Exemple de modélisation

CCI

Principes des Langages de Programmation

Introduction

Objectifs, problèmes

Objectifs

Développer un exemple complet, mettre en évidence les étapes du développement

- Cahier des charges
- Spécification détaillée
- Analyse
- Conception
- Production de code
- Validation

Objectifs-suite

Illustrer des bons principes de programmation

- Structuration
- Evolution
- Commentaires
- Documentation
- Tests

Gestion de salles et d'emplois du temps

Cahier des charges informel, exprimé par le commanditaire.

- Réserver des salles pour des cours
- Consulter des emplois du temps

Cahier des charges détaillé

- Étude des conditions d'utilisation du système
- Établi en interaction avec les usagers

Dans l'exemple :
- Recherche de salles en fonction de critères de capacité
- Réservations simples ou multiples (date initiale, date finale, périodicité)
- Nature des emplois du temps : par salle, par formation, sur un jour, une semaine, un mois
- Informations à faire paraître sur les emplois du temps : intitulé du cours, groupes de TD/TP.

Des questions plus techniques

- Modification de la base : capacités, ajout ou suppression de salles, suppression ou archivage des vieilles réservations
- Sauvegardes entre différentes sessions du programme, interface avec d'autres systèmes ...
- Critères d'accès à l'application : différentes personnes pour administrer le système, effectuer les réservations, consulter les emplois du temps ...
- Dimensionnement prévu (nombre de salles, formations ...)
- Mono ou multi-utilisateurs (exclusion mutuelle lors des modifications)

Analyse

Données, opérations

Analyse détaillée

Identifier le modèle des données.

On peut s'aider de notations.
Par exemple UML (Unified Modeling Language).

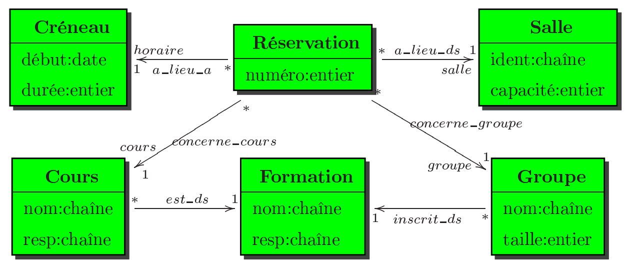
- Diagrammes de classe
  présentent les entités concrètes et abstraites manipulées par le programme.
- Diagrammes de cas d'utilisation
  précisent les opérations que peuvent réaliser les utilisateurs
- Diagrammes de séquence
  précisent le déroulement temporel des opérations

UML-Diagramme de classes

Principe de base

- Des classes identifiées par
  - Un nom
  - Des attributs élémentaires (types de base)
  - Des opérations
- Des relations entre des classes
  - Noms
  - Cardinalité de la relation
- Propriétés des classes et relations

Diagramme de classes-exemple



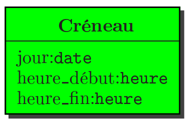
Validation

Confrontation au cahier des charges initial/à des exemples.
Affiner le niveau de détail de la modélisation.

- Groupes :
  - taille ? apparaît dans la demande, utile pour modification future
  - cours/TD/TP : modéliser les relations d'inclusion,
    utile pour gérer les contraintes, ou pour établir un edt au niveau d'un groupe de cours ou de TD
  - rattacher un groupe à plusieurs formations
- Cours :
  - Utilité du rattachement à la formation ?

Validation-suite

- Réservation :
  - ajouter la date de réservation,
    l'identifiant de la personne qui a réservé
- Créneau horaire :
  - Distinguer jour et heure ?
    (édition d'emploi du temps par jour)
    Le choix de représentation sera fait à la conception
  - Heure de fin plutôt que durée



Contraintes

On peut exprimer des contraintes sur le modèle.
Utiliser les noms introduits, un langage formel (OCL)

- Contraintes locales :
  pour chaque réservation, la taille du groupe associé est inférieure ou égale à la capacité de la salle
- La formation associée au groupe et au cours d'une réservation coincident.

Contraintes-suite

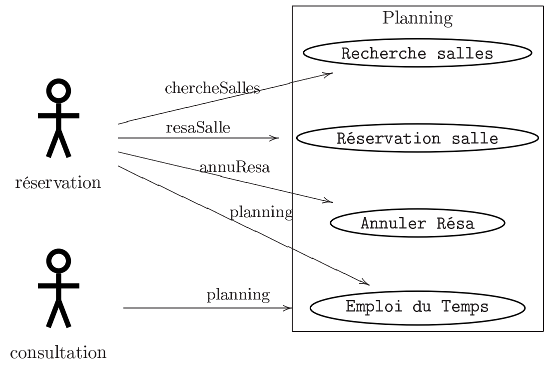
- Contraintes globales :
  - deux réservations différentes ont des numéros différents
  - deux salles différentes ont des numéros différents
  - deux réservations différentes de la même salle ne peuvent pas avoir des horaires qui se chevauchent
  - deux réservations différentes qui concernent le même groupe ne peuvent pas avoir des horaires qui se chevauchent (condition insuffisante si les mêmes personnes sont dans plusieurs groupes . . .)

UML-Cas d'utilisation

Les cas d'utilisation font apparaître :

- Les acteurs extérieurs interagissant avec le système
  - concrets
  - abstraits (horloge)
- Les services offerts par le système
- Les noms des fonctions d'interaction

Cas d'utilisation-Exemple



Autres cas d'utilisation

- Mise à jour système
- Authentification
- Sauvegarde
- Archivage

Modèles d'opération

Spécification des fonctions d'interaction apparaissant dans le cas d'utilisation.

