Compte rendu du projet

Delcroix Matthieu Lasserre Jean-Baptiste Sépulcre Dorian

13 décembre 2013

Liste des principales constantes

```c
#define NB_MAX_EPREUVES 15 // Nombre maximum d'épreuves
#define NB_MAX_PARTICIPANTS 100 // Nombre maximum de participants
#define NB_MAX_VELO 10 // Nombre maximum de vélos
#define NB_MANCHES 4 // Nombre de manches par épreuve

#define TAILLE_GRAND 20 // Grandes chaînes de caractères
#define TAILLE_PETIT 10 // Petites chaînes de caractères

// Constantes de contrôle des dates
#define JOUR_MIN 1
#define JOUR_MAX 31
#define MOIS_MIN 1
#define MOIS_MAX 12
#define ANNEE_MIN 2013
#define ANNEE_MAX 2050

#define AGE_MAX 100 // Âge maximum d'un participant
#define AGE_MIN 16 // Âge minimum d'un participant

#define DISTANCE_MAX 1000 // Distance maximale d'une épreuve
#define DENIVELE_MAX 10000 // Dénivelé maximum d'une épreuve
#define NB_MAX_OBSTACLES 1000 // Nombre maximum d'obstacles d'une épreuve
```

Structures utilisées

Nous utilisons les structures suivantes :
- CrossCountry ;
- Descente ;
- Trial ;
- Orientation ;
- Manche ;
- Velo ;
- Participant ;
- Possession ;
- Date ;
- Resultat ;
- Epreuve.

CrossCountry

```c
typedef struct {
  float fDistance; // Distance en km de l'épreuve
  int iDeniveleNeg; // Dénivelé négatif de la manche en m
  int iDenivelePos; // Dénivelé positif de la manche en m
}CrossCountry;
```

Descente

```
typedef struct {
  int iDeniveleNeg; // Dénivelé négatif de la manche en m
  char sDifficulte[TAILLE_PETIT]; // Difficulté de la manche (vert|bleu|
      rouge|noir)
}Descente;
```

Trial

```
typedef struct {
  int iNbObstacles; // Nombre d'obstacles de la manche
  char sDifficulte[TAILLE_PETIT]; // Difficulté de la manche (vert|bleu|
      rouge|noir)
  char sTypeObstacles[TAILLE_PETIT]; // Types des obstacles de la manche
      (bois|béton|acier|terre);
}Trial;
```

Orientation

```
typedef struct {
  int iDeniveleNeg; // Dénivelé négatif de la manche en m
  int iDenivelePos; // Dénivelé positif de la manche en m
  int iNbObstacles; // Nombre d'obstacles de la manche
  char sCodeIGN[TAILLE_PETIT]; // Code IGN de la carte à utiliser pour la
      manche, ex : 1547 OT
}Orientation;
```

Manche

```
typedef struct {
  CrossCountry * ptrCrossCountry; // Pointeur sur CrossCountry
  Descente * ptrDescente;; // Pointeur sur Descente
  Trial * ptrTrial;; // Pointeur sur Trial
  Orientation * ptrOrientation;; // Pointeur sur Orientation
}Manche;
```

Velo

```
typedef struct{
  int iID; //IDENTIFIANT NUMERIQUE DU VELO
  char sMarque[TAILLE_GRAND]; //MARQUE DU VELO(EX Specialized)
  char sType[TAILLE_GRAND]; //TYPE DU VELO(EX Hotrock)
  char sTailleCadre[TAILLE_PETIT]; //TAILLE CADRE -> XS, S, M, L XL, XXL
} Velo;
```

Participant

```
typedef struct{
  int iID; // Identifiant numérique du participant
  char sNom[TAILLE_GRAND] ; // Nom du participant
  char sPrenom[TAILLE_GRAND] ; // Prénom du participant
  char sClub[TAILLE_GRAND]; // Club du participant
  int iAge; // Âge du participant
  Velo * partVelo[NB_MAX_VELO]; // Liste des vélos du participant
  int iNbrVelo; // Nombre de vélos du participant
  int iListeIDEpreuve[NB_MAX_EPREUVES]; // Liste des épreuves auxquelles
      le participant est inscrit
  int iNbrEprv; // Nombre d'épreuves du participant
}Participant;
```

Possession

Indique le vélo utilisé par un participant durant une épreuve.

```
typedef struct{
  Participant * ptrParticipant; // Pointeur sur Participant
  Velo * ptrVelo; // Pointeur sur vélo
} Possession;
```

Date

```
typedef struct{
  int iJour;
  int iMois;
  int iAnnee;
}Date;
```

Resultat

Résultats d'un participant dans une épreuve.

```
typedef struct{
  Participant * resultatParticipant; // Pointeur sur Participant
  int iClassementManche1; // Points du participant dans la manche 1
  int iClassementManche2; // Idem manche 2
  int iClassementManche3; // Idem manche 3
  int iClassementManche4; // Idem manche 4
  int iScore; // Points du participant dans l'épreuve
}Resultat;
```

Epreuve

```c
typedef struct{
  int iId; // Identifiant numérique de l'épreuve
  char sTitre[TAILLE_GRAND]; // Titre de l'épreuve
  char sLieu[TAILLE_GRAND]; // Lieu de l'épreuve
  char sCpRendu[1024]; // Compte rendu de l'épreuve
  int iDifficulte; // Difficultéde l'épreuve
  int iNbParticipant; // Nombre de participants inscrits à l'épreuve
  int iEpreuveDeroulee; // Booléen qui indique si une épreuve a eu lieu (vrai) ou non (faux)
  int iRemplissageClassement; // Position à laquelle on s'est arrêté lors de la saisie du classement, égal à 4*iNbParticipant quand classement complet
  Date dateEpreuve; // Date de l'épreuve
  Resultat * resultatEpreuve[NB_MAX_PARTICIPANTS]; // Liste des résultats de l'épreuve
  Participant * listeParticipant[NB_MAX_PARTICIPANTS]; // Liste des participants de l'épreuve
  Manche * ListeManche; // Liste des manches de l'épreuve
  Possession * ListePossession[NB_MAX_PARTICIPANTS]; // Liste des vélos utilisés par les participants durant l'épreuve
}Epreuve;
```

Fonctions utilisées

Fonctions générales

void sPage(int n) ;

Cette fonction permet d'effectuer des sauts de ligne.
Paramètres :

– n : Entier représentant le nombre de saut à effectuer.

