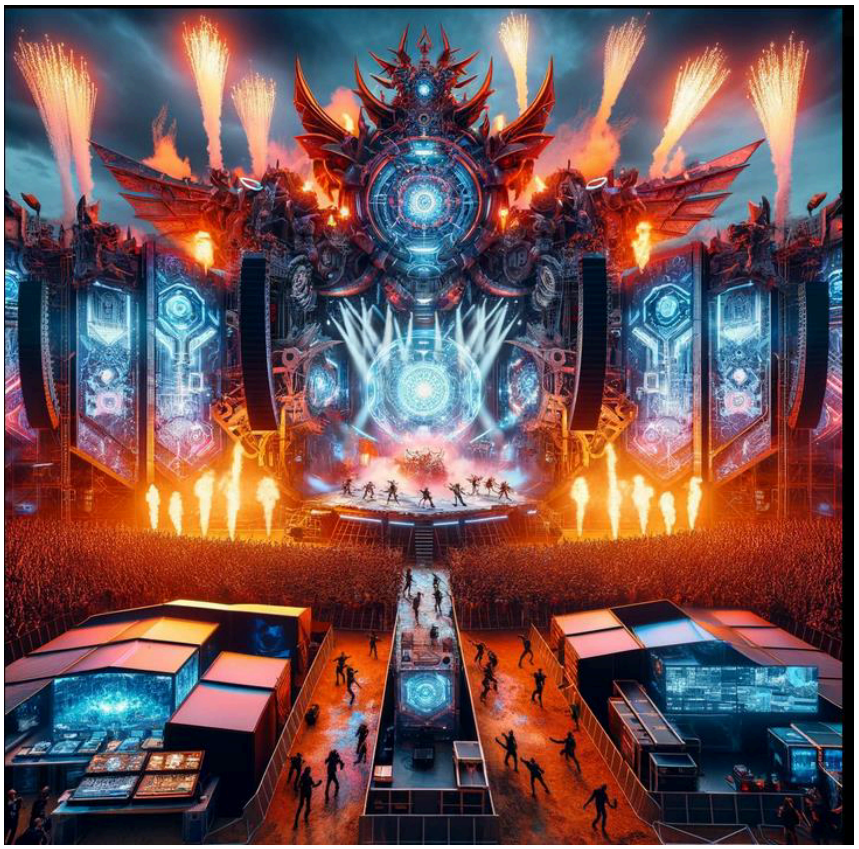
CY Fest



Par:

Jean-Luc MASLANKA
Damien KAE-NUNE
Gaspard SAVES

Classe de:

Pré-ingénieur, première année
MEF 2, groupe J

Sous la direction de:

Khalil BACHIRI, enseignant-chercheur en intelligence artificielle et analyse de données

Romuald GRIGNON, enseignant en informatique

Le 26 mai 2024

Année 2023-2024

SOMMAIRE :

Notre équipe est composée de trois personnes : Gaspard, Damien et Jean-Luc. Au cours de la semaine du 22 avril, nous avons réfléchi aux différents projets potentiels. Deux sujets ont attiré notre attention : CY-BER Path et CY'Fest. Après plusieurs concertations et méditations, nous avons décidé de choisir CY'Fest, car nous le trouvions plus facile sur la partie code même si le débogage allait être plus compliqué.

Au sein de l'équipe, Jean-Luc et Damien avaient pour rôle de coder respectivement les programmes relatifs à manager et à festivalier. En effet, leur rôle principal était de réfléchir et coder le programme. Gaspard a ainsi joué le rôle de débogguer en codant l'interface chargée de relier les programmes et en résolvant les erreurs générées par l'implémentation des nouvelles fonctions. Bien évidemment, au début du projet tout le monde codait car aucune fonction n'était prête.

Pendant l'élaboration du projet, Jean-Luc s'est occupé du programme manager. Il a codé tout ce qui était nécessaire à la bonne création et modification des salles et des concerts. Il a par ailleurs complété le programme hour qui permet de récupérer l'heure actuelle et la compare avec celle du concert afin de voir si un concert est terminé ou non. Damien était chargé de la fonction festivalier. Celle-ci permet de se créer un compte ou de se connecter. Elle permet également de réserver un siège dans une salle. Gaspard s'est occupé du programme interface et a de même créé et complété au fur et à mesure le programme smartrobusnest.