Pour chaque fonction :
- Module
- Arguments d'entrée, valeurs de retour, effets (erreurs)
- Descriptif en langue naturelle
- Conditions d'utilisation
- Propriétés établies

Opérations : recherche de salles

chercheSalles(taille:`entier`,debut:`date`,duree:`entier`)
: ensemble de `chaine`

Descriptif
Fournit un ensemble de salles disponibles au moment souhaité ayant au moins une capacité égale à la taille. On peut borner le nombre de salles proposées par un nombre `MAX` de propositions.
Conditions La taille est positive (non nulle).
Propriétés Les salles proposées sont disponibles à la date voulue et de capacité adaptée.

Opérations : réservation

resaSalle(ident:`chaine`,grp:`chaine`,debut:`date`,duree:`entier`)
: `entier`
: signals nonDispo, nonAdapt, nonAut

Descriptif Enregistre la réservation de la salle `ident` pour le groupe `grp` à l'horaire donné.
Suppose que le groupe, la salle sont valides.
Échoue avec l'erreur `nonDispo`-salle non disponible; `nonAdapt`-capacité insuffisante; `nonAutor`-utilisateur non autorisé à réserver.
Renvoie un numéro de réservation unique.

Opérations : annulation

annuSalle(no:`entier`)
: `entier`
: signals nonDispo, nonAut

Descriptif Supprime la réservation de numéro `no`. Échoue avec l'erreur `nonDispo`-la réservation n'existe pas; `nonAutor`-utilisateur non autorisé à réserver.
Renvoie un numéro d'annulation unique.

Opérations : planning

planning_formation(form:`chaîne`,début:`date`,durée:`entier`)
: ensemble de `entier`
planning_salle(salle:`chaîne`,début:`date`,durée:`entier`)
: ensemble de `entier`

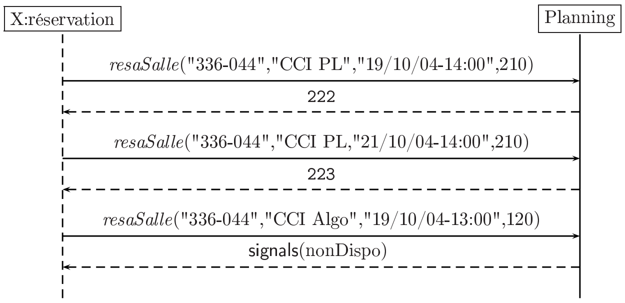
Descriptif Renvoie l'ensemble des réservations concernant une formation ou une salle dans une période donnée exprimée en nombre de jours.

Opérations : indications

Les opérations décrivent l'interface du système :
- Peu de fonctions
- Fonctionalités orthogonales
- Programmation robuste
  (vérifications, maintien de la cohérence)
- Distinguer l'interface utilisateur de l'interface système :
  - système : une fonction de réservation simple, robuste
  - utilisateur : des fonctions de réservation multiple, interface graphique (choix des salles, groupes, dates prédéfinis)...

UML-Diagrammes de séquence

Visualisation du déroulement des opérations sur des données réelles
Préparent les jeux de tests.



Analyse-conclusion

Phase essentielle pour préciser les contours du système
– Affiner, mettre en évidence les choix
– Schémas pour la documentation de la modélisation
– Opérations à l'interface du système et de l'utilisateur
– Hypothèses sur les opérandes des opérations
Les notations à la UML peuvent aider à visualiser le système.
– restent semi-formelles (donc sujettes à interprétation variées)
– contraintes pas toujours adaptées
(types de base dans les attributs/opérations)

Conception

Structures de données, algorithmes

Conception : principes

La conception permet de préciser comment le système va être construit.
Prépare le codage.

– Affiner les structures de données (relations,attributs)
– Préciser les variables d'états du système
– Identifier les principaux algorithmes
– Préparer les tests

Relations et Attributs

Les relations du diagramme de classes deviennent des attributs sur des valeurs structurées ou bien des valeurs calculées.

Reservation Une réservation comportera comme attributs :
– Un numéro
– La salle
– Le groupe
– Le créneau horaire
– Le cours
Le système garde en mémoire un ensemble de réservations.

Autres données

On stocke également :
– Ensemble de salles
– Ensemble de groupes
– Ensemble de cours
– Ensemble de formations
– Table d'autorisations
(login, mot de passe, droit)
– Droit d'utilisation courant
Hypothèse simplificatrice : Un groupe et un cours sont associés à une seule formation.

Dimensionnement

Une formation correspond à une dizaine de cours pour une dizaine de groupes différents (cours/TD/TP/Options).
Chaque cours a lieu environ sur 12 séances.

Salles	100
Groupes	100
Cours	100
Formations	10
Réservations	1200

Organisation en fichiers

Pour chaque table (salles, groupes, cours, formations, autorisations), on peut créer un fichier qui abstrait la représentation.
Type de données de base : `Salle`, `Groupe`, `Cours`, `Formation`, `Autorisation`

Fonctions de base :
– Ajout/Retrait/Modification d'une entrée
– Recherche à partir d'une donnée clé

Contraintes

Vérifier le respect des contraintes :
- Le retrait de formations peut rendre invalides des cours ou des groupes
- Le retrait de cours, salles, groupes peuvent rendre invalides des réservations.

