctionnement du système, en lui assignant
à la fois un client concerné et des tâches en fonction de chaque spécialité
possible
    liste_com [in-out] : on recupère (en pointant vers elle) la structure
liste_com de type Commandes, contenant le tableau des commandes ainsi qu'un
compteur de commandes et un compteur de commandes complétées.
                    Cette structure sera celle qu'on modifiera au cours du
traitement de la commande entrée par l'utilisateur.
    liste_cli [in] : on recupère la structure liste_cli de type Clients,
contenant le tableau des clients ainsi qu'un compteur de clients
                    Cette structure ne sera pas modifiée au cours du
traitement, elle servira à vérifier l'existence du client concerné.
    liste_spe [in] : on recupère la structure liste_spe de type Specialites,
contenant le tableau des spécialités ainsi qu'un compteur du nombre de
spécialités
                     Cette structure ne sera pas modfiée.
    */
    int i, j, cpt_com = liste_com->nb_commandes, cpt_spe = liste_spe-
>nb_specialites;
    Mot nom_commande;
    get_id(nom_commande);
    Mot nom_client;
    get_id(nom_client);
    // printf(MSG_COMMANDE, nom_commande, nom_client);

    for (i = 0; i < MAX_CLIENTS; ++i) {
        if (strcmp(liste_cli->tab_clients[i], nom_client) == 0) {
            strcpy(liste_com->tab_commandes[cpt_com].nom, nom_commande);
            liste_com->tab_commandes[cpt_com].idx_client = i;
            for (j = 0; j < MAX_SPECIALITES; ++j) {
                liste_com-
>tab_commandes[cpt_com].taches_par_specialite[j].nb_heures_requises = 0;
                liste_com-
>tab_commandes[cpt_com].taches_par_specialite[j].nb_heures_effectuees = 0;
                liste_com-
>tab_commandes[cpt_com].taches_par_specialite[j].idx_travailleur = -1; // -1
voulant dire que la tache n'a pas été assignée
                liste_com->tab_commandes[cpt_com].completion_commande = FAUX;
                liste_com->tab_commandes[cpt_com].specialites_actives = 0;
            }
            liste_com->nb_commandes += 1;
        }
    }
}
// tache -------------------------------
void traite_tache(Commandes* liste_com, Specialites* liste_spe, Travailleurs*
liste_tr) {
    /*
```

```
   [brief] La fonction traite_tache a pour but d'enregistrer la tâche entrée
par l'utilisateur dans la commande qu'il aura spécifié, en lui assignant
   pour une spécialité spécifiée, un travailleur responsable ainsi qu'un état
de progression de ladite tâche.
   liste_com [in-out] : on recupère (en pointant vers elle) la structure
liste_com de type Commandes, contenant le tableau des commandes ainsi qu'un
compteur de commandes et un compteur de commandes complétées.
                    Cette structure sera celle qu'on modifiera au cours du
traitement de la commande entrée par l'utilisateur.
                    On y renseignera notamment les propriétés de la tâche
(idx_travailleur, nb_heures_requises... entre autres).
   liste_tr [in] : on recupère la structure liste_tr de type Travailleurs,
contenant le tableau des travailleurs ainsi qu'un compteur de travailleurs
                    On pourra ainsi y déclarer pour le travailleur le nombre
d'heures supplémentaires lui ayant été assigné en prenant cette tâche.
   liste_spe [in] : on recupère la structure liste_spe de type Specialites,
contenant le tableau des spécialités ainsi qu'un compteur du nombre de
spécialités
                     Cette structure ne sera pas modfiée.
   */
   int i, j, k, id_tr_rech = -1, heuremin = -1, cpt_spe = liste_spe-
>nb_specialites, cpt_com = liste_com->nb_commandes, cpt_tr = liste_tr-
>nb_travailleurs;
   Mot nom_commande;
   get_id(nom_commande);
   Mot nom_specialite;
   get_id(nom_specialite);
   int nombre_heures = get_int();
   // printf(MSG_TACHE, nom_commande, nom_specialite, nombre_heures);

