PARIS DESCARTES IUT

IAP - Introduction à l'Algorithmique et à la Programmation

Equipe pédagogique
Marie-José Caraty, Julien Rossit, Camille Kurtz,
Jacques Alès-Bianchetti, Eloi Keita

Cours Projet

Les fondamentaux de la programmation impérative

Analyse du projet

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

Le problème posé

Programmer un interpréteur de commandes pour la gestion d'une table de données générique
- 8 commandes de la base
- 1 commande de sortie de l'interpréteur

Les Commandes
1. Création
2. Destruction
3. Affichage du schéma de la base
4. Insertion d'une ligne de données
5. Affichage d'une ligne de données
6. Affichage de toutes les lignes de données
7. Destruction d'une ligne de données
8. Sélection monocritère
9. Commande de sortie de l'interpréteur

L'interpréteur des commandes

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

Neuf commandes devront être reconnues par l'interpréteur :

1. **Create_table**
Commande la création d'une table de données et la définition de son schéma de données :
une ligne composée de la chaîne de caractères "Create_table" suivie du nom de la table, du nombre de champs,
et pour chaque champ : du nom du champ et de son type.

Ex : *Create_table Pays 5 nom TEXT pib INT capitale TEXT latitude FLOAT longitude FLOAT*
Spécification : Si la table est déjà créée, la commande affichera "table existante" sinon "table inconnue".

L'interpréteur des commandes

2. **Delete_table**
Commande la destruction d'une table de données.
Une ligne composée de la chaîne de caractères "Delete_table" suivie du nom de la table de données.

Ex : *Delete_table Pays*
Spécification : Affiche "table inconnue" si le nom de table est inconnu sinon rien.

3. **Afficher_schema**
Commande l'affichage du schéma d'une table de données.
Une ligne composée de la chaîne de caractères "Afficher_schema" suivie du nom de la table de données.

Ex : *Afficher_schema Pays*
Spécification : Affiche "table inconnue" si le nom de la table est inconnu sinon le schéma de la table (nombre de paramètres suivi du nom et du type de chacun des paramètres).

L'interpréteur des commandes

4. **Insert_enregistrement**
   Commande l'insertion d'une ligne de données en fin de table.
   Une ligne composée de la chaîne de caractères "Insert_enregistrement" suivie du nom de la table et des données du schéma de la base.

   Ex : *Insert_enregistrement Pays Japon 38492 Tokyo 35.70 139.73*

   Spécification : Affiche "table inconnue" si le nom de la table est inconnu sinon rien.

5. **Afficher_enregistrement**
   Commande l'affichage d'un enregistrement d'un numéro donné.
   Ligne composée de la chaîne de caractères "Afficher_enregistrement" suivie du nom de la table et du numéro de l'enregistrement.

   Ex : *Afficher_enregistrement Pays 5*

   Spécification : Les numéros d'enregistrement d'une table commencent à 1. Affiche "table inconnue" si le nom de la table est inconnu et "enregistrement inconnu" s'il n'y a pas d'enregistrement de ce numéro.

L'interpréteur des commandes

6. **Afficher_enregistrements**
   Commande l'affichage de l'ensemble des lignes de données.
   Ligne composée de la chaîne de caractères "Afficher_enregistrements" suivie du nom de la table.

   Ex : Afficher_enregistrements Pays

   Spécification : Affiche "table inconnue" si le nom de la table est inconnu, sinon l'ensemble des lignes de données.

7. **Delete_enregistrement**
   Commande la destruction d'un enregistrement sélectionné par son numéro.

   Ex : Delete_enregistrement Pays 10

   Spécification : Provoque le décalage des numéros d'enregistrement suivants. Les numéros commencent à 1. Affiche "table inconnue" si le nom de la table est inconnu et "enregistrement inconnu" s'il n'y a pas d'enregistrement de ce numéro, sinon rien.