Les de nombreux désaccords, n'ont pas affecté notre cohésion d'équipe. Chacun avait un rôle précis mais il y a eu beaucoup d'aides très précieuses lorsque certains programmes ne fonctionnaient pas. Ainsi, les rôles n'étaient pas fixés mais chacun se retrouvait dans le projet.

Après avoir choisi le sujet, nous avons organisé 2 réunions successives afin de faire un brainstorming des structures, des fonctions et de l'imbrication des fichiers nécessaires à inclure. Le but ici était de mieux comprendre les attentes du sujet et d'avoir une vue d'ensemble sur la programmation de celui-ci.

En premier lieu, nous avons pensé à toutes les catégories de structures nécessaires pour créer un festival. Ainsi, nous avons noté cinq structures essentielles: Date, Siège, Utilisateur, Salle et Concert. Ces structures nous permettent de lier tous les éléments entre eux. Ensuite, nous avons développé ces structures en fonction des besoins logiques par rapport au projet.

Deuxièmement, nous avons réfléchi aux fonctions afin de construire au minimum le projet. Il y avait donc hour ayant pour objectif d'être la fonction de temporisation demandée dans le projet. Nous avons de même pensé à la fonction réservation qui a pour but de réserver les sièges des utilisateurs et de voir les réservations des utilisateurs. Nous avons encore la fonction manager qui est la plus importante du projet. En effet, elle nous servira à créer des salles et des concerts. On y ajoutera également des options de modifications des salles. Nous avons par ailleurs réfléchi à une fonction interface qui aura le rôle de guide dans les commandes. C'est donc elle qui va gérer les actions qu'on effectuera. Enfin, nous avons imaginé la fonction smartrobusnest qui aura donc pour objectif de contenir toutes les fonctions nécessaires à la robustesse de notre code.

En ce qui concerne les fichiers, nous avons décidé de créer une bibliothèque robustesse afin d'appeler les fonctions dont on aurait besoin. L'interface permettant de faire une action en tant que festivalier et manager nous a convaincu qu'il fallait rajouter un fichier interface afin de relier les deux interfaces (festivalier et manager). Par la suite, le fichier manager et le fichier festivalier étaient nécessaires pour toutes les commandes respectives. Il était donc inconcevable de ne pas les créer. Par ailleurs, nous avons ajouté le fichier hour et les fichiers de sauvegarde afin d'alléger le fichier manager et festivalier car ils ne dépendent pas de manager et de festivalier.

Ensuite, il était impératif de créer un espace personnel pour chaque utilisateur, chaque individu devant réserver son siège. Par conséquent, chaque utilisateur devait disposer d'un identifiant et d'un mot de passe propres. Pour cela, il a été nécessaire de développer une fonction de création de compte. Initialement, cette fonction avait pour rôle de saisir un mot de passe et un identifiant. Cependant, nous avons constaté la possibilité de duplication des identifiants. Pour remédier à ce problème, la fonction de création de compte ne prend désormais que le mot de passe et génère un identifiant unique vérifié, évitant ainsi toute duplication.

Par la suite, nous nous sommes confrontés à la réalité en commençant à coder. Ne fusse pas une surprise de constater que, lors de la programmation des fonctions que le projet était bien plus complexe que prévu. En effet, la création d'un compte pour chaque utilisateur nécessitait la sauvegarde de chaque identifiant et mot de passe. Ainsi, nous avons remarqué que cela posait un véritable problème quant aux données stockées, car si le programme s'arrêtait, les identifiants et mots de passe étaient supprimés. Notre sophisme était donc partiellement correct mais il fallait rajouter de la matière. Nous avons donc décidé

de stocker ces informations dans un fichier, ce qui nous permettrait, grâce à une réallocation, de les recopier dans un nouveau tableau d'utilisateurs.

