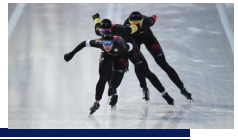
UNIVERSITÉ PARIS DESCARTES



IAP - Introduction à l'Algorithmique et à la Programmation

Equipe pédagogique
Marie-José Caraty, Denis Poitrenaud, Julien Rossit,
Camille Kurtz, Jacques Alès-Bianchetti, Denis Jeanneau

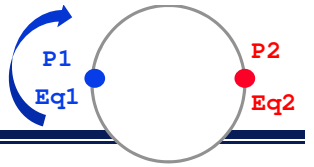
Cours Projet

Les fondamentaux de la programmation impérative

Analyse du projet

http://www.tutorialspoint.com/c_standard_library/

Le problème posé



Objectif Programmer un *interpréteur de commandes*
pour la *gestion d'une compétition de poursuites par équipes*

Dans la **poursuite par équipes**,
deux équipes (`Eq1` et `Eq2`) formées chacune de trois athlètes
doivent faire un nombre donné (`t`) de tours

Le terrain de la compétition est une piste circulaire admettant
deux points diamétralement opposés (`P1` et `P2`) d'où partent
respectivement les trois patineurs des deux équipes (`Eq1` et `Eq2`)
engagées dans la poursuite

A chaque tour de piste (passage aux points `P1` de l'équipe `Eq1`
et au point `P2` de l'équipe `Eq2`), le temps réalisé par l'équipe
est celui obtenu par le dernier patineur de l'équipe passant
le point de référence de l'équipe

Une **compétition de poursuites par équipes**
consiste à organiser *n poursuites par équipes*
pour les *2 x n équipes* engagées dans la compétition

Le problème posé

Objectif Programmer un interpréteur de commandes
pour la gestion d'une compétition de poursuites par équipes

Les commandes sont codées sous forme de chaînes de caractères
et entrées en utilisant l'entrée standard
ou par redirection d'un fichier texte sur l'entrée standard

Dix commandes : neuf commandes de gestion de la compétition
et une commande de sortie de l'interpréteur (`exit`)

```
exit
definir_parcours
definir_nombre_épreuves
inscrire_equipe
afficher_equipes
enregistrer_temps
detection_fin_poursuite
detection_fin_competition
afficher_temps
afficher_temps_equipes
```

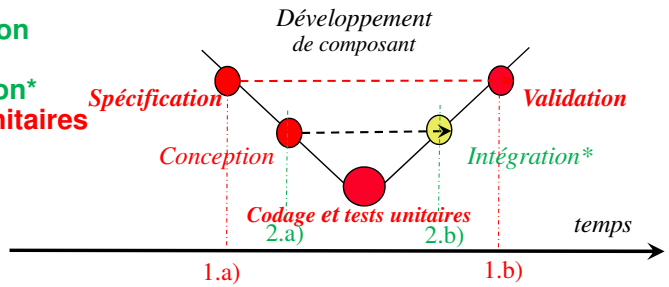

Développement de l'application à base de Sprints

des cycles de développement
appliqués à une fonctionnalité incrémentale
de l'application

Cycle de développement logiciel

Quelque soit la taille de l'application considérée,
spécification, conception, codage et validation sont les phases types de tout cycle de développement logiciel pour aboutir à un code fiable, efficace et de qualité

1. a) Spécification
b) Plan de Validation
2. a) Conception
b) Plan d'intégration*
3. Codage et tests unitaires
4. Intégration*
5. Validation



Les différentes phases du cycle en V sont assimilables à une décomposition en sous-problèmes. Spécification et Conception permettent de concevoir des tests et leurs résultats attendus. Seule la réussite des tests en phase d'intégration puis de validation permet de valider le composant.