L'interpréteur des commandes

8. **Select_enregistrements**
   Commande la sélection et de l'affichage des lignes de données suivant un critère donné et un intervalle de sélection choisi
   Ligne composée de la chaîne de caractères "Select_enregistrements" suivie du nom de la table, du critère de sélection et des bornes inférieure et supérieure de sélection

   Ex : Select_enregistrements Pays pib 5000 10000

   Spécification : Affiche "table inconnue" si le nom de la table est inconnu, sinon l'ensemble des lignes de données.

9. **Exit**
   Commande de sortie de programme

   Ex : Exit

   Spécification : Sortie de la boucle de traitement de commandes de l'interpréteur.

Les constantes du problème – Leur déclaration (1/2)

La déclaration des tableaux statiques
nécessite la connaissance de leur taille

On est amené à dimensionner le problème
et souvent à sur-dimensionner la taille des tableaux

Quelques constantes du problème
- le nombre maximum de champs d'une table (**maxChamps**) peut varier, il est borné à **25**
- Le nombre maximum d'enregistrement d'une table est borné à **100**
- la taille maximale d'une chaîne de caractères considérée est **30**
- la taille maximale d'une ligne de commande (comprenant la commande et ses paramètres) considérée est **80**

Les constantes du problème – Leur déclaration (2/2)

Pour leur déclaration, on utilisera les directives du préprocesseur

Directives (documentées) à insérer après les inclusions et avant le main()

```
#define max_champs 25 // Nombre maximum de champs d'une table
#define lgMot 30 // Longueur max d'une chaine de caractères
#define lgMax 80 // Longueur maximale d'une ligne de commande
…
```

Définir toutes les constantes
Ces directives sont appelées macros (elles peuvent exprimer un calcul)

Le traitement du préprocesseur est alors de remplacer
dans le source toute occurrence

- du mot **`max_champs`** par **`25`**
- du mot **`lgMot`** par **`30`** (littéral, constante entière)
- etc.

Rem : Le tableau de caractères (`mot`) permettant de stocker la chaîne de caractères d'une commande est déclaré :

```
char mot[lgMot+1] // pour stocker le caractère '\0'
                  // l'indicateur de fin de chaîne
```


Structuration des données – Les données en mémoire (1/3)

(1) Définir les types `Champ` et `Table`

Le champ d'une table de données est défini par son nom et son type
Les champs définis pour un pays

```
nom TEXT
pib INT
capitale TEXT
latitude FLOAT
longitude FLOAT
```

Les types de champ considérés

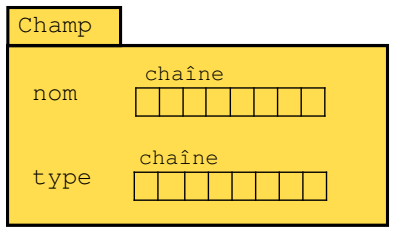
- `TEXT` (chaîne de caractères)
- `INT` (entier)
- `FLOAT` (réel)

Une table est définie par ses caractéristiques

- son nom,
- son schéma de données

Type introduit au Sprint#1 (en noir))

Structuration des données – Les données en mémoire (2/3)
Le type `Champ`



Documenter toute structure de données
Complétez les lignes de documentation du type et des champs repérées par //…

Codage en langage C

```
//…
typedef struct {
   char nom[lgMot+1]; // …
   char type[lgMot+1]; // …
} Champ;
```

Accès aux attributs d'un champ
Soit la déclaration suivante :

```
Champ c;
```

on utilisera la notation pointée ". "

```
c.nom  // pour accéder au nom de c
c.type // pour accéder au type de c
```