Détail de la fonction : une boucle "for" permet d'effectuer "n" saut de lignes.

void etoiles(int n) ;

Cette fonction permet l'affichage d'étoiles afin d'améliorer la présentation des menus.
Paramètres :
– n : Entier représentant le nombre d'étoile à intégrer.
Détail de la fonction : une boule "for" permet l'affichage de "n" étoiles.

void Reinitialiser(Velo ** ListeVelos, int * iNbVelos, int * iIdVelo, Participant ** ListeParticipants, int * iNbParticipants, int * iIdParticipant, Epreuve ** ListeEpreuves, int * iNbEpreuves, int * iIdEpreuve) ;

Procédure permettant de réinitialiser tout le programme. Elle supprime les données crées en mémoire avec des mallocs et réinitialise les différents compteurs. On repart sur une base saine.
Elle reçoit en entrée :

– la liste des vélos (tableau de pointeurs sur Velo)

- un pointeur sur le nombre de vélos existant
- un pointeur sur la valeur de la variable générant un identifiant unique pour chaque vélo ;
- la liste des participants (tableau de pointeurs sur Participant) ;
- un pointeur sur le nombre de participants existant ;
- un pointeur sur la valeur de la variable générant un identifiant unique pour chaque participant ;
- la liste des épreuves (tableau de pointeurs sur Epreuve) ;
- un pointeur sur le nombre de épreuves existant ;
- un pointeur sur la valeur de la variable générant un identifiant unique pour chaque épreuves.

Il n'y a aucune valeur de retour.

void retourMenu() ;

Procédure demandant à l'utilisateur d'appuyer sur la touche « Entrée » avant de revenir au menu précédant.

Il n'y a aucune valeur d'entrée ou de retour.

Fonctions de gestion des structures vélos

int creerVelo(Velo ** velo, int * iIdVelo, char sMarque[], char sType[], char sTailleCadre[], int * iNbVelo) ;

Cette fonction permet de créer des vélos en associant chaque paramètres de la fonction à un champ de la structure Vélo. Le vélo est alloué dynamiquement via un malloc. Si le malloc échoue, la fonction retourne -1 et un message d'erreur. Dans le cas contraire la vélo est crée et chaque champ est associé avec le paramètre correspondant, le nombre de vélo et l'ID sont incrémentés puis la retourne 0.
Liste des paramètres :

- Velo ** velo : Pointeur sur le vélo qui sera créé ;
- int * iIdVelo : ID du vélo, sera incrémenté après la création ;
- char sMarque[] : Marque du vélo qui sera créé ;
- char sType[] : Type du vélo qui sera créé ;
- char sTailleCadre[] : Taille du cadre du vélo qui sera créé ;
- int * iNbVelo : Nombre actuel de vélo, incrémenté au final.

int supprimerVelo(Velo * listeVelos[], int iRangVeloASupprimer, int * iNbVelo, Participant ** ListeParticipants, int iNbParticipants) ;

Cette fonction permet de supprimer le vélo positionné dans la listeVelos[] au rang iRangASupprimer à l'aide d'un free(). Ensuite le pointeur sur ce vélo est mis à NULL et le nombre de vélo est décrémenté. Retourne -1 si un erreur survient lors de la suppression.
Liste des paramètres :

- Velo * listeVelos[] : liste de pointeurs sur tout les vélos ;
- int iRangASupprimer : rang du vélo à supprimer dans la liste ;
- int * iNbVelo : nombre actuel de vélo, décrémenté à la fin.

int rechercheCode(Velo * listeVelos[], int iNbVelo, int iCode) ;

Cette fonction permet de rechercher dans la liste des vélos le vélo identifié par l'identifiant iCode. La fonction parcourt séquentiellement la liste des vélos pour i=0 à iNbVelo-1 La fonction retourne le rang du vélo en question dans la liste des vélos. Si aucun vélo n'est trouvé, la fonction retourne -1.
Liste des paramètres :

- Velo * listeVelos[] : liste de pointeurs sur tout les vélos ;
- int iNbVelo : nombre actuel de vélos ;
- int iCode : ID recherché.

int rechercherVelo(Velo * ListeVelos[], int iNbVelo) ;

Cette fonction permet de rechercher dans la liste des vélos un vélo identifié par son nom ou son id. La fonction parcourt le tableau des vélos et compare le critère demandé avec celui de chaque vélos du tableau. Une fois trouvé, elle retourne le rang du vélo Si aucun n'est trouvé la fonction écrit un message d'erreur et retourne -1.
Liste des paramètres :

- Velo * listeVelos[] : liste de tout les vélos existants ;
- int iNbVelo : nombre actuel de participant pour le parcours de la liste. (de 0 à iNbPart-1).

void afficherVelo(Velo * velo) ;

Cette fonction permet d'afficher les informations du vélo demandé. Retourne -1 si le vélo n'est pas trouvé.
Liste des paramètres :

- Velo * velo : pointeur sur le vélo que l'on veut afficher.

int menuVelo() ;

Cette fonction est une fonction d'affichage d'un menu proposant de créer, afficher, supprimer un vélo. Retourne le choix de l'utilisateur.

int gestionVelo(Velo ** listeVelos, int * iNbVelo, int * iIdVelo, Participant ** ListeParticipants, int iNbParticipants) ;

Cette fonction permet de faire la gestion des vélos via un menu qui propose de créer, supprimer, rechercher ou afficher un vélo.