Ensuite, nous avons rencontré un problème de temporisation. En effet, il n'est pas possible de réaliser un minuteur en C, car le programme perd toute notion du temps lorsqu'il s'exécute. Nous avons donc dû concevoir un programme qui prenait l'heure de notre machine et la comparait avec l'heure du concert afin de déterminer si celui-ci arrivait à échéance. Après avoir débattu longuement sur ce programme, qui ne semblait pourtant pas très compliqué, nous avons réussi à obtenir un résultat satisfaisant. Effectivement, notre programme prend l'heure locale de l'ordinateur et la compare avec une autre heure sans nécessiter de minuteur.

Concernant l'appel croisé de fonctions, nous avons dû créer un fichier distinct pour chaque fonction. Cette méthode nous a permis d'avoir un fichier principal centralisant l'exécution de l'ensemble du code. En décomposant le projet en modules fonctionnels, nous avons pu simplifier le processus de développement et de débogage, bien que cette approche ait également introduit une certaine complexité en termes de gestion des interdépendances entre les fichiers et nous a pris du temps, elle nous a aussi demandé un temps considérable

Le débogage, quant à lui, s'est avéré extrêmement chronophage. Il nous a fallu assembler toutes les fonctions avec des variables absentes du programme initial. Par conséquent, nous avons dû vérifier, corriger et ajouter divers éléments au programme, ce qui a considérablement alourdi notre charge de travail. Cette étape cruciale du développement a mis en lumière les faiblesses initiales de notre conception et la nécessité d'une phase de test rigoureuse et systématique. Toutefois elle nous a appris qu’en informatique, la réflexion du code représente 80%. L’étape décisive et celle qui est la plus chronophage est donc la réflexion puisqu’elle limite ou accroît exponentiellement la difficulté du codage.

Par la suite, il a été indispensable de renforcer la robustesse de la fonction scanf, notamment pour gérer les cas où un entier (%d) est attendu et où un caractère est saisi dans le terminal. Pour ce faire, nous avons dû implémenter un tampon, une solution que nous ignorions au départ et que nous avons mis du temps à découvrir.

De plus, nous avons rencontré un problème majeur dans le codage des fichiers binaires quant à la sauvegarde des sièges. Lorsque nous enregistrons les données de la salle dans un fichier binaire puis qu’ensuite nous créons une deuxième salle afin de s’assurer de la bonne sauvegarde des données, celle-ci ne s’affiche pas. Après un interminable débogage et d'intenses modifications, nous sommes parvenus à résoudre le problème. Nous sauvegardions les données des sièges, mais pas les sièges eux-mêmes. Ainsi, la sauvegarde était inutile.. L’implémentation de la fonction *Savesit* nous a permis de sauvegarder les sièges eux-mêmes puis de sauvegarder les données relatives à ces derniers.

Quant à la conception du programme, elle a été très laborieuse. En effet, il fallut tout au long de la réalisation de ce projet, cerner les moindres détails de nos programmes. Ainsi, il était nécessaire de revenir sur ce qu’on faisait. En l’occurrence, cela nécessite de réfléchir deux fois avant de coder malgré notre élaboration au préalable du projet et même certaines fois modifier la plupart du code.

Concernant la gestion des erreurs, nous pensons qu’il faudrait complexifier la vérification des variables. Effectivement, il faudrait à l’avenir que notre programme ne s’arrête pas si par

exemple un pointeur est nul mais revienne en arrière pour réessayer le pointeur. Par ailleurs, si le programme s'arrête, la sauvegarde dans les fichiers ne s'effectue pas donc nous perdons toutes les données et ne pouvons pas aller au bout de l'exécution du programme. Aussi, remplacer « exit(1) » par une autre vérification serait à l'avenir à privilégier quant à la bonne exécution du programme.

Explication fonction :

En premier lieu, nous avons réfléchi à un programme qui renforce la robustesse de notre projet. Effectivement, le programme *smartrobustness* vérifie plusieurs conditions sous-jacentes à notre programme. Nous avons premièrement la fonction better_scan. Celle-ci a pour objectif de demander ce que le manager souhaite si le flux d'entrée n'est pas celui attendu. D'autre part, nous avons la fonction *VerifPointer.* En effet, afin de s'assurer de la validité d'un pointeur lors de l'exécution du code de manière pérenne, cette fonction va vérifier si le pointeur n'est pas nul. Nous avons également implémenté des fonctions pour les fichiers tels que *checkOpenFile* qui lui vérifie si le fichier a été ouvert correctement, *checkCloseFile* qui vérifie si le fichier a bien été fermé.