   for (i = 0; i < cpt_com; ++i) { // on fouille le tableau de commandes
      if (strcmp(nom_commande, liste_com->tab_commandes[i].nom) == 0) { // si
dedans une commande a le nom de commande déclaré
         for (j = 0; j < cpt_spe; ++j) { // on fouille le tableau de spé
            if (strcmp(nom_specialite, liste_spe->tab_specialites[j].nom) ==
0) { // si la spé existe
               liste_com-
>tab_commandes[i].taches_par_specialite[j].nb_heures_requises =
nombre_heures; // alors à l'index de la spé, on déclare tant d'heures
requises pour la tache dans la commande
               liste_com->tab_commandes[i].specialites_actives += 1;
               for (k = 0; k < cpt_tr; ++k) { // on parcours le tableau des
travailleurs
                  if (liste_tr->tab_travailleurs[k].tags_competences[j] ==
VRAI && id_tr_rech == -1) {// si c'est la première valeur du tab et qu'elle a
la propriété, elle devient auto le plus petit.
                     id_tr_rech = k;
                     heuremin = liste_tr-
>tab_travailleurs[k].nb_heures_pr_completion;
                     // printf("voie a : %d, %d", id_tr_rech, heuremin);
                  }
                  else if (liste_tr->tab_travailleurs[k].tags_competences[j]
== VRAI && id_tr_rech >= 0 && id_tr_rech < cpt_tr && heuremin > 0 && heuremin
> liste_tr->tab_travailleurs[k].nb_heures_pr_completion) { // sinon si c'est
une autre valeur, que la précédente est plus grande qu'elle et qu'elle a le
tag, alors elle devient la plus petite
                     id_tr_rech = k;
                     heuremin = liste_tr-
>tab_travailleurs[k].nb_heures_pr_completion;
                     // printf("voie b : %d, %d", id_tr_rech, heuremin);
                  }
                  else if (liste_tr->tab_travailleurs[k].tags_competences[j]
== VRAI && id_tr_rech >= 0 && id_tr_rech < cpt_tr && heuremin > liste_tr-
>tab_travailleurs[k].nb_heures_pr_completion) {
```

```
                        id_tr_rech = k;
                        // printf("voie c : %d, %d", id_tr_rech, heuremin);
                    }
                }
                if (liste_tr->tab_travailleurs[id_tr_rech].tags_competences[j]
== VRAI) { // si la spé correspond avec celle de l'employé
                    liste_com-
>tab_commandes[i].taches_par_specialite[j].idx_travailleur = id_tr_rech; //
alors on dis que l'id du travailleur attribué est le rang de la recherche
                    liste_tr-
>tab_travailleurs[id_tr_rech].nb_heures_pr_completion += nombre_heures; // on
lui ajoute plus d'heures à faire pour arriver à zéro !
                }
            }
        }

    }
}
}
// progression -------------------------
int traite_progression(Commandes* liste_com, Specialites* liste_spe, Clients*
liste_cli, Travailleurs* liste_tr, int* idx_tache, int* idx_com) {
    /*
    [brief] La fonction traite_progression a pour but de mettre à jour la
progression d'une tâche, elle mets en oeuvre plusieurs des listes de chaque
type
    pour permettre de changer toutes les données nécessaires au traitement
d'une tâche d'une commande.
    liste_com [in-out] : on recupère (en pointant vers elle) la structure
liste_com de type Commandes, contenant le tableau des commandes ainsi qu'un
compteur de commandes et un compteur de commandes complétées.
                    Cette structure sera celle qu'on modifiera au cours du
traitement de la commande entrée par l'utilisateur.
    liste_spe [in] : on recupère la structure liste_spe de type Specialites,
contenant le tableau des spécialités ainsi qu'un compteur du nombre de
spécialités
                    Cette structure ne sera pas modfiée.
    liste_cli [in] : on recupère la structure liste_cli de type Clients,
contenant le tableau des clients ainsi qu'un compteur de clients
                    Cette structure ne sera pas modifiée au cours du
traitement, elle servira à vérifier l'existence du client concerné.
    liste_tr [in-out] : on recupère la structure liste_tr de type
Travailleurs, contenant le tableau des travailleurs ainsi qu'un compteur de
travailleurs
                    On pourra ainsi y modifier pour le travailleur le nombre
d'heures lui ayant été assigné.
    idx_tache [in-out] : cet entier correspond à l'index de la tâche modifiée
lors du traitement de la progression
    idx_com [in-out] : cet entier correspond à l'index de la commande modfiée
lors du traitement de la progression
    program_state [out] : cet entier peut avoir deux valeurs possibles à la
fin du traitement de la progression :
                    soit, aucune facturation n'a été effectuée ou alors une
facturation non finale alors program_state n'est pas déclenché : valeur 0
                    soit, une facturation finale a été émise donc le
programme doit s'arrêter, program_state est égal à 1 (il est déclenché).
    */
    int i, j, k, id_ancien_tr = 0, id_tr_rech = 0, cpt_com = liste_com-
>nb_commandes, cpt_spe = liste_spe->nb_specialites, cpt_tr = liste_tr-
>nb_travailleurs;
    int program_state = 0; //recupère l'état du programme après traitement de
la progression
    Mot nom_commande;
    get_id(nom_commande);
```