Fonctions de gestion des structures de manches

Fonctions de création d'instances des structures des manches

- CrossCountry * creerCrossCountry(float fDistance, int iDeniveleNeg, int iDenivelePos) ;
- Descente * creerDescente(int iDeniveleNeg, int iDifficulte) ;
- Trial * creerTrial(int iNbObstacles, int iDifficulte, int iTypeObstacles) ;
- Orientation * creerOrientation(int iDeniveleNeg, int iDenivelePos, int iNbObstacles, char * sCodeIGN) ;
- Manche * creerManche(CrossCountry * ptrCrossCountry, Descente * ptrDescente, Trial * ptrTrial, Orientation * ptrOrientation) ;

Ces fonctions créeent une instance d'une des strucutres de manche. Cette instance est créée dynamiquement et son adresse dans la mémoire dans la structure manche de l'épreuve associée.
Elles reçoivent entrée :

- toutes les informations nécessaires pour remplir les champs de la structure de la manche à créer.

Si la création a réussie, ces fonctions renvoient l'adresse vers la zone mémoire (un pointeur sur la structure) contenant la structure, sinon elles renvoient NULL.

Création complète d'une manche

- Manche * creationManche() ;

Cette fonctions appelle toutes les fonctions de saisie des manches et gère les erreurs éventuelles. C'est à dire, si il y a une erreur sur la saisie d'une manche, toutes sont supprimées pour ne pas avoir d'erreur par la suite.
Elle ne reçoit aucun paramètre en entrée.

Si la création a réussies, elle renvoie l'adresse vers la zone mémoire (un pointeur sur la structure Manche) contenant la structure Manche, sinon elle renvoie NULL.

Fonctions d'affichage d'instances des structures des manches

- void afficherCrossCountry(CrossCountry * ptrCrossCountry) ;
- void afficherDescente(Descente * ptrDescente) ;
- void afficherTrial(Trial * ptrTrial) ;
- void afficherOrientation(Orientation * ptrOrientation) ;
- void afficherManche(Manche * ptrManche) ;

Ces fonctions affiches les informations d'une instance d'une des strucutres de manche.
Elles reçoivent entrée :

- le pointeur sur la structure à afficher.

Il n'y a aucune valeur de retour.

Fonctions de suppression d'instances des structures des manches

- void supprimerCrossCountry(CrossCountry ** ptrCrossCountry) ;
- void supprimerDescente(Descente ** ptrDescente) ;
- void supprimerTrial(Trial ** ptrTrial) ;
- void supprimerOrientation(Orientation ** ptrOrientation) ;
- void supprimerManche(Manche ** ptrManches) ;

Ces fonctions supprimment une instance d'une des strucutres de manche. Elles libèrent simplement l'espace mémoire alloué à la manche indiquée et réinitialisent le pointeur vers celle-ci.
Elles reçoivent entrée :

- l'adresse du pointeur sur la manche à supprimer (car sa valeur est modifiée).

Il n'y a aucune valeur de retour.

Fonctions de saisie des informations pour la création de manche

- CrossCountry * saisieCrossCountry() ;
- Descente * saisieDescente() ;
- Trial * saisieTrial() ;
- Orientation * saisieOrientation() ;

Ces fonctions demandent à l'utilisateur de renseigner un à un tous les éléments nécessaires à la création d'une manche. Elles vérifient ensuite la validité des informations saisies puis les transmettent aux fonctions de création de manches.
Elles ne reçoivent aucun paramètre en entrée.

Si la création a réussie, ces fonctions renvoient l'adresse vers la zone mémoire (un pointeur sur la structure) contenant la structure, sinon elles renvoient NULL.

Fonctions de modification des informations d'une manche

- int modifierCrossCountry(CrossCountry * ptrCrossCountry) ;
- int modifierDescente(Descente * ptrDescente) ;
- int modifierTrial(Trial * ptrTrial) ;
- int modifierOrientation(Orientation * ptrOrientation) ;

Ces fonctions permettent à l'utilisateur de modifier une manche. Elles affichent la manche et demandent champ après champ si l'utilisateur veut apporter des modifications. Toute information saisie est vérifiée avant que la modification soit effective sur la manche. Ces fonctions sont appelées par la fonction de modification d'une épreuve.
Elles reçoivent en entrée :

- un pointeur sur la structure à modifier.

Ces fonctions renvoient un code de retour indiquant si la modification s'est bien déroulée ou non. Ce code est à 0 si tout s'est bien passé.

Fonctions de gestion des structures participants

int creerParticipant(Participant ** part, int * iIdPart, char sNom[], char sPrenom[], int iAge, char sClub[], int * iNbPart) ;

Cette fonction permet de créer des participants en associant chaque paramètres de la fonction à un champ de la structure Participant. Le participant est alloué dynamiquement via un malloc. Si le malloc échoue fonction retourne -1 et un message d'erreur. Dans le cas contraire le participant est crée et chaque champ est associé avec le paramètre correspondant, le nombre de participant et l'ID sont incrémenté puis la fonction retourne 0.
Liste des paramètres :

- Participant ** velo : Pointeur sur le part. qui sera créé ;
- int * iIdPart : ID du part., sera incrémenté après la création ;
- char sNom[] : Nom du part. qui sera créé ;
- char sPrenom[] : Prenom du part. qui sera créé ;
- int iAge : TAge du part. qui sera créé ;
- int * iNbPart : Nombre actuel de participant, incrémenté au final.

int afficherParticipant(Participant * part, Epreuve ** ListeEpreuves, int iNbEpreuves) ;

Cette fonction permet d'afficher le participant demandé. Retourne -1 si le part. n'est pas trouvé.
Liste des paramètres :