Développement par Sprints (1/2)

Phase de développement

Deux fichiers `inSp#n.txt` et `outSp#n.txt` sont donnés pour un `Sprint#n` sur COMMUN ($1 \leq n \leq 5$)
Le premier fichier est un Jeu de Données de Test (JDT), il correspond à des entrées de l'application à tester.
Le second correspond aux sorties attendues de l'application : résultats de référence qui doivent coïncider avec le résultat de votre `run` (exécution) si votre programme fonctionne correctement

En fin de la phase de développement du `Sprint#n`, vous comparerez les sorties de votre programme (`run.txt`) résultant de l'exécution de votre application à partir du jeu de données de test `inSp#n.txt` avec le résultat de référence donné (`outSp#n.txt`)

Pour comparer les deux solutions, vous utiliserez un comparateur de fichiers texte (cf. **Diff.jar,** archive de l'exécutable d'un programme Java de test d'identité de fichiers)
si les deux fichiers (`run.txt` et `outSp#n.txt`) sont identiques,
votre application est 0-défaut pour ce test,
vous pouvez passer au développement du Sprint suivant
sinon corrigez les erreurs de votre application

Développement par Sprints (2/2)

En référence à la programmation agile pour lequel le cycle de développement (analyse fonctionnelle, spécification, codage, test) est court, des Sprints vous sont proposés pour le développement de votre projet.

(1) Chaque Sprint validé correspond à un très bon travail que vous pouvez démontrer. Vous devez en garder la trace : conservez les sources intactes au niveau de votre solution Projet.

Le plus haut sprint (#n, avec 1<=n<=5) validé en phase de recette, sera la base de votre notation de sprint (NS), ses sources seront ajoutées à votre Dossier de Programmation pour une évaluation fondée sur la qualité de code.

Lorsque vous passez à un autre Sprint, créez un nouveau projet où **vous recopierez au niveau des répertoires** toutes les entités logicielles (validées) du précédent projet qui sont nécessaires au nouveau sprint ou à y adapter.

(2) Dans le cas où votre programme ne fonctionne pas sur l'ensemble des fonctionnalités demandées du sprint, conservez ces sources pour un éventuel bonus lié à ce développement.

Données nominales

Les données nominales sont des données « normales »
c'est-à-dire valides au sens où elles sont sans erreur

Relativement au type,
Relativement aux différentes spécifications des commandes (longueur autorisée pour les chaînes de caractères, domaine de variation des paramètres)

Cohérence de l'ensemble des paramètres de commande

Pour ce projet, les Jeux de Données de Test (JDT) des 5 sprints seront constitués de données nominales
On ne vérifiera donc pas la cohérence des données de JDT

Les constantes du problème

Quelques constantes du problème

- Le nombre maximum de tours : 10
- Le nombre maximum d'épreuves : 16
- Le nombre de patineurs par équipe : 3
- Le nombre d'équipes par épreuves : 2

Les constantes du problème – Leur déclaration

Pour leur déclaration, on utilisera les directives du préprocesseur

```
#define MaxTours 10 // Nombre maximum de tours
#define maxEpreuves 16 // Nombre maximal d'épreuves
...
```

Définir les autres constantes
- le nombre de patineurs par équipe (3)
- le nombre d'équipes par épreuve (2)
- taille maximale d'une chaîne de caractère lue (30)

Ces directives sont appelés macros (elles peuvent exprimer un calcul)

Le traitement du préprocesseur est alors de remplacer dans le source toute occurrence
- du mot **`MaxTours`** par **`10`** (littéral, constante entière)
- du mot **`maxEpreuves`** par **`16`**, `etc.`

Rem : Le tableau de caractères (`mot`) permettant de stocker la chaîne de `lgMot` caractères est déclaré :

```
char mot[lgMot+1] // +1 pour le code de la fin de chaîne '\0'
```


Les sprints de l'application

Cinq sprints représentant
cinq incréments de fonctionnalité de l'application

Sprint #1 – inSp1.txt et outSp1.txt

inSp1

```
inscrire_equipe Canada Blondin Weidemann Morrison
inscrire_equipe Japon Takagi Sato Takagu
afficher_equipes
exit
```

outSp1

```
inscription dossard 101
inscription dossard 102
inscription dossard 103
inscription dossard 104
inscription dossard 105
inscription dossard 106
Canada Blondin 101 Weidemann 102 Morrison 103
Japon Takagi 104 Sato 105 Takagu 106
```

Développement par sprints – Sprint #1

Créez un projet de nom `Sprint1` au sein d'une `Solution/Projet` nommée par exemple `Competition`