```
    Mot nom_specialite;
    get_id(nom_specialite);
    int    nombre_heures_faites = get_int();
    // printf(MSG_PROGRESSION, nom_commande, nom_specialite,
nombre_heures_faites);

        for (i = 0; i < cpt_com; ++i) { // on fouille le tableau de commandes
            if (strcmp(nom_commande, liste_com->tab_commandes[i].nom) == 0) { //
si dedans une commande a le nom de commande déclaré
                for (j = 0; j < cpt_spe; ++j) { // on fouille le tableau de spé
                    if (strcmp(nom_specialite, liste_spe->tab_specialites[j].nom)
== 0) { // si la spé existe
                        liste_com-
>tab_commandes[i].taches_par_specialite[j].nb_heures_effectuees +=
nombre_heures_faites; // alors à l'index de la spé, on déclare tant d'heures
effectuees pour la tache dans la commande
                        for (k = 0; k < cpt_tr; ++k) { // on parcours le tableau
des travailleurs pour lui retirer les heures effectuees
                            if (liste_com-
>tab_commandes[i].taches_par_specialite[j].idx_travailleur == k) {
                                liste_tr->tab_travailleurs[k].nb_heures_pr_completion
-= nombre_heures_faites;
                                *idx_com = i;
                                *idx_tache = j;
                            }
                        }
                        if (liste_com-
>tab_commandes[i].taches_par_specialite[j].nb_heures_effectuees == liste_com-
>tab_commandes[i].taches_par_specialite[j].nb_heures_requises &&
                            liste_com-
>tab_commandes[i].taches_par_specialite[j].nb_heures_effectuees != 0) { // à
ajouter et si TOUTES les spés de la liste de spé de la commande sont vraies
                            program_state = facturation(liste_com, liste_spe,
liste_cli, nom_commande);
                        }
                    }
                }
            }
        }
        return program_state;
}

int facturation(Commandes* liste_com, Specialites* liste_spe, Clients*
liste_cli, Mot nom_commande) {
    /*
    [brief] La fonction facturation a pour but d'émettre une facturation
correspondant à une commande complétée (à l'aide de l'état de chaque tâche)
ou alors
    une facturation finale correspondant à celles de toutes les commandes (et
donc la fin du programme).
    liste_com [in-out] : on recupère (en pointant vers elle) la structure
liste_com de type Commandes, contenant le tableau des commandes ainsi qu'un
compteur de commandes et un compteur de commandes complétées.
                    Cette structure sera celle qu'on modifiera au cours du
traitement.
    liste_spe [in] : on recupère la structure liste_spe de type Specialites,
contenant le tableau des spécialités ainsi qu'un compteur du nombre de
spécialités
                    Cette structure ne sera pas modfiée.
    liste_cli [in] : on recupère la structure liste_cli de type Clients,
contenant le tableau des clients ainsi qu'un compteur de clients
                    Cette structure ne sera pas modifiée au cours du
traitement, elle servira à vérifier l'existence du client concerné.
     nom_commande [in] : le Mot nom_commande provient de la fonction
```

```
traite_progression, il s'agit du nom de commande entré par l'utilisateur lors
de l'appel de l'instruction
                    progression. Il sera utilisé pour vérifier la présence
d'une commande avec le même nom dans la liste des commandes liste_com
   */
   int i, j, affiche = 0, compteur_spe = 0, cpt_com = liste_com-
>nb_commandes, cpt_spe = liste_spe->nb_specialites, cpt_cli = liste_cli-
>nb_clients;
   long nb_heures_rest = -1, cout_total = 0, cout_client = 0;
   Mot nom_client;
   Booleen fin = FAUX;
   for (i = 0; i < cpt_com; ++i) { // on parcours le tableau des commandes
      if (strcmp(nom_commande, liste_com->tab_commandes[i].nom) == 0) { // si
la commande existe
         for (j = 0; j < cpt_spe; ++j) { // on parcours les spé de la
commande
            cout_total = liste_spe->tab_specialites[j].cout_horaire *
liste_com->tab_commandes[i].taches_par_specialite[j].nb_heures_effectuees;
            nb_heures_rest = liste_com-
>tab_commandes[i].taches_par_specialite[j].nb_heures_effectuees - liste_com-
>tab_commandes[i].taches_par_specialite[j].nb_heures_requises;
            if (nb_heures_rest == 0 && cout_total > 0) {
               ++compteur_spe;
            }
         }
         for (j = 0; j < cpt_spe; ++j) { // on parcours le tableau des
spécialtiés
            cout_total = liste_spe->tab_specialites[j].cout_horaire *
liste_com->tab_commandes[i].taches_par_specialite[j].nb_heures_effectuees;
            nb_heures_rest = liste_com-
>tab_commandes[i].taches_par_specialite[j].nb_heures_effectuees - liste_com-
>tab_commandes[i].taches_par_specialite[j].nb_heures_requises;
            if (nb_heures_rest == 0 && cout_total > 0 && liste_com-
>tab_commandes[i].specialites_actives == compteur_spe) {
               if (affiche == 0) {
                  printf(MSG_FACTURATION, nom_commande);
                  printf("%s:%d", liste_spe->tab_specialites[j].nom,
cout_total);
                  affiche += 1;
               }
               else if (affiche > 0) {
                  printf(", ");
                  printf("%s:%d", liste_spe->tab_specialites[j].nom,
cout_total);
                  ++affiche;
               }
            }
         }
         if (affiche == liste_com->tab_commandes[i].specialites_actives) { //
si l (l'entier incrémenté lorsqu'une spé d'une commande est complète) est
égal au nombre de spé d'une commande, alors la commande est complète.
            printf("\n");
            liste_com->tab_commandes[i].completion_commande = VRAI;
            liste_com->tab_commandes[i].specialites_actives = 0;
            ++liste_com->nb_commandes_completees;
         }
         affiche = 0;
      }
   }
   int k;
   if (liste_com->nb_commandes_completees == liste_com->nb_commandes &&
liste_com->nb_commandes > 0) { // si à l'issue de la dernière facturation,
toutes les commandes ont été facturées...
      printf(MSG_FACTURATION_FIN);
```