- Participant * part : pointeur sur le part. à afficher ;
- Epreuve ** ListeEpreuves : liste des épreuves ;
- int iNbEpreuves : Nombre d'épreuves actuelles.

int modifierParticipant(Participant * part[], Velo * listeVelo[], int * iNbVelo, int * iIdVelo, int iNbPart, int iNbEpreuves, Epreuve * listeEpreuves[]) ;

Cette fonction permet de modifier un participant. Elles appelle la fonction de recherche pour trouver le participant à modifier puis propose de mettre à jour ses informations, champ après champ. Le participant

peut également en profiter pour changer ses vélos ou gérer ses inscriptions aux épreuves. Retourne 0 si la modification s'est bien passée.
Liste des paramètres :

- Participant * part[] : liste des participants ;
- Velo * listeVelo[] : liste des vélos ;
- int * iNbVelo : pointeur sur le nombre de vélos enregistrés ;
- int * iIdVelo : pointeur sur la variable donnant un identifiant unique à chaque vélo ;
- int iNbPart : nombre de participants enregistrés ;
- Epreuve * ListeEpreuves[] : liste des épreuves ;
- int iNbEpreuves : Nombre d'épreuves actuelles.

int supprimerParticipant(Participant * listePart[], int iRangPartASupprimer, int * iNbPart, Epreuve * listeEpreuves[], int iNbEpreuves) ;

Cette fonction permet de supprimer le participant demandé. Elle est réalisée grâce à un free(), puis le pointeur sur ce participant est mis à NULL. Le nombre de participant est ensuite décrémenté. Retourne -1 si une erreur survient lors de la suppression. Un participant ne peut être supprimé que s'il ne participe à aucune épreuve ayant eu lieu (dans ce cas, la désinscription est impossible). Une vérification est effectuée. Le participant est désinscrit de toutes les épreuves avant sa suppression.
Liste des paramètres :

- Participant * listePart[] : liste de tout les part. existants ;
- int iRangASupprimer : rang du part. à supprimer dans la liste ;
- int * iNbPart : nombre actuel de part., décrémenté au final.

int rechercherParticipant(Participant * part[], int iNbPart, Epreuve ** ListeEpreuves, int iNbEpreuves) ;

Cette fonction permet de rechercher dans la liste des parts. un part identifié par son nom ou son id. La fonction parcourt le tableau des parts. et compare le critère demandé avec celui de chaque participants du tableau. Une fois trouvé, elle retourne le rang du part. Si aucun n'est trouvé la fonction écrit un message d'erreur et retourne -1.
Liste des paramètres :

- Participant * part[] : liste de tout les parts. existants ;
- int iNbPart : nombre actuel de participant pour le parcours de la liste. (de 0 à iNbPart-1) ;
- int iNbEpreuves : Nombre d'épreuves actuelles.

int ajouterVelo(Participant * part, Velo * listeVelo[], int * iNbVelo, int * iIdVelo) ;

Cette fonction permet d'ajouter un vélo à la liste des vélos du participant demandé. La fonction demande à l'utilisateur les champs du nouveau vélo puis elle recherche si ce vélo n'existe pas déjà. Si c'est le cas, alors la fonction ajoute le vélop déjà existant à la liste des vélos du participant. Sinon elle appelle la fonction creerVelo() avec les champs demandé si le nombre max de vélos n'est pas atteinds puis ajoute le vélo nouvellement crée dans la liste du participant. Retourne 0 si tout c'est bien passé sinon -1.
Liste des paramètres :

- Participant * part : pointeur sur le part. à qui l'on veut ajouter un vélo. ;
- Velo * listeVelo[] : liste de tout les vélos existants ;
- int * iNbVelo : nombre actuel de vélos, utilisé s'il y a création d'un nouveau vélo via creerVelo() (cf. velo.h) ;
- int * iIdVelo : ID du vélo à créer, utilisé s'il y a création d'un nouveau vélo via creerVelo().

int menuPart() ;

Cette fonction est une fonction d'affichage d'un menu proposant de creer, supprimer, rechercher, modifier, inscrire un participant. Retourne le choix de l'utilisateur.

int gestionPart(Velo * listeVelo[], int * iNbVelo, int * iIdVelo, Participant * listeParticipants[], int * iNbPart, int * iIdPart, Epreuve ** ListeEpreuves, int iNbEpreuves) ;

Cette fonction permet de faire la gestion des participants avec de menuPart() et des différentes fonctions associées.
Liste des paramètres :

- Velo * listeVelo[] : liste des tout les vélos existants ;
- int * iNbVelo : nombre actuel de vélo, utilisé s'il y a une modification de ce nombre (création..) ;
- int * iIdVelo : ID du prochain vélo à créer ;
- Participant * listeParticipants[] : liste de tout les participants existants ;
- int * iNbPart : nombre actuel de participants, utilisé s'il y a modification de ce nombre (création..) ;
- int * iIDPart : ID du prochain participant à créer ;
- Epreuve ** ListeEpreuves : liste des épreuves ;
- int iNbEpreuves : Nombre d'épreuves actuelles.

Fonctions de gestion des structures épreuves

int gestionEpreuve(Epreuve ** ListeEpreuves, int * iNbEpreuves, int * iIdEpreuves, int iNbPart) ;

Menu principal de la partie Epreuve.

void creerEpreuve(Epreuve ** ListeEpreuves, int * iNbEpreuves, int * iIdEpreuves) ;

Cette fonction permet la création d'une nouvelle épreuve.
Paramètres :

- ListeEpreuve : Tableu de pointeurs de type Epreuve ;
- iNbEpreuve : Entier contenant le nombre d'épreuves existantes ;
- ListeManches : Tableu de pointeurs de type Epreuve ;
- ListeCrossCountry : Tableu de pointeurs de type Epreuve ;
- ListeDescente : Tableu de pointeurs de type Epreuve ;
- ListeTrial : Tableu de pointeurs de type Epreuve ;
- ListeOrientation : Tableu de pointeurs de type Epreuve.