Solutions possibles
- Interdire les retraits
- Interdire les retraits dans le cas où on invalide les contraintes
- Mettre à jour le système de manière cohérente

Algorithmes

Repérer les opérations difficiles
- Recherche de salles suivant un critère (capacité, disponibilité).
- Fabrication d'emplois du temps

Trouver de bonnes solutions algorithmiques

Algorithmes : recherche de salles/emploi du temps

Salles :

1. Parcourir toutes les salles pour trouver celles de bonne capacité
2. Parcourir toutes les réservations pour ne retenir que les salles disponibles

Emploi du temps :
- Parcourir toutes les réservations pour ne retenir que celles concernant la salle ou la formation souhaitée sur les jours concernés par l'emploi du temps.

Solutions possibles

Suggère une organisation des réservations par salle/date/formation

Structures de données :
- Des données triées pour améliorer l'efficacité de la recherche
- Structure de données supplémentaire
  - Un tableau par salle, donnant une liste créneaux de réservations triés par date
  - Un tableau par formation, donnant une liste créneaux de réservations triés par date

Maintien de la cohérence assuré par la fonction d'ajout de réservation.

Créneaux horaires

Introduire un type de données de base `créneau` pour les créneaux horaires.

Fonctions utiles :
- Compatibilité entre deux créneaux c_1 et c_2.
  non-chevauchement :
  $\texttt{fin}(c_1) \leq \texttt{debut}(c_2)$ ou $\texttt{fin}(c_2) \leq \texttt{debut}(c_1)$
- Ordre entre de deux créneaux c_1 et c_2.
  $\texttt{debut}(c_1) \leq \texttt{debut}(c_2)$
  Ordre total pour les emplois du temps par formation

Choix de conception

- Réutiliser des bibliothèques existantes :
  (en particulier pour les dates)
- Utiliser des algorithmes génériques :
  - Tables d'associations
  - Recherche salles/réservations suivant critères

Interfaces

Interfaces : `formation.h`

```c
#define MAXFORM 10
struct formation {char * nom;char * resp;};
typedef struct formation Formation;

/* renvoie NULL si la formation n'existe pas */
Formation *cherche_formation(char nom[]);
```

Interfaces : formation.h (suite)

```
/* Ajout/Modification/Retrait */
void ajout_formation(char nom[],char resp[]);
void modif_formation(char nom[],char resp[]);
void retire_formation(char nom[]);

/* Acces */
char * resp_formation (char nom[]);
```


Interfaces : formation.h (suite)

```
/* Ajout/Modification/Retrait */
void ajout_formation(char nom[],char resp[]);
void modif_formation(char nom[],char resp[]);
void retire_formation(char nom[]);

/* Acces */
char * resp_formation (char nom[]);
```


Interfaces : salle.h

```
#define MAXFORM 100
struct salle {char *nom;int capacite;};
typedef struct salle Salle;

Salle *cherche_salle(char nom[]);

void ajout_salle(char nom[],int capa);
void modif_salle(char nom[],int capa);
void retire_salle(char nom[]);

int capacite_salle (char nom[]);
```


Interfaces: groupe.h

```
#define MAXGRP 100

struct groupe {char * nom; Formation * form;int taille;};
typedef struct groupe Groupe;

/* renvoie NULL si la groupe n'existe pas */
Groupe * cherche_groupe(char nom[]);
```


Interfaces: groupe.h (suite)

```
/* Ajout/Modification/Retrait */
void ajout_groupe(char nom[],Formation *form, int taille);
void modif_taille_groupe(char nom[],int taille);
void modif_formation_groupe(char nom[],Formation *form);
void retire_groupe(char nom[]);
/* Acces */
Formation * formation_groupe (char nom[]);
int taille_groupe (char nom[]);
```


Interfaces: reservation.h

```
#define MAXRESA 1000
struct reservation {
  int no;
  Salle *salle;
  Creneau *creneau;
  Cours *cours;
  Groupe *groupe
  };
typedef struct reservation Reservation;

/* renvoie NULL si la reservation n'existe pas */
Reservation *cherche_reservation(int no);
```


Interfaces: reservation.h (suite)

```
void ajout_reservation
  (int no,Salle *salle,Creneau *creneau,Cours *cours,Groupe
void retire_reservation(int no);

Salle *salle_reservation (int no);
Creneau *creneau_reservation (int no);
Cours *cours_reservation (int no);
Groupe *groupe_reservation (int no);
```


Créneaux horaires : <time.h>

```
struct tm {
  int tm_sec;   /* Seconds.     [0-60] (1 leap second) */
  int tm_min;   /* Minutes.     [0-59] */
  int tm_hour;  /* Hours.       [0-23] */
  int tm_mday;  /* Day.         [1-31] */
  int tm_mon;   /* Month.       [0-11] */
  int tm_year;  /* Year - 1900.        */
  int tm_wday;  /* Day of week. [0-6]  */
  int tm_yday;  /* Days in year.[0-365] */
  int tm_isdst; /* DST.         [-1/0/1]*/
}
```

<time.h> : fonctions de base

```c
/* Retourne le temps courant */
time_t time(time_t *tp);

/* Retourne la forme locale du temps */
struct tm *localtime(const time_t *tp)

/* Affichage */
char *asctime (const struct tm *tp)
char *strftime
   (char *s, size_t smax, const char *fmt, const struct tm
```