Premier Sprint

- **Analyse fonctionnelle :** Limitation à des données de test nominales.
Implémentez les commandes "`inscrire_equipe`", "`afficher_equipes`" et "`exit`".
- **Spécification :** Définissez les types/structures de données nécessaires : `Patineur`, `Equipe` (~~`Epreuve et Competition`~~)
Développez la fonctionnalité associée aux commandes :
-mémoriser lors de l'inscription les attributs d'un patineur, -afficher les attributs des patineurs inscrits à la compétition dans leur ordre d'inscription suivant le format spécifié, -`exit` la sortie de l'interpréteur de commande.
- **Codage :** Prototypez et codez (a) la fonction `inscription_equipe` à partir des champs d'information de la commande, (b) la fonction `affichage_equipes` qui affiche les patineurs dans leur ordre d'inscription et (c) le `main()` : boucle sur les trois commandes "`inscription_equipe`", "`affichage_equipe`" et "`exit`".
- **Test :** Testez votre application par redirection des entrées à partir du fichier `inSp1.txt (JDT du Sprint#1)` et sa sortie vers le fichier `run.txt`.
Comparez votre fichier `run.txt` au fichier des résultats de référence `outSp1.txt` : si les deux fichiers coïncident, votre `Sprint#1` est validé, vous pouvez passer au `Sprint#2` sinon corrigez les erreurs.

Développement par sprints – Sprint #1

C4. Commande d'inscription d'une équipe
Une ligne composée de la chaîne de caractères "**`inscription_equipe`**" suivie du nom du pays et des trois noms des membres de l'équipe.
Les numéros de dossard commencent à 101 et seront automatiquement attribués par programme dans l'ordre séquentiel d'inscription.
Le nombre maximal d'équipes est de 32 (16 épreuves au maximum).
La commande affiche sur la sortie standard "inscription dossard" suivie des trois numéros de dossards attribués.

C5. Commande d'affichage des équipes de l'épreuve de poursuite
Une ligne composée de la chaîne de caractères "**`affichage_equipes`**".
Cette commande provoquera l'affichage de deux lignes (une par équipe) correspondant aux deux équipes de l'épreuve de poursuite en cours.
Chaque ligne sera composée du nom du pays suivi des noms et numéros de dossard des membres de l'équipe. L'ordre d'affichage est celui de l'enregistrement des équipes.

C1. Commande de sortie du programme
Une ligne composée de la chaîne de caractères "**`exit`**".

Le programme principal

`main` du `Sprint #1` à coder
Boucle de traitement des commandes se terminant à la commande « `exit` »
À compléter de sprint en sprint jusqu'au Sprint #5
Pour identifier la commande, vous utiliserez la fonction **`strcmp`** de comparaison des chaînes de caractères (cf. <stdlib.h>)

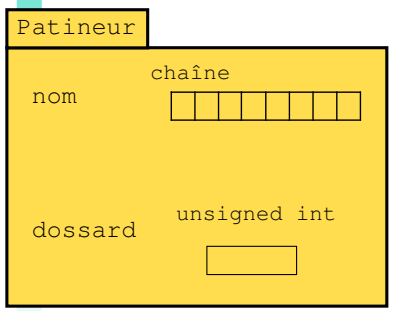
```
int main() {
   …
   while (1) { // ou do {…} while(1);

       // Lecture de la commande (mot)
       scanf("%s", mot);
       // si la commande est "inscrire_equipe"
       //    appeler la fonction inscription_equipe
          …
       // si la commande est "exit"          .
       if (strcmp(mot, "exit")==0) {
         exit(0); // sortie du programme principal
       }
   }
   system("pause"); return 0;
}
```


Structuration des données – Les données en mémoire 2/3

Le type `Patineur`

```
typedef struct {
   char nom[lgMot+1];
   unsigned int dossard;
} Patineur;
```



Type à introduire au Sprint#1

Accès aux attributs (champs) d'un patineur
Si `p` est une variable de type Patineur
`Patineur p;`
on utilise la notation pointée ". " pour accéder aux différents champs. Exemple :