Détail de la fonction :

1. Reservation de l'espace mémoire pour la nouvelle épreuve.
2. Intégration de l'identifiant de l'épreuve. La vérification est effectué dans le but d'intrégrer la valeur 1 au premier élément.Cette intégration est faite à partir du nombre d'éléments "épreuve" créés.
3. Remplissage de l'épreuve à l'aide de questions-réponses.
4. Création des manches. remplissage des différentes strucutres "manches".
5. Verification afin d'attester que l'épreuve est bien créée.

void afficherUneEpreuve(Epreuve * eprElement) ;

Cette fonction permet l'affichage d'une épreuve créée. Avant l'affichage de l'épreuve, la fonction effectue une vérification afin de savoir si l'élément envoyé en paramètre existe.
Paramètres :

- Le pointeur vers une épreuve.

void afficherToutesEpreuves(Epreuve ** ListeEpreuves, int * iNbEpreuves) ;

Cette fonction permet l'affichage de toutes les Epreuves créées. Pour chaque epreuve existante, la fonction envoie son pointeur en paramètre de la fonction afficherUneEpreuve. Cette dernière charge les épreuves et gère un tri avant l'affichage.
Paramètres :

- La liste des pointeurs "Epreuves" ;
- Le nombre d'épreuves existantes.

void afficherUneEpreuve(Epreuve * eprElement) ;

Cette fonction permet l'affichage d'une épreuve créée. Avant l'affichage de l'épreuve, la fonction effectue une vérification afin de savoir si l'élément envoyé en paramètre existe.
Paramètres :

- Le pointeur vers une épreuve.

void afficherToutesEpreuves(Epreuve ** ListeEpreuves, int * iNbEpreuves) ;

Cette fonction permet l'affichage de toutes les Epreuves créées. Pour chaque epreuve existante, la fonction envoie son pointeur en paramètre de la fonction afficherUneEpreuve. Cette dernière charge les épreuves et gère un tri avant l'affichage.
Paramètres :

- La liste des pointeurs "Epreuves" ;
- Le nombre d'épreuves existantes.

int rechercherEpreuve(Epreuve ** ListeEpreuves, int * iNbEpreuves) ;

Cette fonction permet l'affichage d'une épreuve. La fonction demande, dans un premier temps le titre de l'épreuve recherchée. Elle stocke ce titre dans la variable "sTitreRecherche". Par la suite, nous effectuons une comparaison de "sTitreRecherche" avec tous les titres existants en parcourant un à un les éléments du tableau de pointeur ListeEpreuve. Une fois l'épreuve trouvée, nous affichons entièrement cette épreuve grâce à la fonction afficherUneEpreuve(). Si bool reste égal à 0, alors l'épreuve recherchée n'existe pas. Nous affichons, donc, un message en conséquence.
Paramètres :

- ListeEpreuves : La liste des pointeurs "Epreuves" ;
- iNbEpreuves : Le nombre d'épreuves existantes ;
- iModification : Prend 0 si la fonction est appellé pour l'affichage simple. Prend 1 si la fonction est appelée pour la modification de l'épreuve.

void modifierEpreuve(Epreuve * eprElement) ;

Cette fonction permet l'affichage d'une épreuve.
Paramètres :

– eprElement : Pointeurs sur l'élément à modifier de type "Epreuves" ;

Détail de la fonction :

1. Dans un premier temps, la fonction procède à la modification des contenus "principaux" de l'épreuve.
2. Ensuite, la fonction procède à la modification des contenus "manches" par appels de fonctions.

void supprimerUneEpreuve(Epreuve ** ListeEpreuves, int * iNbEpreuves, int iRangEpreuveASupprimer) ;

Cette fonction permet la suppression d'une épreuve. Dans un premier temps, la fonction procède à la recherche de l'épreuve à supprimer. Une fois l'élément trouvé, nous intégrons la valeur de chacun des champs de la dernière épreuve dans l'élément à supprimer. A la fin, nous mettons le nombre d'épreuves à "iNbEpreuve-1" afin d'écraser le dernier élément lors de la création d'une nouvelle épreuve.
Paramètres :

– ListeEpreuves : La liste des pointeurs "Epreuves" ;
– iNbEpreuves : Le nombre d'épreuves existantes.

int rechercheCodeRes(Resultat * listeResultat[], int iNbPart, int iCode) ;

Permet de recherche le participant ayant l'ID iCode dans le tableau listeResultat, en le parcourant jusqu'à iNbPart.
Liste des paramètres :

– Resultat * listeResultat : tableau de pointeurs sur Résultat pour faire la recherche ;
– int iNbPart : nombre actuel de participants ;
– int iCode : ID à rechercher dans le tableau.

int afficherClassement(Epreuve * epreuve) ;

Permet de trier puis d'afficher le classement de l'épreuve demandée
Liste des paramètres :

– Epreuve * epreuve : pointeur sur l'épreuve dont on veut afficher le classement.

int saisirClassement(Epreuve * epreuve) ;

Permet de saisir le classement de l'épreuve demandée. La fonction vérifie toujours que l'on ne rentre pas un participant déjà classé. On peut reprendre la saisie du classement là ou on l'a arrêté, en entrant -1 à la place de l'ID. La fonction calcule automatiquement le score en fonction du rang du participant dans la classement
Liste des paramètres :

– Epreuve * epreuve : pointeur sur l'épreuve dont on veut saisir le classement.