Pas suffisant pour la gestion d'un calendrier
Construire ses propres fonctions

Créneaux horaires

Version simplifiée de `struct tm`

```c
struct date {
  int jour;     /*1-31*/
  int mois;     /*0-11*/
  int annee;    /* annee - 2000 */
  int heure;    /*0-23*/
  int minute;   /*0-59*/
  int jour_sem; /*0-6*/
};
typedef struct date Date;
```


Bibliothèque sur les dates

```c
/* Retourne la date du jour, utilise time */
Date aujourdhui();
/* Vrai si annee bissextile */
int bissextile(int annee);
/* Numero du jour dans l'annee */
int jour_annee(Date d);
/* Jour dans la semaine du premier de l'an */
int jour_premier_an(int annee);
/* Met à jour le jour dans la semaine d'une date */
int update_jour_semaine(Date d);
```


Documentation

Manuels de référence et d'utilisation

Documentation

Manuel d'utilisation À l'usage de l'utilisateur

Manuel de référence Maintenance du système

Manuel d'utilisation

- Identification : nom, version, auteurs, hotline . . .
- Description générale : possibilités et limites, terminologie, plan du manuel, références
- Environnement du logiciel : matériel minimal, logiciels, OS
- Installation/Démarrage du logiciel : connexion de périphériques, chargement de fichiers, paramétrage . . .
- Guide de l'utilisateur :
  Principales fonctions illustrées sur des exemples simples
- Liste des commandes : messages, cas d'erreurs.
- Format des fichiers.

Codage

Codage

- Choix d'un langage de programmation
- Traduction des choix de conceptions
  - Codage des structures de données
  - Codage des algorithmes

  Utiliser des bibliothèques

Interfaces : formation.h

```c
#define MAXFORM 10
struct formation {char * nom;char * resp;};
typedef struct formation Formation;

/* renvoie NULL si la formation n'existe pas */
Formation *cherche_formation(char nom[]);
```


Interfaces : formation.h (suite)

```c
/* Ajout/Modification/Retrait */
void ajout_formation(char nom[],char resp[]);
void modif_formation(char nom[],char resp[]);
void retire_formation(char nom[]);

/* Acces */
char * resp_formation (char nom[]);
```


Remarques sur les interfaces

- Choix de noms longs, uniformité
- Des constantes pour tous les nombres arbitraires
- Définition systématique de types
- Choix de pointeurs pour le partage des données
- Cohérence des prototypes

Exemple de code : formation.c

```c
#include "formation.h"
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

static Formation tab [MAXFORM];
static int tabsp = 0;

Formation * cherche_formation(char nom[]) {
  int i;
  for (i=0; i<tabsp;i++)
    {if (strcmp(tab[i].nom,nom)==0) return &(tab[i]);}
  return NULL;
}
```


formation.c (suite)

```c
void ajout_formation(char nom[],char resp[]) {
  Formation tmp = {nom,resp};
  if (tabsp==MAXFORM) fprintf(stderr,"Table des formations
  else tab[tabsp++]=tmp;
}

void modif_formation(char nom[],char resp[]) {
  Formation * tmp = cherche_formation(nom);
  if (tmp != NULL) tmp->resp = resp;
  else fprintf(stderr,"Formation %s non trouvee\n",nom);
}
```


formation.c (fin)

```c
void retire_formation(char nom[]) {
  Formation * tmp = cherche_formation(nom);
  if (tmp != NULL) {
      *tmp = tab[--tabsp];
      printf("Formation %s retiree\n",nom);
      return;
  }
  fprintf(stderr,"Formation %s inexistante\n",nom);
}
```


Erreurs

- L'impression de messages pour indiquer le succès ou l'échec n'est pas très aproprié :
  on ne peut pas tester le comportement normal/exceptionnel dans une autre fonction.
- Certains langages ont des mécanismes indépendants d'exception.
  En C les valeurs exceptionnelles sont traitées comme des valeurs de retour spéciales.

Gérer les cas d'erreur en C

- Introduire un type énuméré correspondant aux différents cas d'erreur possible
- Si le type de retour est void on peut lui substituer le type des erreurs
- Si le type de retour est incompatible avec le type des erreurs, on peut convenir de renvoyer le type de retour dans un argument (de type pointeur) passé en argument.

On peut ensuite encapsuler ces fonctions de base dans des fonctions qui affichent des messages d'erreur ou de succès.

Exemple des formations

Les erreurs qui peuvent se produire sont :
- table pleine
- entrée déjà présente
- droit d'accès incompatible avec l'opération à réaliser
- recherche/modif d'un objet inexistant

On peut introduire le type suivant qui pourra être utilisé par l'ensemble des modules du programme :

```
enum erreur {OK=0,PLEIN,PRESENT,AUTOR,INEXISTANT};
typedef enum erreur Erreur;
```


Nouvelle interface formation.h

```
Erreur ajout_formation(char nom[],char resp[]);
Erreur modif_formation(char nom[],char resp[]);
Erreur retire_formation(char nom[]);

/* renvoient NULL si la formation n'existe pas */
Formation * cherche_formation(char nom[]);
char * resp_formation (char nom[]);
```


Problème des retraits

Le choix de codage n'est pas correct dans le cas du retrait d'une formation.

La formation en dernière position est déplacée.

Un groupe qui aurait un pointeur sur cette formation devient incorrect.