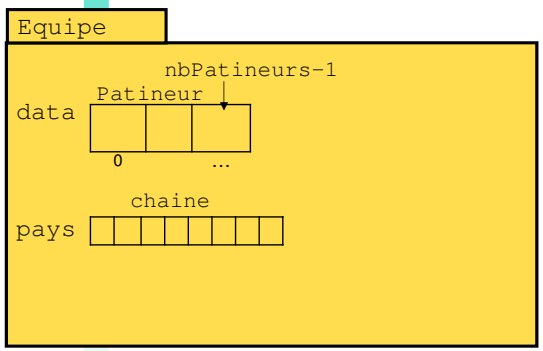
```
p.nom      // accès au nom de p
p.dossard // accès au dossard de p
```

Si p est un pointeur sur un patineur
`Patineur* p; //ou encore Patineur *p;`
on utilise la notation flèche "-> "

```
p->pays    // accès au pays de p
p->dossard // accès au dossard de p
```

Sprint#1

Structuration des données – Les données en mémoire 3/3
Le type Equipe




```
typedef struct {
   ...
   ...
} Equipe;
//à compléter
```

Accès aux attributs du 2ème patineur de l'équipe `e`

```
e.data[1].nom   // accès à son nom
e.data[1].dossard //accès dossard
```

Type à introduire au Sprint#1

Sprint#1

Sprint#1 – Prototypage des fonctions

Sprint#1

```
void inscription_equipe(Inscrits* ins);
void affichage_equipe(const Inscrits* ins);

// Rappel :
// void inscription_equipe(Inscrits ins);
// est un prototype possible
// néanmoins, ce prototype n'est pas optimisé en raison
// de la copie du paramètre effectif dans le paramètre formel
// à l'appel de la fonction inscription_equipe
// on préfèrera passer l'adresse de ins (réduisant un appel
// à la copie de 4 octets)
// Idem pour affichage_equipes, auquel on ajoutera const
// pour éviter toute modification du paramètre d'entrée (ins)
// dans la fonction
```

Spécification : les champs d'information d'une commande sont lus dans la fonction de traitement de la commande

Sprint#2

Sprint #2 – inSp2.txt et outSp2.txt

inSp2

```
inscrire_equipe Canada Blondin Weidemann
Morrison
inscrire_equipe Japon Takagi Sato Takagu
enregistrer_temps 101 1 53.1
enregistrer_temps 102 1 53.2
enregistrer_temps 104 1 53.3
enregistrer_temps 105 1 53.7
enregistrer_temps 106 1 53.9
enregistrer_temps 103 1 54.1
enregistrer_temps 105 2 100.6
afficher_temps 102
afficher_temps 105
exit
```

outSp2

```
inscription dossard 101
inscription dossard 102
inscription dossard 103
inscription dossard 104
inscription dossard 105
inscription dossard 106
Canada 1 Weidemann 53.2
Japon 1 Sato 53.7
Japon 2 Sato 100.6
```


Sprint#2

Développement par sprints – Sprint #2 (1/3)

Créez un projet de nom `Sprint2` au sein de la même Solution que le `Sprint1` : `Competition`. Au niveau des répertoires des projets, recopiez le source du `Sprint1` dans le répertoire des sources de `Sprint2`. Adaptez le source au `Sprint#2`.

Deuxième Sprint

- **Analyse fonctionnelle :** Limitation à des données de test nominales. Implémentez les commandes " `enregistrer_temps`" et "`afficher_temps`"
- **Spécification :** Définissez les structures de données pour l'application : `Mesure` et `Course`. Développez la fonctionnalité associée aux deux nouvelles commandes : mémoriser un temps de chronométrage de la course et afficher tous les temps de chronométrage enregistrés pour un patineur repéré par son dossard dans l'ordre chronologique des temps.
- **Codage :** Prototypez et codez (a) la fonction `enregistrement_temps` et (b) la fonction `affichage_temps`, (b) mettre à jour le `main()` avec ces deux autres commandes. Les champs d'information d'une commande sont lus dans la fonction de traitement de la commande.
- **Test :** Testez votre application par redirection des entrées à partir du fichier `inSp2.txt (JDT du Sprint#2)` et sa sortie vers le fichier `run.txt`. Comparez votre fichier `run.txt` au fichier des résultats de référence `outSp2.txt` : si les deux fichiers coïncident, votre `Sprint#2` est validé, vous pouvez passer au `Sprint#3` sinon corrigez les erreurs.