Epreuve * rechercheEpreuveParID(Epreuve ** ListeEpreuves, int iNbEpreuves, int iIdEpr) ;

Cette fonction permet la recherche d'une épreuve par identifiant. Cette fonction recherche une épreuve dont l'identifiant lui est donné. Elle retourne le pointeur sur l'épreuve trouvée. Cette fonction est appelée, essentiellement lors de l'inscription d'un participant.
Paramètres :

- ListeEpreuves : La liste des pointeurs "Epreuves" ;
- iNbEpreuves : Le nombre d'épreuves existantes ;
- iIdEpr : L'identifiant de l'épreuve.

Fonctions d'inscription

int inscriptionParticipantEpreuve(Participant * participant, Epreuve * epreuve, Velo * velo) ;

Cette fonction effectue tous les traitement nécessaires pour inscrire un participant à une épreuve. Elle note l'ID de l'épreuve dans la liste des épreuves auxquelles le participant est inscrit et crée les éléments nécessaires dans la structure Epreuve pour rendre l'inscription effective.
Elle reçoit en entrée :

- un pointeur sur le participant à inscrire ;
- un pointeur sur l'épreuve à laquelle le participant s'inscrit ;
- un pointeur sur le vélo que le participant va utiliser pour l'épreuve.

Elle retourne 0 si l'inscription s'est bien passée, une autre valeur entière sinon.

int desinscriptionParticipantEpreuve(Participant * participant, Epreuve * epreuve) ;

Cette fonction effectue tous les traitement nécessaires pour désinscrire un participant d'une épreuve. Elle "efface" tout ce qui a été créé par la fonction d'inscription.
Elle reçoit en entrée :

- un pointeur sur le participant à désinscrire ;
- un pointeur sur l'épreuve à laquelle le participant se désinscrit.

Elle retourne 0 si la désinscription s'est bien passée, une autre valeur entière sinon.

int choixInscriptionParticipantAEpreuves1(Participant * participant, Epreuve ** ListeEpreuves, int iNbEpreuves) ;

Cette fonction permet de choisir à quelle épreuve inscrire un participant donné. Elle affiche la liste des épreuves auxquelles il peut s'inscrire (celles où il n'est pas déjà inscrit et qui n'ont pas encore eu lieu). Ensuite, le participant choisit l'épreuve qu'il veut ainsi que le vélo qu'il utilisera (parmi ceux qu'il possède). Enfin, toutes ces données sont transmises à la fonction d'inscription.
Elle reçoit en entrée :

- un pointeur sur le participant à inscrire ;
- la liste des épreuves ;
- le nombre d'épreives.

Elle retourne 0 si l'inscription s'est bien passée, une autre valeur entière sinon.

int choixDesinscription1(Participant * participant, Epreuve ** ListeEpreuves, int iNbEpreuves) ;

Cette fonction permet de choisir à quelle épreuve désinscrire un participant donné. Elle affiche la liste des épreuves auxquelles il est inscrit (et qui n'ont pas encore eu lieu) et le participant choisit celle à laquelle il ne veut plus participer. Enfin, toutes ces données sont transmises à la fonction de désinscription.
Elle reçoit en entrée :

- un pointeur sur le participant à inscrire ;
- la liste des épreuves ;
- le nombre d'épreives.

Elle retourne 0 si l'inscription s'est bien passée, une autre valeur entière sinon.

Fonctions de classement

Resultat * classementSaison(Epreuve * listeEpreuve[], Participant * listeParticipant[], int iNbEpreuve,int iNbParticipant, Resultat Classement[], int * iTaille) ;

Cette fonction permet de créer le classement de la saison. La fonction créé un modèle de résultat modRes[] contenant tout les participants existants avec des résultats initialisés à 0. La fonction créé dynamiquement le tableau de résultat Classement qui contiendra tout les résultats de tout les participants. Le tableau contiendra des sous tableaux pour chaque épreuve et chaque sous tableau contiendra autant de tableau résultat qu'il y a de participants au total. Ensuite la fonction compare le tableau de la première épreuve avec le modèle de résultat. Chaque résultat de chaque participant de l'épreuve est copié dans le tableau Classement, puis ceux ne participant pas sont recherchés dans le modèle et copié dans chaque espaces restant du premier sous tableau avec le score de 50 * la difficulté de l'épreuve. À partir de la seconde épreuve, chaque tableau de résultat est comparé avec celui de la première épreuve en effectuant les mêmes étapes que précédemment. Une fois que le tableau Classement est rempli, la fonction va rechercher et supprimer chaque doublon de participants dans le tableau en copiant chacun des scores dans le premier résultat du participant. Lorsqu'il ne reste que les doublons du dernier participant à supprimer, on supprime simplement toutes les cases restantes du tableau. Finalement, la fonction réalise un tri par sélection du minimum en fonction du score. Ainsi le premier de la liste sera le premier du classement de la saison.
Liste des paramètres :

- Epreuve * listeEpreuve[] : liste de toutes les épreuves existantes ;
- Participant * listeParticipant[] : liste de tous les parts. existants ;
- int iNbEpreuve : nombre actuel d'épreuves ;
- int iNbParticipant : nombre actuel de participans ;
- Resultat Classement[] : tableau de résultat qui contiendra le classement retourné en fin de fonction ;
- int * iTaille : taille du tableau Classement retourné en fin de fonction.

int afficherClassementSaison(Resultat Classement[], int iTaille) ;

Cette fonction affiche le classement de la saison.
Liste des paramètres :

- Resultat Classement : tableau contenant les résultats de la saison triés ;
- int iTaille : taille de ce tableau.