Solution possible Utiliser plutôt un tableau de pointeurs :

Nouvelle implantation

```
static Formation *tab [MAXFORM];
static int tabsp = 0;

Erreur ajout_formation(char nom[],char resp[]) {
  Formation * tmp;
  if (tabsp < MAXFORM) {
    tmp = (Formation *) malloc(sizeof(Formation));
    tmp->nom = nom; tmp->resp = resp;
    tab[tabsp++]=tmp;
    return OK;}
  else return(PLEIN);
}
```


formation.c-suite

```
Formation * cherche_formation(char nom[]) {
  int i;
  for (i=0; i<tabsp;i++)
    {if (strcmp(tab[i]->nom,nom)==0) return tab[i];}
  return NULL;
}
Erreur modif_formation(char nom[],char resp[]) {
  Formation * tmp = cherche_formation(nom);
  if (tmp != NULL) {tmp->resp = resp; return(OK);}
  else return(INEXISTANT);
}
```


Le retrait d'un élément peut libérer la mémoire. On ne peut pas utiliser la fonction `cherche_formation` qui renvoie le pointeur sur la formation, alors que pour effectuer la permutation des deux cases du tableau, il nous faut un pointeur sur la case correspondante et donc un pointeur sur une pointeur sur une formation.

formation.c libération de la mémoire

```
Erreur retire_formation(char nom[]) {
  Formation ** tmp = NULL;
  int i;
  for (i=0; i<tabsp;i++)
    {if (strcmp(tab[i]->nom,nom)==0) tmp = &tab[i];}
  if (tmp != NULL) {
    free(tmp);
    tmp = tab[--tabsp];
    return(OK);
  }
  return(INEXISTANT);
}
```

formation.c-autres solutions

On peut choisir de trier les données suivant les clés.
Pas forcément utile pour 10 formations!
– Recherche dichotomique
– Insertion/suppression avec déplacement des données
On peut aussi implanter une table de hachage
– Accès direct

salles.c

Solution avec des indices de tableau

```
#include "type.h"
#include "salle.h"
#include <stdlib.h>
#include <stdio.h>
static Salle *tab [MAXSALLE];
static int tabsp = 0;
```


salles.c-recherche dichotomique

```
/* Cherche le plus petit i<=k<j tel que tab[k]->nom >= nom
   ou k = j */

static int index_dicho (char nom[],int i, int j) {
   int k,cmp;
   if (i == j) return i;
   k = (i + j)/2;
   cmp=strcmp(tab[k]->nom,nom);
   if (cmp > 0) index_dicho(nom,i,k);
   else if (cmp < 0) index_dicho (nom,k+1,j);
   else return k;
}
```


salle.c-fonctions auxiliaires

```
int index_salle (char nom[]) {
  return index_dicho(nom,0,tabsp);
  }
int comp_nom(int k,char nom[]) {
   if (tabsp <= k) return 1;
   else return strcmp(tab[k]->nom,nom);
}
```


salle.c-ajout

```
Erreur ajout_salle(char nom[],int capa) {
  int k,i;
  Salle *tmp;
  if (tabsp >= MAXSALLE) return PLEIN;
  k = index_salle(nom);
  if (comp_nom(k,nom)==0) return PRESENT;
  for (i=tabsp; k<i;i--) tab[i]=tab[i-1];
  tmp = (Salle *) malloc(sizeof(Salle));
  tmp -> nom = nom; tmp -> capacite = capa;
  tab[k]=tmp;tabsp++;
  return OK;
}
```


salle.c-retrait

```
Erreur retire_salle(char nom[]) {
  int k,i;
  k = index_salle(nom);
  if (comp_nom(k,nom)!=0) return INEXISTANT;
  free(tab[k]);
  for (i=k; i+1<k;i++) tab[i]=tab[i+1];
  tabsp--;
  return OK;
}
```


salle.c-Accès salle

```
Salle * cherche_salle(char nom[]) {
  int k = index_salle(nom);
  if (comp_nom(k,nom)!=0) return NULL;
  return tab[k];
}
int capacite_salle (char nom[]) {
  Salle * tmp = cherche_salle(nom);
  if (tmp != NULL) return (tmp -> capacite);
  return 0;
}
```


salle.c-Modification

```
Erreur modif_salle(char nom[],int capa) {
  Salle * tmp = cherche_salle(nom);
  if (tmp != NULL) {tmp->capacite = capa; return(OK);}
  return INEXISTANT;
}
```

Validation

Tests

Tests unitaires

Les tests unitaires servent à mettre en évidence les erreurs au niveau de chaque fonction.

Ils peuvent être exécutés module par module, par exemple au niveau de la fonction `main`.

Ils reposent sur une observation du résultat de la fonction et de l'état du système.
- Valeur exacte
- Ensemble de valeurs

Automatisation des tests

- Faire des fonctions de test
  Afficher l'énoncé du test, la valeur attendu, le résultat du test
  Il peut être utile de sortir du programme en cas d'erreur.
- Un fichier de tests unitaires séparé par module à tester
- Introduire des fonctions d'observation sur les variables statiques

Attention à la correction des fonctions de test et d'observation

Choix des tests unitaires

- Prendre des valeurs limites dans les domaines
- Prendre des valeurs générales
  utiliser des valeurs singulières dont le résultat est facile à calculer ou des outils indépendants
- Tester les différents chemins d'exécution du programme

Tests unitaires-test_salle.h

```
#include "type.h"
#include "salle.h"
int index_salle (char nom[]);
void test_index_salle (char nom[],int res) {
  int test = index_salle(nom);
  if (test!=res)
  printf ("Echec : index_salle(%s)=%d \n au lieu de %d\n",
          nom, test , res);
  else
  printf ("Succes : index_salle(%s)=%d \n", nom, test);
}
```