Développement par sprints – Sprint #2 (2/3)

C6.Commande d'enregistrement d'un temps chronométré pour un nombre de tours de piste effectué
Une ligne composée de la chaîne de caractères "`enregistrer_temps`" suivie du numéro de dossard, du nombre de tours de piste effectué et du temps chronométré (un réel). Les numéros de dossard sont ceux des patineurs participant à l'épreuve de poursuite en cours.

Il n'y a pas d'affichage sur la sortie standard. Les enregistrements sont faits dans l'ordre chronologique.

C7.Commande d'affichage des temps chronométrés d'un patineur
Une ligne composée de la chaîne de caractères "`afficher_temps`" suivie du numéro du dossard. Cette commande donne les temps chronométrés disponibles dans les enregistrements pour le patineur. Elle affiche sur la sortie standard, une ligne pour chaque temps chronométré dans l'ordre chronologique. Chaque ligne a la forme suivante : nom du pays, numéro du tour effectué, nom du patineur et temps chronométré sous forme d'un réel avec une précision d'un chiffre après la virgule.

Sprint#2 – Prototypage des fonctions (3/3)

Sprint#2

```
void enregistrement_temps(Course* crs);
void affichage_temps(const Course* crs);
```

Spécification : les champs d'information d'une commande sont lus dans la fonction de traitement de la commande

.

Sprint #3 – inSp3.txt et outSp3.txt

inSp3

```
inscrire_equipe Canada Blondin Weidemann
Morrison
inscrire_equipe Japon Takagi Sato Takagu
enregistrer_temps 101 1 53.1
enregistrer_temps 102 1 53.2
enregistrer_temps 104 1 53.3
enregistrer_temps 105 1 53.7
enregistrer_temps 106 1 53.9
enregistrer_temps 103 1 54.1
enregistrer_temps 105 2 100.6
afficher_temps_equipes
exit
```

outSp3

```
inscription dossard 101
inscription dossard 102
inscription dossard 103
inscription dossard 104
inscription dossard 105
inscription dossard 106
Canada 54.1
Japon 53.9
```


Développement par sprints – Sprint #3 (1/3)

Créez un projet de nom `Sprint3` au sein de la même Solution que les projets `Sprint1` et `Sprint2` : `Competition`. Au niveau des répertoires des projets, recopiez le source du `Sprint2` dans le répertoire des sources du `Sprint3`. Adaptez le source au `Sprint#3`.

Troisième Sprint

- **Analyse fonctionnelle :** Limitation à des données de test nominales. Implémentez la commande "`afficher_temps_equipes` "
- **Spécification :** Développez la fonctionnalité associée à la nouvelle commande : affichage du temps du dernier patineur de chaque équipe.
- **Codage :** Prototypez et codez (a) la fonction `afficher_temps_equipes` (b) mettre à jour le `main()` avec cette commande. Les champs d'information d'une commande sont lus dans la fonction de traitement de la commande.
- **Test :** Testez votre application par redirection des entrées à partir du fichier `inSp3.txt (JDT du Sprint#3)` et sa sortie vers le fichier `run.txt`. Comparez votre fichier `run.txt` au fichier des résultats de référence `outSp3.txt` : si les deux fichiers coïncident, votre `Sprint#3` est validé, vous pouvez passer au `Sprint#4` sinon corrigez les erreurs.

Développement par sprints – Sprint #3 (2/3)

C8. Commande d'affichage du temps des équipes pour un nombre de tour donné
Une ligne composée de la chaîne de caractères "`afficher_temps_equipes`" suivie du nombre de tours donné. Cette commande affiche pour les deux équipes engagées dans l'épreuve de poursuite le nom du pays suivi du temps effectué par le dernier patineur de cette équipe. L'ordre d'affichage est celui de l'inscription des équipes. Dans le cas où ce dernier patineur n'a pas encore effectué le nombre de tours donné, « `indisponible` » est affiché.