```
      for (k = 0; k < cpt_cli; ++k) { // on parcours le tableau des clients
         strcpy(nom_client, liste_cli->tab_clients[k]); // on copie le nom du
client
         cout_client = 0;
         for (i = 0; i < cpt_com; ++i) { // on parcours le tableau des
commandes
            if (liste_com->tab_commandes[i].idx_client == k) { // si le
client concerné par la commande est bien désigné par son index dans le
tableau des clients
               for (j = 0; j < cpt_spe; ++j) { // on parcours les spé de la
commande
                  cout_total = liste_spe->tab_specialites[j].cout_horaire *
liste_com->tab_commandes[i].taches_par_specialite[j].nb_heures_effectuees; //
le cout de chaque spé = cout horaire * temps effectué
                  cout_client += cout_total; // on l'ajoute au cout déjà
calculé
               }
            }
         }
         if (k == 0)
            printf("%s:%d", nom_client, cout_client);
         if (k > 0) {
            printf(", ");
            printf("%s:%d", nom_client, cout_client);
         }
      }
      printf("\n");
      fin = VRAI;
   }
   if (fin == VRAI)
      return 1;
   if (fin == FAUX)
      return 0;
   return 0;
}

void passe(Specialites* liste_spe, Travailleurs* liste_tr, Commandes*
liste_com, int idx_tache, int idx_com) {
   //printf(MSG_PASSE);
   /* chaque instruction est executée en ligne, l'une après l'autre; donc si
on dit progression
    et que tous les arguments pour progression sont passés, on passe à
l'instruction suviante */

    /*
    [brief] La fonction passe a pour but d'effectuer une réallocation de la
tâche spécifiée précédemment pour la commande spécifiée précédemment (par les
nombres idx_tache et idx_com).
    liste_com [in-out] : on recupère (en pointant vers elle) la structure
liste_com de type Commandes, contenant le tableau des commandes ainsi qu'un
compteur de commandes et un compteur de commandes complétées.
                  Cette structure sera celle qu'on modifiera au cours du
traitement.
    liste_tr [in-out] : on recupère la structure liste_tr de type
Travailleurs, contenant le tableau des travailleurs ainsi qu'un compteur de
travailleurs
                  On pourra ainsi y modifier pour le travailleur le nombre
d'heures lui ayant été assigné.
    liste_spe [in] : on recupère la structure liste_spe de type Specialites,
contenant le tableau des spécialités ainsi qu'un compteur du nombre de
spécialités
                   Cette structure ne sera pas modfiée.
    idx_tache [in-out] : cet entier correspond à l'index de la tâche modifiée
lors du traitement de la progression
```

```
     idx_com [in-out] : cet entier correspond à l'index de la commande
modfiée lors du traitement de la progression
    */
   int k, id_tr_rech = -1, cpt_tr = liste_tr->nb_travailleurs;
   long heuremin = -1;