Tests unitaires-cherche_salle

Cas de valeur de retour non élémentaire

```
Salle * cherche_salle(char nom[]);
void test_cherche_salle_ok(char nom[],int capa) {
 Salle *test = cherche_salle(nom);
 if (test==NULL)
 printf("Echec: cherche_salle(%s)=NULL au lieu de (%s,%d)\n
           nom, nom, capa);
 else if (test->nom !=nom || test->capacite !=capa)
 printf("Echec: cherche_salle(%s)=(%s,%d) au lieu de (%s,%
           nom, test->nom, test->capacite , nom, capa);
 else printf("Succes: cherche_salle(%s)= (%s,%d)\n",nom,nom
}
```


Tests unitaires-cherche_salle-cas null

```
void test_cherche_salle_null(char nom[]) {
  Salle *test = cherche_salle(nom);
  if (test==NULL)
  printf ("Succes: cherche_salle(%s)= NULL\n", nom);
  else
  printf ("Echec: cherche_salle(%s)=(%s,%d)\n \
           Valeur attendue : NULL\n",
          nom,test->nom,test->capacite);
}
```


Tests de l'état

On peut vouloir tester les modifications apportées à l'état du système.

Il faut pouvoir observer l'état.

Dans `salle.c`

```
int taille () { return tabsp; }
char * acces_nom (int i) { return tab[i]->nom; }
int acces_capacite (int i) { return tab[i]->capacite; }
```

Dans `test_salle.c`

```
int taille ();
char * acces_nom (int i);
int acces_capacite (int i);
```

Tests de propriétés logiques

On peut tester les propriétés des fonctions telles qu'elles ont été spécifiées.

Exemple

La recherche dichotomique sur `nom` renvoie un entier k tel que

$$0 \leq k < tabsp \wedge \forall i,\ \ (0 \leq i < k \Rightarrow tab[i] \rightarrow \texttt{nom} < \texttt{nom})$$
$$\wedge$$
$$k \leq i < \texttt{tabsp}, \texttt{nom} \leq tab[i] \rightarrow \texttt{nom}$$

Attention, toutes les propriétés ne sont pas testables
(problème de finitude/décidabilité)

Tests de cherche_salle_dicho

En simplifiant les messages d'échec.

```
void test_index_salle_spec (char nom[]) {
int test = index_salle(nom), i;
if (test < 0 || taille() < test) {printf ("Echec");
return;}
for (i=0; i<test; i++)
  if (strcmp(acces_nom(i),nom)>0) {printf ("Echec"); return
for (i=test; i<taille(); i++)
  if (strcmp(acces_nom(i),nom)<0) {printf ("Echec"); return
printf ("Succes : index_salle(%s)\n",nom);}
```


Mise en place des tests

Initialisation

```
test_ajout_salle("336-044",50,OK);
test_ajout_salle("308-001",20,OK);
test_cherche_salle_ok("308-001",20);
test_cherche_salle_ok("336-044",50);
```

Test des différents cas d'erreur

```
test_ajout_salle("336-044",10,PRESENT);
```


Remplissage de la table

Ajouter le nombre maximum de salles

```
for (i=taille(); i<MAXSALLE; i++) {
    s=calloc(10,sizeof(char));
    sprintf(s,"Salle %d",i);
    test_ajout_salle (s,i,OK);
  }
test_ajout_salle("Plein",10,PLEIN);
```

Remarque Que se passe-t-il si on oublie le `calloc` ?

Systématiser les tests

- Associer à chaque module `fichier.c` un fichier `test_fichier.c` définissant les fonctions de tests.
  Celles-ci ne dépendent que de l'interface `fichier.h`.
- Créer des scénarios pour tester les fonctions individuellement ou en interaction dans la fonction `main` de différents fichiers de test.
- Rejouer régulièrement les tests.
- Surveiller les sorties
  (redirection vers un fichier, recherche de `Echec`)

Tests du système

On peut également faire les tests de manière externe sur le programme exécutable.

- Prévoir des options et arguments passés à l'exécutable pour observer les différentes fonctionalités
- Utiliser le langage du shell pour créer des fonctions de test
- Tests correspondant aux bugs identifiés
- Test de non-régression :
  On compare le comportement du système en cours d'évolution avec une version de référence.
  S'applique à des systèmes stables.

Bibliothèques générales

Une bibliothèque pour les tables d'associations

Les tables utilisées pour les différents ensembles (salles, groupes, formations) sont similaires.

- Factoriser le code commun
- Possibilité de développer des algorithmes efficaces
- Abstraction par rapport au type des données stockées

Analyse du code

- Une table `tab` de taille max `MAX`
- Un variable stockant l'indice de la première case libre
- Données triées suivant un ordre sur les index
- Ajout, recherche, retrait d'un élément identifié par une clé de type `char *`.

Utiliser des objets de type `void *` pour rendre les algorithmes génériques.