Festivalgoer.c

La fonction *constrTabFestivalGoers* alloue de la mémoire pour un tableau d'utilisateurs, qui a pour taille userCount. Elle va vérifier que le pointeur alloué n'est pas NULL. Enfin, elle retourne ce tableau alloué.

La fonction *checkIdFest* parcourt le tableau d'utilisateurs pour vérifier si l'ID de connexion (idco) existe déjà. Si effectivement l'ID est trouvé, on va afficher "Identifiant correct sinon on doit afficher le message d'erreur.

La fonction *checkPasswordF* est vérifié que l'ID de connexion (idco) et le mot de passe (passwordco) sont les mêmes. Elle parcourt donc le tableau d'utilisateurs pour trouver une correspondance et affichera "Mot de passe correct". Sinon, elle affiche une erreur.

La fonction *generateUniqueId* donne un ID unique pour un utilisateur. En effet, elle crée un nouvel ID aléatoire et vérifie par la suite s'il est déjà utilisé dans le tableau d'utilisateurs. Si l'ID est déjà utilisé, elle continue à en générer un jusqu'à en trouver un qui est unique.

La fonction *accountCreationFestivalGoers* crée, elle, la création d'un compte utilisateur. Elle demande à l'utilisateur de saisir un mot de passe, génère un ID unique pour l'utilisateur, et va implémenter le nouvel utilisateur au tableau des utilisateurs. Elle affiche ensuite les identifiants de l'utilisateur puis appelle la fonction *choiceUser* pour que l'utilisateur fasse une autre action.

La fonction *displayUsers* affiche la liste de tous les utilisateurs avec leur ID et leur mot de passe.

La fonction *reserveSeat* crée la réservation d'un siège pour un concert. Elle demande à l'utilisateur de choisir une rangée et un siège et vérifie si le siège est déjà réservé. Si le siège est libre, affecte l'ID au siège et affiche la réservation.

La fonction *réservation* sert à l'utilisateur de réserver un siège pour un concert. Elle demande à l'utilisateur de saisir le nom du concert qu'il veut voir. Si le concert est trouvé, on appelle reserveSeat pour réserver un siège.

La fonction *my_reservation* va montrer la réservation d'un concert pour l'utilisateur. Elle demande donc à l'utilisateur le nom du concert qu'il veut voir et affiche sa réservation. Sinon il est indiqué qu'aucune réservation est trouvée

La fonction *interfaceFestivalGoers* propose trois options : se déconnecter, voir ses réservations, ou réserver un concert.

Interface.c:

La fonction *connectionFestivalGoers* gère la connexion des festivaliers. Elle demande un identifiant et vérifie s'il existe dans le tableau des utilisateurs. Si l'ID est correct, elle demande ensuite le mot de passe et vérifie s'il est correct.

La fonction *choiceCoFestivalGoers* permet plusieurs actions pour les festivaliers. L'utilisateur choisi entre créer un compte ou se connecter. L'utilisateur peut de même choisir de revenir en arrière.

La fonction *connectionManager* permet de se connecter à l'interface manager. Elle demande à l'utilisateur le code du manager (2000). Si le code est correct, on a accès à l'interface manager. Sinon on retourne à l'interface de choix « général ».

La fonction *choiceUser* va avoir différent choix de type de connexions. L'utilisateur doit choisir entre l'interface festivalier ou l'interface manager. Si l'utilisateur choisit d'exit le programme, elle libère la mémoire allouée et termine l'exécution.

Manager.c :

Le programme MANAGER.C a pour objectif principal de créer des salles et des concerts d'un festival. En effet, on utilise l'allocation dynamique pour créer des tableaux de salles et de concerts. Il y a aussi des fonctions pour initialiser, réinitialiser et afficher les sièges des salles avec des couleurs spécifiques. Le manager a aussi la possibilité de modifier des prix des sièges par catégorie et de créer de multiple de salles et de concerts. Le manager personnalise les noms, les catégories de sièges, les prix et les informations relatives aux concerts. En outre, une fonction de type « business management » permet au manager de voir le taux d'occupation des sièges ainsi que le chiffre d'affaires potentiel.