Sprint#3 – Prototypage des fonctions (3/3)

Sprint#3

```
void afficher_temps_equipes(Course* crs);
```

Spécification : les champs d'information d'une commande sont lus dans la fonction de traitement de la commande

Sprint #4 – inSp4.txt et outSp4.txt

inSp4

```
definir_parcours 2
inscrire_equipe Canada Blondin Weidemann Morrison
inscrire_equipe Japon Takagi Sato Takagu
enregistrer_temps 101 1 53.1
enregistrer_temps 102 1 53.2
enregistrer_temps 104 1 53.3
enregistrer_temps 105 1 53.7
enregistrer_temps 106 1 53.9
enregistrer_temps 103 1 54.1
enregistrer_temps 105 2 100.6
enregistrer_temps 106 2 101.7
enregistrer_temps 104 2 102.3
enregistrer_temps 101 2 102.5
enregistrer_temps 103 2 102.8
enregistrer_temps 102 2 103.1
exit
```

outSp4

```
inscription dossard 101
inscription dossard 102
inscription dossard 103
inscription dossard 104
inscription dossard 105
inscription dossard 106
detection_fin_poursuite
Japon 102.3
Canada 103.1
```


Développement par sprints – Sprint #4 (1/3)

Créez un projet de nom `Sprint4` au sein de la même Solution que les autres Sprints: `Competition`. Au niveau des répertoires des projets, recopiez le source du `Sprint#3` dans le répertoire des sources de `Sprint#4`. Adaptez le source au `Sprint#4`.

Quatrième Sprint

- **Analyse fonctionnelle :** Limitation à des données de test nominales. Implémentez les commandes " `definir_parcours` " et " `detection_fin_poursuite` ".
- **Spécification :** Développez la fonctionnalité associée aux commandes : - définir le nombre de tours d'un parcours, - détecter la fin d'une poursuite et afficher le classement des équipes.
- **Codage :** (a) Prototypez et codez les fonctions `definir_parcours` et `detection_fin_poursuite` (b) mettre à jour le `main()` avec cette nouvelle commande. Les champs d'information d'une commande sont lus dans la fonction de traitement de la commande.
- **Test :** Testez votre application par redirection des entrées à partir du fichier `inSp4.txt` (JDT `du Sprint#4`) et sa sortie vers le fichier `run.txt`. Comparez votre fichier `run.txt` au fichier des résultats de référence `outSp4.txt` : si les deux fichiers coïncident, votre `Sprint#4` est validé, vous pouvez passer au `Sprint#5` sinon corrigez les erreurs.

Développement par sprints – Sprint #4 (2/3)

C2. Commande de définition du parcours de l'épreuve de poursuite
Une ligne composée de la chaîne de caractères "`definir_parcours`" suivie du nombre de tours `t` (entier supérieur ou égal à 2 et inférieure ou égal à 10).

détection de la fin de la poursuite
La détection de la fin de la poursuite est faite à chaque enregistrement. Elle intervient si tous les patineurs ont effectué `t` tours de piste. En cas de détection de fin de poursuite, "`detection_fin_poursuite`" est affiché, puis la commande d'affichage du temps final des deux équipes est appelée. L'ordre d'affichage des équipes est fait dans l'ordre croissant des temps finaux (équipe gagnante de la poursuite en premier).

Sprint#4 – Prototypage des fonctions (3/3)

Sprint#4

```
void definir_parcours(Course* crs);
void detection_fin_poursuite(Course* crs);
```

Spécification : les champs d'information d'une commande sont lus dans la fonction de traitement de la commande

Sprint #5 – inSp5.txt

inSp5

```
definir_parcours 2
definir_nombre_epreuves 2
inscrire_equipe Canada Blondin Weidemann Morrison
inscrire_equipe Japon Takagi Sato Takagu
inscrire_equipe France Pierron Huot Monvoisin
inscrire_equipe Italie Lollobrigida Mascitto Valcepina
enregistrer_temps 101 1 53.1
enregistrer_temps 102 1 53.2
enregistrer_temps 104 1 53.3
enregistrer_temps 105 1 53.7
enregistrer_temps 106 1 53.9
enregistrer_temps 103 1 54.1
enregistrer_temps 105 2 100.6
enregistrer_temps 106 2 101.7
enregistrer_temps 104 2 102.3
enregistrer_temps 101 2 102.5
enregistrer_temps 103 2 102.8
enregistrer_temps 103 2 103.1
enregistrer_temps 111 1 50.9
enregistrer_temps 108 1 52.1
enregistrer_temps 112 1 53.2
enregistrer_temps 107 1 53.5
enregistrer_temps 109 1 53.8
enregistrer_temps 110 1 54.1
enregistrer_temps 110 2 99.1
enregistrer_temps 109 2 100.3
enregistrer_temps 107 2 101.5
enregistrer_temps 112 2 101.8
enregistrer_temps 108 2 102.1
enregistrer_temps 111 2 102.6
exit
```