Abstraction de `index_dicho`

Cas instancié `salle.h` :

```
static Salle *tab [MAXSALLE];
static int tabsp = 0;
static int index_dicho_salle (char nom[],int i,int j);
```

Cas général (Essai) `table.h` :

```
int index_dicho (void *tab[],int *tabsp, char nom[],int i,i
```


Abstraction de la fonction de comparaison

Le code de `index_dicho_salle` utilise une construction spécifique

```
strcmp(tab[k]->nom,nom);
```

Généraliser en une fonction de comparaison `comp` :

```
comp(tab[k],nom);
```

Passer la fonction `comp` en argument à `index_dicho`
(pointeur de fonction)

Notations pour les pointeurs de fonction

<type> (*`ident`) (*<type$_1$>*,...,*<type$_n$>*)

Déclare `ident` comme un pointeur sur une fonction qui attend des arguments de type *<type$_1$>*, ..., *<type$_n$>* et renvoie une valeur de type *<type>*.
Par exemple pour introduire la variable f qui est un pointeur sur une fonction qui attend deux entiers et renvoie un entier, on écrit :

```
int (*f) (int,int)
```

Attention à ne pas confondre avec :

```
int *f (int,int)
```


Fonction comme argument

On peut passer un pointeur sur une fonction en argument d'une autre fonction. La syntaxe du type de la variable sera alors :

<type> (*) (*<type$_1$>*,...,*<type$_n$>*)

Le prototype d'une fonction `hsf` qui prendrait la fonction `f` précédente en argument s'écrira donc :

```
void Hsf(int (*) (int,int) f)
```


Cas de la fonction de comparaison

Dans notre cas, la fonction de comparaison `comp` paramétrique prend en argument un pointeur sur `void` et un pointeur sur `char` et renvoie un entier.

On doit donc avoir :

```
int (*comp) (void *, int)
```

Afin de simplifier les notations, on peut introduire une notation de type :

```
typedef int (*Comp)(void *,char *);
```


Prototypes des fonctions de la bibliothèque

```
int index_table (void *tab[],int *tabsp, Comp comp, char nc
Erreur retire_table (void *tab[],int *tabsp,Comp comp,char
```

`cherche_table` renvoie un pointeur sur `void` qui sera plus tard converti vers un pointeur sur la structure instanciée.

```
void *cherche_table (void *tab[],int *tabsp, Comp comp, cha
```


Prototype de `ajout_table`

Dans `salle.h`

```
Erreur ajout_salle(char nom[],int capa);
```

Comment abstraire `capa` ?
La fonction `ajout_salle` crée un nouvel objet qu'elle instancie et insère dans le tableau.

Solution séparer la création d'objet (particulière au type de donnée) et l'ajout dans la table (générique).
La valeur à insérer est passée en argument à `ajout_table`.

```
Erreur ajout_table
 (const int MAX,void *tab[],int *tabsp,Comp comp,
  char nom[],void *val);
```

Code générique-index_dicho

```
static int index_dicho
   (void *tab[],int *tabsp,Comp comp,char nom[],int i,int j
   int k,cmp;
   if (i == j) return i;
   k = (i + j)/2;
   cmp=(*comp)(tab[k],nom);
   if (cmp > 0) index_dicho(tab,tabsp,comp,nom,i,k);
   else if (cmp < 0) index_dicho (tab,tabsp,comp,nom,k+1,j
   else return k;
}
int index_table (void *tab[],int *tabsp,Comp comp,char nom
  return index_dicho(tab,tabsp,comp,nom,0,*tabsp);
}
```


Code générique-cherche_table

```
void * cherche_table
    (void *tab[],int *tabsp, int (*comp) (void *,char *), c
    int k;
    k = index_table(tab,tabsp,comp,nom);
    if (k==*tabsp || (*comp)(tab[k],nom)!=0) return NULL;
    return tab[k];
}
```


Code générique-retire_table

```
Erreur retire_table(void * tab[],int *tabsp,Comp comp,char
  int k,i;
  k = index_table(tab,tabsp,comp,nom);
  if (k==*tabsp || (*comp)(tab[k],nom)!=0) return INEXISTAN
  free(tab[k]);
  for (i=k; i+1<k;i++) tab[i]=tab[i+1];
  (*tabsp)--;
  return OK;
}
```


Code générique-ajout_table

```
Erreur ajout_table
  (const int MAX,void *tab[],int *tabsp,Comp comp,char nom[
{ int k,i;
  k = index_table(tab,tabsp,comp,nom);
  if ( (k!=*tabsp) && (*comp)(tab[k],nom)==0) return PRESEN
  if (*tabsp >= MAX) return PLEIN;
  for (i=*tabsp; k<i;i--) tab[i]=tab[i-1];
  tab[k]=val;(*tabsp)++;
  return OK;
}
```


Instanciation pour les salles

```
static Salle *tab [MAXSALLE];
static int tabsp = 0;

int comp_salle(Salle *s,char nom[]) {
  return strcmp(s->nom,nom);}
```

Donner à ces objets des types génériques :

```
void **ptab=(void **)tab;
int *ptabsp = &tabsp;
Comp comp = (Comp)comp_salle;
```


Instanciation des fonctions

```
int index_salle(char nom[]) {
  return index_table(ptab,ptabsp,comp,nom);
}
Salle * cherche_salle(char nom[]) {
  return (Salle *)cherche_table(ptab,ptabsp,comp,nom);
}
Erreur retire_salle(char nom[]) {
  return retire_table(ptab,ptabsp,comp,nom);
}
```


Cas de la fonction ajout_salle

```
Salle *cree_salle(char nom[],int capa) {
    Salle *val;
    val=(Salle *) malloc(sizeof(Salle));
    val->nom=nom;val->capacite=capa;
    return val;
}
Erreur ajout_salle(char nom[],int capa) {
    Salle *val;
    Erreur res;
    val=cree_salle(nom,capa);
    res = ajout_table(MAXSALLE,ptab,ptabsp,comp,nom,val);
    if (res) free(val);
    return res;
}
```

Robustesse-Évolution

Évolution

Le système doit pouvoir être modifié.