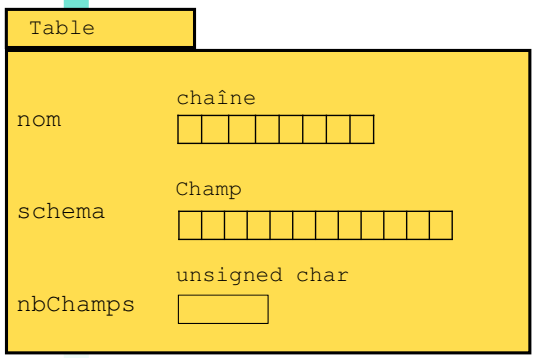
Si c est un pointeur sur une variable de type Champ

```
Champ* c; // ou Champ *c;
```

on utilisera la notation flèche "-> "

```
c->nom  // pour accéder au nom de c
c->type // pour accéder au type de c
```


Structuration des données – Les données en mémoire (3/3)
Le type `Table`



Type introduit au Sprint#1
À compléter au fil des sprints

```
typedef struct {
   char nom[lgMot+1];
   Champ schema[max_champs];
   unsigned char nbChamps;
} Table;
```

`schema` tableau statique sur-dimensionné à `max_champs`
`nbChamps` : nombre de champs de la table

Soit la déclaration :

```
Table t;
```

Accès aux type du 1er champ de la `Table t`

```
t.schema[0].type
```

Accès au nom du dernier champ de la table `t`

```
t.schema[nbChamps-1].nom
```

Lecture des données (cf. sscanf)

Toutes les données (chaînes, entiers et réels) seront lues à partir d'une chaîne de caractères
Vous utiliserez les fonctions `atoi()` et `atof()` pour convertir la chaîne de caractères respectivement en entier et en réel

`int atoi(const char *str)`
Retourne l'entier correspondant à la conversion en entier de la chaîne pointée par `str`

`double atof(const char *str)`
Retourne l'entier correspondant à la conversion en flottant de la chaîne pointée par `str`

Exemples :

```
unsigned int n;
scanf("%s" , mot);   // mot entré "123"
n=atoi(mot)          // n vaut 123

double r;
scanf("%s" , mot);   // mot entré "1.34"
r=atof(mot)          // r vaut 1.34
```


Manipulation des chaînes de caractères – `strcmp, strcpy`

http://www.tutorialspoint.com/c_standard_library/
Fonctions de String.h, ajoutez l'inclusion :
`#include <string.h>`

Copie de chaînes de caractères `strcpy` (string copy)
`char* strcpy(char* dest, const char* src)`
Copie la chaîne pointée par `src` dans la chaîne pointée par `dest`

Comparaison de chaînes de caractères `strcmp` (string compare)
`int strcmp(const char* str1, const char* str2)`
Compare la chaîne pointée par `str1` avec la chaîne pointée par `str2`
- si la valeur retournée est < 0 alors str1 est inférieure à str2
- si la valeur retournée est > 0 alors str1 est supérieure à str2
- si la valeur retournée est égale à 0 alors str1 est égale à str2
- "inférieure, supérieure ou égale" : au sens de l'ordre alphabétique

Le programme principal

Boucle infinie

```
int main() {
   …
   while (1) { // ou do {…} while(1);
      // boucle infinie sur les 7 commandes
      scanf("%s" , mot); // le mot lu est une chaîne de 30 caract.

      …
      …
      if (strcmp(mot, "Exit")==0) {
          exit(0); // sortie du programme principal
      }
   }
   system("pause"); return 0;
}
```

Sprint #1 `main` à coder.
Il sera complété de sprint en sprint jusqu'au Sprint #5

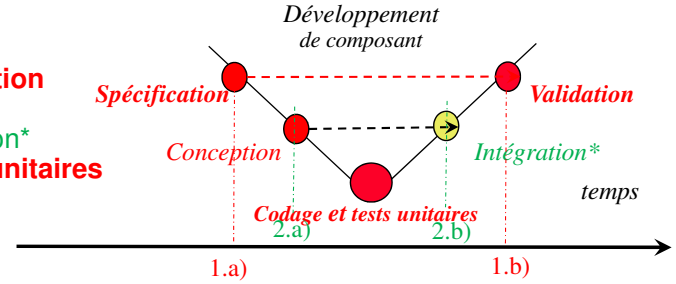
Développement de l'application à base de Sprints