   for (k = 0; k < cpt_tr; ++k) { // on parcours le tableau des travailleurs
      if (liste_tr->tab_travailleurs[k].tags_competences[idx_tache] == VRAI
&& id_tr_rech == -1) {// si c'est la première valeur du tab et qu'elle a la
propriété, elle devient auto le plus petit.
         id_tr_rech = k; // id_tr_rech ("id travailleur recherché" correspond
au travailleur trouvé par application de l'algorithme)
         heuremin = liste_tr->tab_travailleurs[k].nb_heures_pr_completion;
         // printf("voie a : %d, %d", id_tr_rech, heuremin);
      }
      else if (liste_tr->tab_travailleurs[k].tags_competences[idx_tache] ==
VRAI && id_tr_rech >= 0 && id_tr_rech < cpt_tr && heuremin > 0 && heuremin >
liste_tr->tab_travailleurs[k].nb_heures_pr_completion) { // sinon si c'est
une autre valeur, que la précédente est plus grande qu'elle et qu'elle a le
tag, alors elle devient la plus petite
         id_tr_rech = k;
         heuremin = liste_tr->tab_travailleurs[k].nb_heures_pr_completion;
         // printf("voie b : %d, %d", id_tr_rech, heuremin);
      }
      else if (liste_tr->tab_travailleurs[k].tags_competences[idx_tache] ==
VRAI && id_tr_rech >= 0 && id_tr_rech < cpt_tr && heuremin > liste_tr-
>tab_travailleurs[k].nb_heures_pr_completion) {
         id_tr_rech = k;
         // printf("voie c : %d, %d", id_tr_rech, heuremin);
      }
   }

   liste_tr->tab_travailleurs[liste_com-
>tab_commandes[idx_com].taches_par_specialite[idx_tache].idx_travailleur].nb_
heures_pr_completion -= liste_com-
>tab_commandes[idx_com].taches_par_specialite[idx_tache].nb_heures_requises -
liste_com-
>tab_commandes[idx_com].taches_par_specialite[idx_tache].nb_heures_effectuees
;
   liste_com-
>tab_commandes[idx_com].taches_par_specialite[idx_tache].idx_travailleur =
id_tr_rech;
   int nb_heures_taches = 0;
   nb_heures_taches = liste_com-
>tab_commandes[idx_com].taches_par_specialite[idx_tache].nb_heures_requises -
liste_com-
>tab_commandes[idx_com].taches_par_specialite[idx_tache].nb_heures_effectuees
;
   liste_tr->tab_travailleurs[id_tr_rech].nb_heures_pr_completion +=
nb_heures_taches;

}
// specialites ------------------------
void traite_specialites(const Specialites* liste_spe) {
   /*
   [brief] La fonction traite_specialites a pour but de lister les
spécialités déclarées dans le tableau des spécialités de la structure
liste_spe.
   liste_spe [in-out] : on recupère la structure liste_spe de type
Specialites, contenant le tableau des spécialités ainsi qu'un compteur de
spécialités
   */
   printf(MSG_SPECIALITES);
   int i, cpt_nb_spe = liste_spe->nb_specialites;
```

```
    for (i = 0; i < cpt_nb_spe; ++i) { // on parcours le tableau des
spécialités
        if (i < cpt_nb_spe - 1) // si c'est pas l'avant dernière
            printf(" %s/%d,", liste_spe->tab_specialites[i].nom, liste_spe-
>tab_specialites[i].cout_horaire); // avec virgule
        else
            printf(" %s/%d", liste_spe->tab_specialites[i].nom, liste_spe-
>tab_specialites[i].cout_horaire); // sinon sans
    }
    printf("\n");
}
// travailleurs ------------------------
void traite_travailleurs(Travailleurs* liste_tr, Specialites* liste_spe) {
    /*
    [brief] La fonction traite_travailleurs a pour but de lister les
travailleurs, soit en fonction de la spécialité qu'ils possèdent, soit dans
leur totalité
    liste_tr [in-out] : on recupère la structure liste_tr de type
Travailleurs, contenant le tableau des travailleurs ainsi qu'un compteur de
travailleurs
    liste_spe [in-out] : on recupère la structure liste_spe de type
Specialites, contenant le tableau des spécialités ainsi qu'un compteur de
spécialités
    */
    Mot nom_specialite;
    get_id(nom_specialite);
    int i, cpt_nb_spe = liste_spe->nb_specialites;
    if (strcmp(nom_specialite, "tous") == 0) { // si le mot clé est "tous"...
        for (i = 0; i < cpt_nb_spe; ++i) { // on affiche pour chaque spé
possible ses travailleurs
            strcpy(nom_specialite, liste_spe->tab_specialites[i].nom);
            liste_travailleurs(liste_spe, liste_tr, nom_specialite);
        }
    }
    else {
        liste_travailleurs(liste_spe, liste_tr, nom_specialite);
    }
}