Sprint #5 – outSp5.txt

outSp5

```
inscription dossard 101
inscription dossard 102
inscription dossard 103
inscription dossard 104
inscription dossard 105
inscription dossard 106
inscription dossard 107
inscription dossard 108
inscription dossard 109
inscription dossard 110
inscription dossard 111
inscription dossard 112
detection_fin_poursuite
Japon 102.3
Canada 103.1
detection_fin_poursuite
France 102.1
Italie 102.6
Italie 102.6
detection_fin_competition
France 102.1
Japon 102.3
Italie 102.6
Canada 103.1
```

Développement par sprints – Sprint #5 (1/4)

Créez un projet de nom `Sprint5` au sein de la même Solution que les `Sprints 1 à 4 : Competition`. Au niveau des répertoires des projets, recopiez le source du `Sprint4` dans le répertoire des sources de `Sprint5`. Adaptez le source au `Sprint#5`.

Cinquième Sprint

- **Analyse fonctionnelle** : Définir le nombre d'épreuves et gérer la fin de la compétition
- **Spécification :** La détection de fin de compétition est testée à chaque enregistrement d'un point de mesure. Le test consiste à vérifier que tous les épreuves de poursuite sont terminées. Si la fin de compétition est détectée, le classement final sera affiché (cf. algorithme de tri).
- **Codage :** Codez (a) les fonctions *definir_nombre_epreuves et detection_fin_competition,* (b) mettre à jour le `main()` avec cette nouvelle commande. Les champs d'information d'une commande sont lus dans la fonction de traitement de la commande.
- **Test :** Testez votre application par redirection des entrées à partir du fichier `inSp5.txt (JDT du Sprint#5)` et sa sortie vers le fichier `run.txt`. Comparez votre fichier `run.txt` au fichier des résultats de référence `outSp5.txt` : si les deux fichiers coïncident, votre `Sprint#5` est validé. **Bravo !**

Développement par sprints – Sprint #5 (2/4)

C3. Commande de définition du nombre d'épreuves de poursuite
Une ligne composée de la chaîne de caractères "`definir_nombre_epreuves`" suivie du nombre d'épreuves de poursuite `n` (entier supérieur ou égal à 1 et inférieure ou égal à 16 et donc de 2 à 32 équipes)

Détection de la fin de la compétition
La détection de la fin de la compétition de poursuite est faite également à chaque enregistrement. Elle intervient si toutes les épreuves de poursuite ont eu lieu.
En cas de détection de la fin de la compétition, "`detection_fin_competition`" est affiché, puis la commande d'affichage du classement de l'ensemble des équipes en fonction du temps final réalisé par chaque équipe, enfin la commande "`exit`" est appelée.

Sprint#5 – Prototypage des fonctions (3/4)

Sprint#5

```
void definir_nombre_epreuves(Course* crs);
void detection_fin_competition(Course* crs);
```

Spécification : les champs d'information d'une commande sont lus dans la fonction de traitement de la commande

Sprint#5 – Algorithme de tri (4/4)

Soit un tableau de n éléments (indexés de 1 à n), le principe du tri par insertion est le suivant : construire une liste triée constitué **au début du 1er élément du tableau** et que l'on agrandira en y **insérant un à un** les autres éléments (de gauche à droite). Il s'agit d'un **tri stable** conservant l'ordre d'apparition des éléments égaux dans le tableau.

Algorithme de tri par sélection

```
 // tri d'un tableau t de taille n (indexé de 1 à n)
procedure tri_insertion(tableau t, entier n)
debut_procedure
    pour i allant de 2 à n
       v ← t[i]
       j ← i
       tant que j > 1 et t[j - 1] > v
          t[j] ← t[j - 1]
          j ← j - 1
       fin tant que
       T[j] ← v
    fin pour
fin_procedure
```

Attention : vous aurez à adaptez l'algorithme à la déclaration en langage C d'un tableau de taille `n` (indexé de `0` à `n-1`)