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



* Phase « des gros projets » assimilable à une décomposition en sous-problèmes

Développement par Sprints (1/3)

Phase de développement

Deux fichiers `inSpn.txt` et `outSpn.txt` (pour 1 ≤ n ≤ 5) sont donnés pour un `Sprint#n`.
Le premier fichier correspond aux entrées de l'application testées appelées le Jeu de Données de Test (JDT), le second correspond aux sorties attendues de l'application appelées le résultat de référence

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 `inSpn.txt` avec le résultat de référence donné (`outSpn.txt`)

Pour comparer les deux solutions, vous utiliserez la comparaison de fichiers cf. **Diff.jar** (archive de l'exécutable d'un programme Java de test d'identité de fichiers)
si les 2 fichiers (`run.txt` et `outSpn.txt`) sont identiques (pas de différence détectée),
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 (2/3)

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

Attention (1) Chaque Sprint validé donne lieu à un développement matérialisé par un projet et représente une évolution dont vous devez garder la trace.
Lorsque vous passez à un autre Sprint, créez un nouveau projet où **vous recopierez au niveau des répertoires** (cf. toutes les entités logicielles (validées) du précédent projet qui sont nécessaires au nouveau sprint ou à y adapter). Pour aller au niveau des répertoire <clic droit> sur la solution, sélectionnez le/les fichier(s) à copier <CTRL>C puis <CTRL>V .

Attention (2) Dans le cas où votre programme ne fonctionne pas sur l'ensemble des fonctionnalités demandées
Les projets liés aux Sprints sont indispensables pour montrer l'état de développement de votre logiciel.
Chaque sprint validé correspond à un très bon travail que vous pouvez démontrer (tests réussis) et qui vous rapportera des points.

Développement par Sprints (3/3)

1. Dans le cas où votre programme ne fonctionne pas sur l'ensemble des fonctionnalités demandées

   Dans le dossier de développement logiciel demandé,
   vous donnerez les listings et traces d'exécution correspondant au sprint de plus haut niveau que vous avez passé avec succès en recette et les listings du sprint suivant (refusé en recette)

2. Dans le cas où votre programme fonctionne sur l'ensemble des fonctionnalités demandées

   Dans le dossier de développement logiciel demandé,
   seuls les listings et les traces d'exécution du dernier sprint sont à donner.

Les sprints de l'application

Cinq sprints
représentant cinq incréments de fonctionnalité
de l'application

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

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

Premier Sprint

- **Analyse fonctionnelle :** Implémentez les commandes "`Exit`", "`Create_table`", et "`Afficher_schema`".
- **Spécification :** Définissez les structures de données pour l'application : `Champ` et `Table`. Développez la fonctionnalité associée aux commandes :
  – définir la structure de la table de données: nom, nombre de champs, pour chaque champ : son nom et son type.
  – afficher le structure de la table de données (schéma).
- **Codage :** Prototypez et codez (**a**) la fonction `Create_table` à partir des champs d'information de la commande, (**b**) la fonction `Afficher_schema` qui affiche le schéma de la table de données et (**c**) le `main()` : boucle sur les trois commandes : "`Exit`", "`Create_table`", "`Afficher_schema`". 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 `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.