– Modification de l'implantation des types de base sans modification des interfaces

Exemple
Nos différentes implantations de `table.c` ne nécessitent pas de modifier `table.h` ou bien `test_table.h` ni les autres fonctions qui en dépendent.

Évolution

Modification des fonctionalités

– Au cours du temps, le système doit pouvoir être enrichi sans remettre en cause l'architecture générale.

Exemple
On peut ajouter des caractéristiques d'équipement à chaque salle (ordinateurs, écran, rétro-projecteur ...). Les algorithmes de recherche ne seront pas modifiés, il suffit de remplacer la capacité qui est un entier par une structure d'équipement et d'effectuer la recherche en fonction d'un ordre sur cette structure. Les possibilités d'évolution doivent être anticipées à la conception.
Un code plus abstrait sera souvent plus lisible et plus évolutif.

Persistance des données

Les tables ne restent en mémoire que le temps d'exécution du programme.

– Nécessité de sauvegarder les données sur le disque
– Format lisible pour communication avec d'autres outils, traitements syntaxiques.
Le format texte est souvent le plus adapté.

Des formats standards :
– Structure linéaire en tableau (Format CSV : Comma Separated Value)
– Indexation par des étiquettes
– Structuration arborescente avec balises (XML)

Exemple

Format CSV pour les salles

```
336-044,45
308-001,30
"336 042",21
```

Format avec étiquette (autorise des infos partielles)

```
{ nom = "336-044"
  capacite = "45"}
{ nom = "308-001"
  capacite = "30"}
{ nom = "336 042"
  capacite = "21"}
```


Écriture dans un fichier

Format étiqueté


```
void ecrit_salles(FILE *fp) {
  int i;
  for (i=0;i<tabsp;i++){
    fprintf(fp,"{ nom = \"%s\"\n  capacite = \"%d\" }\n",
            tab[i]->nom,tab[i]->capacite);
  }
}
void sauve_salles(char *fichier) {
  FILE * fp=fopen(fichier,"w");
  if (fp==NULL)
    fprintf(stderr, "Impossible d'ouvrir le fichier %s", fi
  else {ecrit_salles(fp); fclose(fp); }
}
```


Résultat

```
{ nom = "308-001"
  capacite = "20" }
{ nom = "336-044"
  capacite = "40" }
{ nom = "Salle 10"
  capacite = "10" }
{ nom = "Salle 11"
  capacite = "11" }
```

Fonction de lecture

Utilise une lecture par ligne

```
char * fgets(char *s, int n, FILE *fp)
```

Reconstruit des chaînes ou des entiers entre les guillemets,
renvoie `NULL` en cas d'erreur ou de fin de fichier

```
char *lit_chaine(FILE *fp,char *res);
int *lit_entier(FILE *fp, int *i);
```


Code de lecture d'une chaîne

```
#define MAXLIGNE 80
static char ligne[MAXLIGNE];
char *lit_chaine(FILE *fp,char *res) {
  char *deb, *fin;
  char *li=ligne;
  if (fgets(li,MAXLIGNE,fp)==NULL) return NULL;
  deb = strchr(li,'\"');
  fin = strrchr(li,'\"');
  ++deb;*fin='\0';
  res=calloc(fin-deb+1,sizeof(char));
  strncpy(res,deb,fin-deb+1);
  return(res);
}
```


Code de lecture d'un entier

```
int *lit_entier(FILE *fp, int *i) {
  char *deb, *fin;
  char *li=ligne;
  if (fgets(li,MAXLIGNE,fp)==NULL) return NULL;
  deb = strchr(li,'\"');
  fin = strrchr(li,'\"');
  ++deb;
  sscanf(deb,"%d",i);
  return(i);
}
```


Mise à jour d'une salle

```
Erreur lit_salle(FILE *fp) {
  char *nom;
  int capa;
  nom=lit_chaine(fp,nom);
  if (nom==NULL) return FIN;
  lit_entier(fp,&capa);
  if (tabsp < MAXSALLE) {tab[tabsp++]=cree_salle(nom,capa);
  else return PLEIN;
}
```


Mise à jour de la table

```
int lit_salles(char *fichier) {
 FILE * fp;
 Erreur err;
  if ((fp=fopen(fichier,"r"))==NULL)
    fprintf(stderr, "Impossible d'ouvrir le fichier %s", fi
  else {
    for (err=lit_salle(fp);err!=FIN;err=lit_salle(fp));
    fclose(fp);
    }
}
```


Conclusion

Conclusion

- Un survol des fonctionnalités du langage C
  - Fonctionnalités élémentaires
  - Fonctionnalités avancées (gestion mémoire, généricité)
- Principes de programmation
  - Structurer et abstraire les données et les développements (interface)
  - Approche systématique du développement

À renforcer par la pratique, la lecture de codes de référence !!

Ce qu'il faut retenir

- Savoir lire et écrire une description syntaxique (expressions régulières, règles BNF)
- Types principaux du langage C
- Appel de fonctions, passage de paramètres
- Organisation de la mémoire : variable, pointeur, valeur
- Codage en C d'algorithmes élémentaires sur les tableaux et les structures chaînées
- Entrées/Sorties élémentaires
- Découper un projet en fichiers, écrire des interfaces, choisir des tests