Fonctions de sauvegarde

int Sauvegarde(Velo ** ListeVelos, int iNbVelos, int iIdVelo, Participant ** ListeParticipants, int iNbParticipants, int iIdParticipant, Epreuve ** ListeEpreuves, int iNbEpreuves, int iIdEpreuve) ;

Fonction de sauvegarde. Elle parcours toutes les structures principales (vélo, participant, épreuve) afin d'en sauvegarder les informations essentielles permettant de régénérer toutes ces structures avec les même valeurs. La sauvegarde se fait via la création d'un fichier texte écrit selon un modèle précis et permettant de savoir exactement quel type d'information se trouve à un endroit donné (à condition que le fichier ne soit pas corrompu).
Elle reçoit en entrée :

- la liste des vélos (tableau de pointeurs sur Velo) ;
- le nombre de vélos existant ;
- la valeur de la variable générant un identifiant unique pour chaque vélo ;
- la liste des participants (tableau de pointeurs sur Participant) ;
- le nombre de participants existant ;
- la valeur de la variable générant un identifiant unique pour chaque participant ;
- la liste des épreuves (tableau de pointeurs sur Epreuve) ;
- le nombre de épreuves existant ;
- la valeur de la variable générant un identifiant unique pour chaque épreuves.

Elle retourne une valeur entière indiquant si la sauvegarde s'est bien déroulée (0) ou non (valeur entière différente de 0).

int charger(Velo ** ListeVelos, int * iNbVelos, int * iIdVelo, Participant ** ListeParticipants, int * iNbParticipants, int * iIdParticipant, Epreuve ** ListeEpreuves, int * iNbEpreuves, int * iIdEpreuve) ;

Fonction permettant de charger une sauvegarde du programme. Elle petit à petit le fichier texte contenant la sauvegarde afin de s'en servir pour créer toutes les structures nécessaires et les initaliser aux bonnes valeurs. On ne teste pas l'intégrité du fichier, c'est à dire et conforme à celle prévue. On teste en revanche chaque valeur lue pour s'assurer qu'elle est bien dans les limites fixées. Ainsi, on se protège des erreurs résultant de la modification manuelle des valeurs du fichier de sauvegarde (si des informations sont erronées, on fait le nécessaire pour que ça n'empêche pas le bon fonctionnement du programme. Cette fonction a été découpée en plusieurs sous-fonctions qu'elle appelle pour ne pas avoir une fonction de 800 lignes -¿ rend le code plus lisible.
Elle reçoit en entrée :

- la liste des vélos (tableau de pointeurs sur Velo) ;
- un pointeur sur le nombre de vélos existant ;
- un pointeur sur la valeur de la variable générant un identifiant unique pour chaque vélo ;
- la liste des participants (tableau de pointeurs sur Participant) :
- un pointeur sur le nombre de participants existant :
- un pointeur sur la valeur de la variable générant un identifiant unique pour chaque participant ;
- la liste des épreuves (tableau de pointeurs sur Epreuve) ;
- un pointeur sur le nombre de épreuves existant ;
- un pointeur sur la valeur de la variable générant un identifiant unique pour chaque épreuves.

Elle retourne une valeur entière indiquant si le chargement s'est biendéroulé (0) ou non (valeur entière différente de 0)

int chargerVelos(Velo ** ListeVelos, int * iNbVelos, int * iIdVelo, int * iSauvegardeCorrompue, FILE * fichier) ;

Fonction permettant de charger les vélos enregistrés dans un fichier de sauvegarde du programme. Elle lit le fichier pas à pas afin de créer en mémoire toutes les structures correspondant aux vélos stockés dans le fichier.
Elle reçoit en entrée :

- la liste des vélos (tableau de pointeurs sur Velo) ;
- un pointeur sur le nombre de vélos existant ;
- un pointeur sur la valeur de la variable générant un identifiant unique pour chaque vélo ;
- un pointeur sur le fichier de sauvegarde contenant les vélos à charger.

Elle retourne une valeur entière indiquant si le chargement s'est bien déroulé (0) ou non (valeur entière différente de 0).

int chargerParticipants(Velo ** ListeVelos, int * iNbVelos, Participant ** ListeParticipants, int * iNbParticipants, int * iIdParticipant, int * iSauvegardeCorrompue, FILE * fichier) ;

Fonction permettant de charger les participants enregistrés dans un fichier de sauvegarde du programme. Elle lit le fichier pas à pas afin de créer en mémoire toutes les structures correspondant aux participants stockés dans le fichier.
Elle reçoit en entrée :

- la liste des vélos (tableau de pointeurs sur Velo) ;
- un pointeur sur le nombre de vélos existant ;
- la liste des participants (tableau de pointeurs sur Participant) ;
- un pointeur sur le nombre de participants existant ;
- un pointeur sur la valeur de la variable générant un identifiant unique pour chaque participant ;
- un pointeur sur le fichier de sauvegarde contenant les participants à charger.

Elle retourne une valeur entière indiquant si le chargement s'est bien déroulé (0) ou non (valeur entière différente de 0).

int chargerEpreuves(Velo ** ListeVelos, int * iNbVelos, Participant ** ListeParticipants, int * iNbParticipants, int * iIdParticipant, Epreuve ** ListeEpreuves, int * iNbEpreuves, int * iIdEpreuve, int * iSauvegardeCorrompue, FILE * fichier) ;

Fonction permettant de charger les épreuves enregistrés dans un fichier de sauvegarde du programme. Elle lit le fichier pas à pas afin de créer en mémoire toutes les structures correspondant aux épreuves stockées dans le fichier. Elle ne charge que les informations principales des épreuves et fait appel à d'autres fonctions pour charger les informations concernant les manches de l'épreuve et les participants inscrits (permet de découper la fonction qui serait trop longue sinon).
Elle reçoit en entrée :

- la liste des vélos (tableau de pointeurs sur Velo) ;
- un pointeur sur le nombre de vélos existant ;
- la liste des participants (tableau de pointeurs sur Participant) ;
- un pointeur sur le nombre de participants existant ;
- la liste des épreuves (tableau de pointeurs sur Epreuve) ;
- un pointeur sur le nombre de épreuves existant ;
- un pointeur sur la valeur de la variable générant un identifiant unique pour chaque épreuves ;
- un pointeur sur le fichier de sauvegarde contenant les épreuves à charger.