Sprint#1 – Prototypage des fonctions (2/2)

Les champs d'information d'une commande seront lus dans la fonction correspondante

Sprint#1

Prototype des fonctions

```
void create_table(Table* t);

void afficher_schema(Table* t);


// void create_table(Table t);
// Prototype non optimisé en raison de la copie
// du paramètre effectif dans le paramètre formel
```


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

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

Deuxième Sprint

- **Analyse fonctionnelle** "`Inserer_enregistrement`" et "`Afficher_enregistrements`" pour la table de données d'un nom donné.
- **Spécification :** Définissez les structures de données pour l'application : `Data`, `Enregistrement`. Modifiez la structure de données `Table`. Développez les fonctionnalités associées aux nouvelles commandes : insérer une ligne de données avec (enregistrement) en fin de table, afficher l'ensemble des lignes de données dans l'ordre d'insertion.
- **Codage :** Prototypez et codez (**a**) la fonction `inserer_enregistrement`, (**b**) la fonction `afficher_enregistrements` et (**c**) mettre à jour le `main()` avec ces deux nouvelles commandes.
- **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.

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

Les champs d'information d'une commande seront lus dans la fonction correspondante

Sprint#2

```
void inserer_enregistrement(Table* t);

void afficher_enregistrements(Table* t);
```


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

Créez un projet de nom `Sprint3` au sein de la même Solution que les `Sprints 1` et `2` : `GestionTableDonnee`. Au niveau des répertoires des projets, recopiez le source du `Sprint2` dans le répertoire des sources de `Sprint3`. Adaptez le source au `Sprint3`.

Troisième Sprint

- **Analyse fonctionnelle :** Implémentez les commandes "`Delete_enregistrement`" et "`Afficher_enregistrement`".
- **Spécification :** Développez les fonctionnalités associées aux nouvelles commandes : supprimer une ligne de données d'un numéro donné, afficher une ligne de données d'un numéro donné.
- **Codage :** Prototypez et codez (**a**) la fonction `delete_enregistrement` qui détruit un enregistrement sélectionné par son numéro et décale les numéros des enregistrements suivants, (b) la fonction `afficher_enregistrement` qui affiche un enregistrement d'un numéro donné, (c) mettre à jour le `main()` avec ces deux nouvelles commandes.
- **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.

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

Les champs d'information d'une commande seront lus dans la fonction correspondante

Sprint#3

```
void delete_enregistrement(Table* t);

void afficher_enregistrement(Table* t);
```

.

Développement par sprints – Sprint #4 (1/2)

Créez un projet de nom `Sprint4` au sein de la même Solution que les `Sprints 1 à 3` : `GestionTableDonnee`. Au niveau des répertoires des projets, recopiez le source du `Sprint3` dans le répertoire des sources de `Sprint4`. Adaptez le source au `Sprint4`.

Quatrième Sprint

- **Analyse fonctionnelle :** Implémentez la commande "`Delete_table`".
- **Spécification :**

Développez la fonctionnalité associée à la nouvelle commande : supprimer la table de donnée d'un nom donné.

- **Codage :** (a) Prototypez et codez la fonction `delete_table` qui supprime la table de données courante d'un nom donné et rend possible la création d'une nouvelle table de données, (b) mettre à jour le `main()` avec cette nouvelle 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.

Sprint#4 – Prototypage des fonctions (2/2)

Les champs d'information d'une commande seront lus dans la fonction correspondante

Sprint#4

```
void delete_table(Table* t);
```

.

Développement par sprints – Sprint #5

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

Cinquième Sprint

- **Analyse fonctionnelle : :** Implémentez la commande "`Select_enregistrement`"
- **Spécification :** Développez la fonctionnalité associée à la commande : sélection et affichage des enregistrements suivant un champ donné et un intervalle de sélection donné
- **Codage :** Prototypez et codez (a) la fonction `select_enregistrement` qui affiche les enregistrements sélectionnés, (b) la fonction `compare_enregistrement` qui vérifie si un champ d'un enregistrement est dans un intervalle donné, (c) mettre à jour le `main()` avec cette nouvelle 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, sinon corrigez les erreurs.

Sprint#5 – Prototypage des fonctions (2/2)

Les champs d'information d'une commande seront lus dans la fonction correspondante

Sprint#5

```
void select_enregistrement(Table* t);
```

.