void liste_travailleurs(Specialites* liste_spe, Travailleurs* liste_tr, Mot
nom_specialite) {
    /*
    [brief] La fonction traite_travailleurs a pour but de lister les
travailleurs, soit en fonction de la spécialité qu'ils possèdent uniquement
(et peut
    donc être appelée plusieurs fois pour couvrir tous les cas de figure en
cas d'usage du mot-clé "tous"
    liste_tr [in-out] : on recupère la structure liste_tr de type
Travailleurs, contenant le tableau des travailleurs ainsi qu'un compteur de
travailleurs
    liste_spe [in-out] : on recupère la structure liste_spe de type
Specialites, contenant le tableau des spécialités ainsi qu'un compteur de
spécialités
    nom_specialite [in] : le Mot nom_specialites provient de la fonction
traite_travailleurs, il s'agit du nom de specialite entré par l'utilisateur
lors de l'appel de l'instruction
                          specialite. Il sera utilisé pour vérifier la présence
d'une specialite avec le même nom dans la liste des specialites liste_spe
    */
    int i, j, k, set = 0, nb_pr_spe = 0, cpt_nb_spe = liste_spe-
>nb_specialites, cpt_nb_tr = liste_tr->nb_travailleurs;
    printf(MSG_TRAVAILLEURS, nom_specialite);
    for (i = 0; i < cpt_nb_spe; ++i) { // on fouille le tableau de spécialités
        if (strcmp(nom_specialite, liste_spe->tab_specialites[i].nom) == 0) {
```

```
// si on trouve notre spé...
        for (j = 0; j < cpt_nb_tr; ++j) { // on fouille celui des
travailleurs
            if (liste_tr->tab_travailleurs[j].tags_competences[i] == VRAI) {
// si l'un des travailleurs a la spé
                if (set == 0) { // (si on a déjà vérifié le nombre de
travailleurs lors de cet usage de la fct)
                    for (k = 0; k < MAX_TRAVAILLEURS; ++k) { // sinon, vérifie
combien de travailleurs ont la spé
                        if (liste_tr->tab_travailleurs[k].tags_competences[i] ==
VRAI) {
                            ++nb_pr_spe;
                        }
                    }
                    set = 1;
                }
                if (j < nb_pr_spe - 1) // enfin on affiche le nom selon le
rang du travailleur avec la spé par rapport au prochain avec la spé
                    printf(" %s,", liste_tr->tab_travailleurs[j].nom);
                else
                    printf(" %s", liste_tr->tab_travailleurs[j].nom);
            }
        }
        printf("\n");
    }
  }
}

// client ------------------------------
void traite_client(Clients* liste_cli, Commandes* liste_com) {
   /*
   [brief] La fonction traite_client a pour but de lister les clients, soit
l'un en particulier, soit dans leur totalité
   liste_cli [in-out] : on recupère la structure liste_cli de type Clients,
contenant le tableau des clients ainsi qu'un compteur de clients
   liste_spe [in-out] : on recupère la structure liste_spe de type
Specialites, contenant le tableau des spécialités ainsi qu'un compteur de
spécialités
   */
   Mot nom_client;
   get_id(nom_client);
   int i, cpt_nb_cli = liste_cli->nb_clients;
   if (strcmp(nom_client, "tous") == 0) { // si le mot clé est "tous"...
      for (i = 0; i < cpt_nb_cli; ++i) { // on affiche pour chaque client
possible ses commandes
         strcpy(nom_client, liste_cli->tab_clients[i]);
         liste_commandes(liste_cli, liste_com, nom_client);
      }
   }
   else {
      liste_commandes(liste_cli, liste_com, nom_client);
   }
}

void liste_commandes(Clients* liste_cli, Commandes* liste_com, Mot
nom_client) {
   /*
   [brief] La fonction liste_commandes a pour but de lister les commandes de
chaque client, soit l'un en particulier, soit dans leur totalité
   liste_cli [in-out] : on recupère la structure liste_cli de type Clients,
contenant le tableau des clients ainsi qu'un compteur de clients
   liste_spe [in-out] : on recupère la structure liste_spe de type
Specialites, contenant le tableau des spécialités ainsi qu'un compteur de
spécialités
```

```
   nom_client [in-out] : le Mot nom_client provient de la fonction
traite_client, il s'agit du nom de client entré par l'utilisateur lors de
l'appel de l'instruction
                     client. Il sera utilisé pour identifier le client
assigné à une commande.
   */
   unsigned int i, j, affiche = 0, cpt_cli = liste_cli->nb_clients,
com_pr_cli = 0, cpt_com = liste_com->nb_commandes;
   printf(MSG_CLIENT, nom_client);
   for (i = 0; i < cpt_cli; ++i) { // on parcours le tableau des clients
      if (strcmp(liste_cli->tab_clients[i], nom_client) == 0) { // on
retrouve le notre
         /*if (set == 0) { // si on ne sait pas combien il a de commandes
            for (j = 0; j < cpt_com; ++j) { //... on parcours le tableau des
commandes...
               if (liste_com->tab_commandes[j].idx_client == i) { // (si l'id
client commande = id client)
                  ++com_pr_cli; // pour les compter !
               }
            }
            set = 1;
         }*/
         for (j = 0; j < cpt_com; ++j) { // on parcours le tableau client
            if (liste_com->tab_commandes[j].idx_client == i) { // si une
commande est assignée au client
               if (affiche == 0) {
                  printf("%s", liste_com->tab_commandes[j].nom);
                  ++affiche;
               }
               else if (affiche > 0) {
                  printf(", %s", liste_com->tab_commandes[j].nom);
               }
            }
         }
      }
   }
   printf("\n");
}