Elle retourne une valeur entière indiquant si le chargement s'est bien déroulé (0) ou non (valeur entière différente de 0).

int chargerManchesEpreuves(Epreuve ** ListeEpreuves, int * iNbEpreuves, int * iIdEpreuve, int * iSauvegardeCorrompue, FILE * epreuves) ;

Fonction permettant de charger les manches d'une épreuves enregistrés dans un fichier de sauvegarde du programme. Elle lit le fichier pas à pas afin de créer en mémoire toutes les structures correspondant aux manches des épreuves stockées dans le fichier.
Elle reçoit en entrée :

- la liste des épreuves (tableau de pointeurs sur Epreuve) ;
- un pointeur sur le nombre de épreuves existant ;
- un pointeur sur la valeur de la variable générant un identifiant unique pour chaque épreuves à charger.

Elle retourne une valeur entière indiquant si le chargement s'est bien déroulé (0) ou non (valeur entière différente de 0).

int chargerParticipantsEpreuves(Velo ** ListeVelos, int * iNbVelos, Participant ** ListeParticipants, int * iNbParticipants, int * iIdParticipant, Epreuve ** ListeEpreuves, int * iNbEpreuves, int * iSauvegardeCorrompue, FILE * epreuves) ;

Fonction permettant de charger les participants inscrits à une épreuves et enregistrés dans un fichier de sauvegarde du programme. Elle lit le fichier pas à pas afin de créer en mémoire toutes les structures correspondant aux participants inscrit à l'épreuve dans le fichier.
Elle reçoit en entrée :

- la liste des vélos (tableau de pointeurs sur Velo) ;
- un pointeur sur le nombre de vélos existant ;
- la liste des participants (tableau de pointeurs sur Participant) ;
- un pointeur sur le nombre de participants existant ;
- la liste des épreuves (tableau de pointeurs sur Epreuve) ;
- un pointeur sur le nombre de épreuves existantes ;
- un pointeur sur le fichier de sauvegarde contenant les épreuves à charger.

Elle retourne une valeur entière indiquant si le chargement s'est bien déroulé (0) ou non (valeur entière différente de 0).

void lectureLigne(char * sBufferChaine, int iNbCaracteresALire, FILE * ptrFichier) ;

Procédure permettant de lire une ligne dans un fichier texte. Elle lit le contenu d'une ligne (tout en limitant le nombre de caractères lus) et le stocke dans une chaîne de caractère. Puis, elle teste si la ligne a été entièrement lue ou non. Si oui, elle remplace le retour à la ligne par la fin de chaîne. Sinon, elle lit la fin de la ligne sans la conserver afin que la prochaine lecture se fasse sur une toute nouvelle ligne.
Elle reçoit en entrée :

- la chaîne de caractère devant recevoir les données ;
- le nombre de caractères maximum à lire ;
- un pointeur vers le fichier à lire.

Elle ne retourne aucune valeur, la chaîne de caractères étant directement modifiée.

Organisation

Pour réaliser le projet, nous nous sommes d'abord réuni tous ensembles pour discuter de ce que l'on voulait faire et des structures nécessaires. Nous avons donc élaboré une liste des structures à créer en détaillant chacun de leur champ et leur utilité. Une fois ce premier travail fait, nous nous sommes

réparti la création de ces structures ainsi que de toutes les fonctions nécessaires à leur manipulation (création/modification/affichage/suppression/recherche). Cette première répartition fut :

- Jean-Baptiste : CrossCountry, Descente, Trial, Orientation, Manche ;
- Dorian : Participant, Velo ;
- Matthieu : Date, Resultat, Epreuve.

Une fois ce travail effectué, nous avons défini ensemble toutes les fonctionnalités voulues pour le programme. Durant le développement, nous nous répartissions le travail au fur et à mesure, selon l'avancement de chacun. C'est à dire, dès que quelqu'un terminait le travail qui lui était confié, il entamait autre chose. Par conséquent, la répartition n'était pas totalement équitable (ceux qui allaient plus vite s'occupaient de plus de choses) mais a permit de terminer le projet rapidement en s'adaptant au temps libre de chacun. On évitait ainsi de prendre du retard en attendant que l'on ait tous fini notre partie avant de passer à la suite.

Matthieu s'est occupé des fonctions correspondant aux épreuves (gestion des structures et tri). Dorian s'est occupé du module concernant les vélos et les participants. Il a également créé les fonctions de saisie, calcul et affichage des classements. Enfin, Jean-Baptiste a créé les fonctions de gestion des manches, d'inscription et de sauvegarde.

En plus de cette organisation, nous nous entraidions quand cela était nécessaire. Ainsi, on ne s'est pas limité à nos parties mais collaborions pour que chaque pièce du puzzle soit fonctionnelle.

Avancement

Le programme est terminé dans le sens où toutes les fonctionnalités demandées sont implémentées. Cependant, il y aurait encore beaucoup à faire pour améliorer l'ergonomie et la souplesse du programme. Nous avons géré les erreurs possibles de l'utilisateur (comme une double inscription à une épreuve) de manière à ne conserver l'intégrité des données saisies. Pour cela, nous sommes assez dirigistes et empêchons par exemple de désinscrire un participant à une épreuve ayant déjà eu lieu (dont le classement a été au moins partiellement saisi) car un tel acte fausserait le classement. Mais on peut imaginer un espace administrateur protégé par mot de passe où l'utilisateur disposerait de tous les droits, rendant le programme plus souple. Cela n'a pas été fait par manque de temps. Nous préférions nous concentrer sur la qualité des choses développées pour avoir un programme fiable.

Ce programme permet de gérer plusieurs saisons de la compétition, dans la mesure où l'on demande un nom à la sauvegarde. On peut ainsi créer une sauvegarde par saison.