// supervision -------------------------
void traite_supervision(Commandes* liste_com, Specialites* liste_spe) {
   /*
   [brief] La fonction traite_supervision a pour but de lister pour chaque
commande les tâches par spécialité active ainsi que le nombre d'heures
restant à effectuer pour
   chacune d'entre elles.
   liste_com [in-out] : on recupère (en pointant vers elle) la structure
liste_com de type Commandes, contenant le tableau des commandes ainsi qu'un
compteur de commandes et un compteur de commandes complétées.
   liste_spe [in-out] : on recupère la structure liste_spe de type
Specialites, contenant le tableau des spécialités ainsi qu'un compteur de
spécialités
   */
   int i, j, nb_h_r_verif = 0, affiche = 0, tache_pr_spe = 0, nb_h_e = -1,
nb_h_r = -1, cpt_com = liste_com->nb_commandes, cpt_spe = liste_spe-
>nb_specialites;
   Mot nom_specialite;
   for (i = 0; i < cpt_com; ++i) { // on consulte la liste des commandes
      printf(MSG_SUPERVISION, liste_com->tab_commandes[i].nom);
      for (j = 0; j < cpt_spe; ++j) { // on consulte la liste des spé
         nb_h_e = liste_com-
>tab_commandes[i].taches_par_specialite[j].nb_heures_effectuees; // on
récupère le nb heures effectuees pour cette spé dans la commande
         nb_h_r = liste_com-
```

```
>tab_commandes[i].taches_par_specialite[j].nb_heures_requises; // on recupère
le nb heures requises pour cette spé dans la commande
         strcpy(nom_specialite, liste_spe->tab_specialites[j].nom); //
nom_specialite devient celui de la spe
         if (nb_h_e >= 0 && nb_h_r > 0 && nb_h_r < 1000) {
            if (affiche == 0) { // si la spé vérifiée a pour id le nb de spé
possible - 1 ET que il y a plus de zéro spé possible ET que le nb heures
requises est sup à zero
               printf("%s:%d/%d", nom_specialite, nb_h_e, nb_h_r); // alors
on écrit la spé sans virgule
               affiche += 1;
            }
            else if (affiche > 0) { // sinon
               printf(", %s:%d/%d", nom_specialite, nb_h_e, nb_h_r); // avec
virgule (ou on écrit pas)
            }
         }
      }
      affiche = 0;
      printf("\n");
   }

}
// charge ------------------------------
void traite_charge(Travailleurs* liste_tr, Specialites* liste_spe, Commandes*
liste_com) {
   /*
   [brief] La fonction traite_charge a pour but de lister pour un travailleur
les tâches en cours qui lui sont assignées.
   liste_tr [in-out] : on recupère la structure liste_tr de type
Travailleurs, contenant le tableau des travailleurs ainsi qu'un compteur de
travailleurs
   liste_com [in-out] : on recupère (en pointant vers elle) la structure
liste_com de type Commandes, contenant le tableau des commandes ainsi qu'un
compteur de commandes et un compteur de commandes complétées.
   liste_spe [in-out] : on recupère la structure liste_spe de type
Specialites, contenant le tableau des spécialités ainsi qu'un compteur de
spécialités
   */
   int i, j, k, affiche = 0, nb_heures_rest = 0, cpt_tr = liste_tr-
>nb_travailleurs, cpt_com = liste_com->nb_commandes, cpt_spe = liste_spe-
>nb_specialites;
   Mot nom_travailleur;
   get_id(nom_travailleur);

   for (k = 0; k < cpt_tr; ++k) { // on parcours le tableau des travailleurs
      if (strcmp(nom_travailleur, liste_tr->tab_travailleurs[k].nom) == 0) {
         printf(MSG_CHARGE, nom_travailleur);
         for (j = 0; j < cpt_com; ++j) { // on parcours le tableau des
commandes déclarées
            for (i = 0; i < cpt_spe; ++i) { // on parcours le tableau des spe
déclarées
               if (liste_com-
>tab_commandes[j].taches_par_specialite[i].idx_travailleur == k) { // si l'id
du travailleur correspond à celui de la spe de la tache de la commande
                  nb_heures_rest = liste_com-
>tab_commandes[j].taches_par_specialite[i].nb_heures_requises - liste_com-
>tab_commandes[j].taches_par_specialite[i].nb_heures_effectuees;
                  if (nb_heures_rest == 0) // si une tache est déjà remplie
                     continue; // on passe à la prochaine itération de la
boucle (donc rien ne s'affiche)
                  if (affiche == 0) { // si c'est la première fois qu'on
affiche, alors on mets sans virgule
                     printf("%s/%s/%dheure(s)", liste_com-
```

```
>tab_commandes[j].nom, liste_spe->tab_specialites[i].nom, nb_heures_rest); //
sans virgule
                    affiche += 1;
                    continue;
                }
                else if (affiche > 0) // sinon les valeurs suivantes
préparent une virgule avant de s'afficher
                    printf(", %s/%s/%dheure(s)", liste_com-
>tab_commandes[j].nom, liste_spe->tab_specialites[i].nom, nb_heures_rest); //
sinon sans virgule
                }
            }
        }
        printf("\n");
    }
  }
}
// interruption ------------------------
void traite_interruption() {
   /*
   [brief] La fonction traite_interruption affiche un message d'interruption
(avant dans le main d'interrompre le programme)
   */
   printf(MSG_INTERRUPTION);
}

//Boucle principale ---------------------------------------------------------
-
int main(int argc, char* argv[]) {
   if (argc >= 2 && strcmp("echo", argv[1]) == 0) {
      EchoActif = VRAI;
   }
   // on commence par initialiser quatre strctures qui porteront les données
nécessaires au fonctionnement du programme
   Specialites liste_spe;
   Travailleurs liste_tr;
   Clients liste_cli;
   Commandes liste_com;

   // on initialise ensuite certaines de leurs valeurs incrémentables (pour
éviter d'avoir un incrément sur une valeur inconnue)
   liste_com.nb_commandes = 0;
   liste_com.nb_commandes_completees = 0;
   liste_spe.nb_specialites = 0;
   liste_tr.nb_travailleurs = 0;
   liste_cli.nb_clients = 0;

   Mot buffer;
   Booleen progression_last_used = FAUX; // se déclenche uniquement lorsque
progression est la dernière instruction exécutée
   int idx_tache = -1;
   int idx_com = -1;
   int program_state = 0; //program_state prend le statut d'exécution du
programme; 0 et le programme continue, 1 et le programme s'arrête
immédiatement.

   while (VRAI) {
      get_id(buffer);
      if (strcmp(buffer, "developpe") == 0) {
         traite_developpe(&liste_spe);
         continue;
      }
      if (strcmp(buffer, "embauche") == 0) {
         traite_embauche(&liste_tr, &liste_spe);
```

```
            continue;
        }
        if (strcmp(buffer, "demarche") == 0) {
            traite_demarche(&liste_cli);
            continue;
        }
        if (strcmp(buffer, "commande") == 0) {
            traite_commande(&liste_com, &liste_cli, &liste_spe);
            continue;
        }
        if (strcmp(buffer, "tache") == 0) {
            traite_tache(&liste_com, &liste_spe, &liste_tr);
            continue;
        }
        if (strcmp(buffer, "progression") == 0) {
            program_state = traite_progression(&liste_com, &liste_spe,
&liste_cli, &liste_tr, &idx_tache, &idx_com);
            if (program_state == 0) { // si la progression n'a pas déclenché de
facturation finale
                continue; // le programme continue
            }
            if (program_state == 1) { // si la progression a déclenché une
facturation finale
                break; // le programme sort de sa boucle (il s'arrête)
            }
        }
        if (strcmp(buffer, "passe") == 0) {
            passe(&liste_spe, &liste_tr, &liste_com, idx_tache, idx_com);
            continue;
        }
        if (strcmp(buffer, "specialites") == 0) {
            traite_specialites(&liste_spe);
            continue;
        }
        if (strcmp(buffer, "travailleurs") == 0) {
            traite_travailleurs(&liste_tr, &liste_spe);
            continue;
        }
        if (strcmp(buffer, "client") == 0) {
            traite_client(&liste_cli, &liste_com);
            continue;
        }
        if (strcmp(buffer, "supervision") == 0) {
            traite_supervision(&liste_com, &liste_spe);
            continue;
        }
        if (strcmp(buffer, "charge") == 0) {
            traite_charge(&liste_tr, &liste_spe, &liste_com);
            continue;
        }
        if (strcmp(buffer, "interruption") == 0) {
            traite_interruption();
            break;
        }
        printf("!!! instruction inconnue >%s< !!!\n", buffer);
    }
    return 0;
}
```