RAPPORT
TECHNIQUES DE PROGRAMMATION
LOGICIEL D'AIDE À LA GESTION DES SERVICES DE SECOURS

GOLOVAN Mykola et LACROIX Rémi
GROUPE 2 (B)

Introduction

Les structures mises en place

Pour pouvoir bien structurer et gérer le développement du logiciel, nous avons décidé de séparer la structure correspondante aux alertes, la structure correspondante aux unités et toutes les données nécessaires qui traitent chacune d'elles. Autrement dit, nous avons modulé deux fichier d'interface *alerte.h* et *unite.h*.

Le fichier d'interface alerte.h

Nous avons protégé ce fichier contre les inclusions multiples en utilisant l'approche suivante :

```
#ifndef ALERTE_H
#define ALERTE_H
#include <stdio.h>

/* corps du fichier interface alerte.h */

#endif
```

Ensuite, nous avons défini la structure *Alerte*. Elle a pour but de faire des traitements sur les alertes dans le logiciel de secours. La représentation de cette structure est la suivante :

```
typedef struct {
	int iCode;
	char cType[TAILLE_BUFFER];
	char cNiveau[TAILLE_BUFFER];
	char cLieu[TAILLE_BUFFER];
	int iNombreVictimes;
	char cDescription[TAILLE_BUFFER];
```

```
    int iEstTraiteParUnite;
    int iNbUniteQuiTraite;
    int *iCodeUniteQuiTraite;
} Alerte;
```

Nous avons déclaré au total 9 attributs de la structure *Alerte.* Selon le cahier de charges, chaque alerte doit contenir un code individuel généré par le logiciel. Le premier membre est donc `int iCode` qui sert pour le stockage du code aléatoire généré par le logiciel. Les membres suivants :

```
char cType[TAILLE_BUFFER];
char cNiveau[TAILLE_BUFFER];
char cLieu[TAILLE_BUFFER];
char cDescription[TAILLE_BUFFER];
```

ont pour but de stocker l'information contenant le type, niveau, lieu et description d'alerte. Ils ont une TAILLE_BUFFER qui contient 150 cases vides pour les chaînes de caractères. Ensuite, il y a un entier int `iNombreVictimes` qui stocke le nombre de victimes et deux autres entiers :

```
int iEstTraiteParUnite;
int iNbUniteQuiTraite;
int *iCodeUniteQuiTraite;
```

Puisque nous n'avons pas vraiment le type `boolean` dans le langage C et pour éviter d'utiliser d'autres librairies qui contiennent ce type, nous avons décidé de simplifier les choses et ne pas surcharger le logiciel. Nous utilisons une variable de type entier qui va définir avec les valeurs 0 (faux) et 1 (vrai) si une alerte est traitée par une unite `int iEstTraiteParUnite;`. Et une autre variable qui stocke le code d'unité qui traite une alerte donnée.

Enfin, nous avons un tableau dynamique d'entier `int *iCodeUniteQuiTraite` qui stocke tous les codes des alertes qui traitent une alerte donnée.

La liste de fonctions

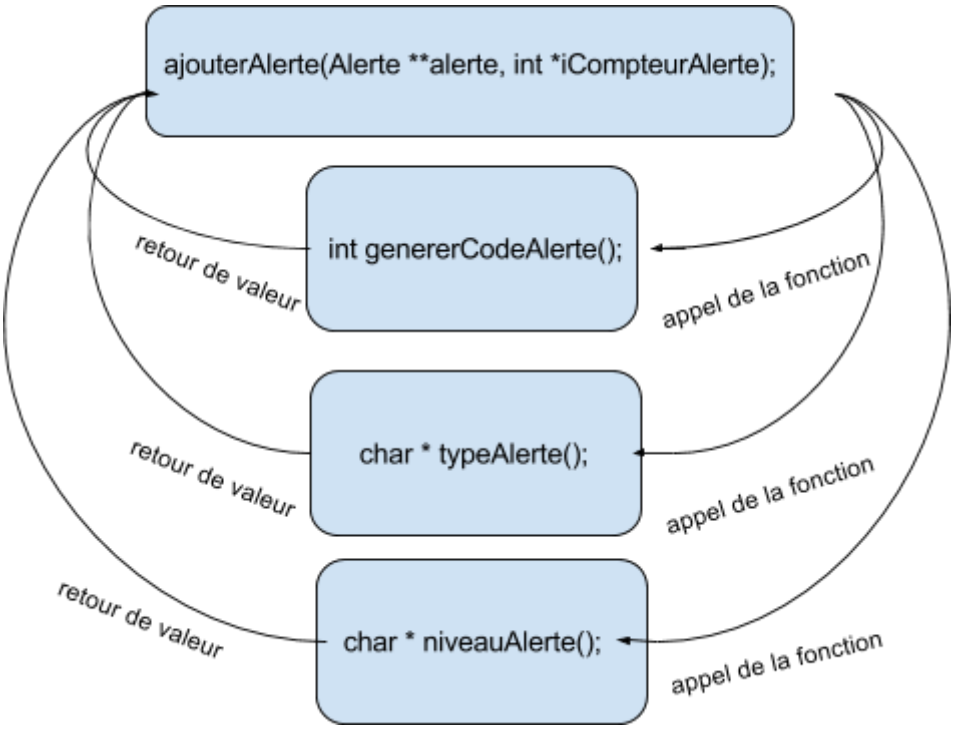
Nous avons déclaré 12 fonctions au total qui permettent de traiter une alerte :

```
void ajouterAlerte(Alerte **alerte, int *compteurAlerte);
void afficherUneAlerte(Alerte **alerte, int *compteurAlerte);
void afficherToutesAlertes(Alerte **alerte, int *compteurAlerte);
void modifierAlerte(Alerte **alerte, int *compteurAlerte);
void supprimerAlerte(Alerte **alerte, int *compteurAlerte);
char * typeAlerte();
char * niveauAlerte();
int genererCodeAlerte();
int alertesStatiques(Alerte **alerte, int *iCompteurAlerte, int
                     *iLesAlertesDefinis);
void imprimerLesAlertes(Alerte **alerte, int *iCompteurAlerte);
void chargerLesAlertes(Alerte **alerte, int *iCompteurAlerte);
void lesAlertesTraitee(Alerte **alerte, int *iCompteurAlerte);
```

La fonction `ajouterAlerte(Alerte **alerte, int *compteurAlerte);` sert pour la création d'une nouvelle alerte. Elle est de type `void` et donc elle ne retourne pas de valeurs. Elle prend en paramètre un pointeur de pointeur de la structure *Alerte* `Alerte **alerte` et un pointeur de type entier `int *iCompteurAlerte` qui donne le nombre de cases total dans le tableau dynamique d'*Alerte* et qui sert également pour le parcours du tableau et le traitement des cases. Cette fonction fait aussi appel aux fonctions supplémentaires qui contiennent des informations par rapport aux alertes. Ce sont les fonctions utilisées à l'intérieur de la fonction :

```
char * typeAlerte();
char * niveauAlerte();
int genererCodeAlerte();
```

Elles ne prennent pas de paramètres mais elles retournent un pointeur de type `char`. Ces pointeurs contiennent une chaîne de caractères selon le choix que l'utilisateur a saisi et un entier `int` qui est le code aléatoire de l'alerte qui vient d'être créée. L'ensemble du déroulement de ces fonctions a la forme schématique suivante :



Les fonctions `*typeAlerte();` et `*niveauAlerte();` retournent un pointeur de type de chaîne de caractères selon le choix de l'utilisateur.

Les fonctions

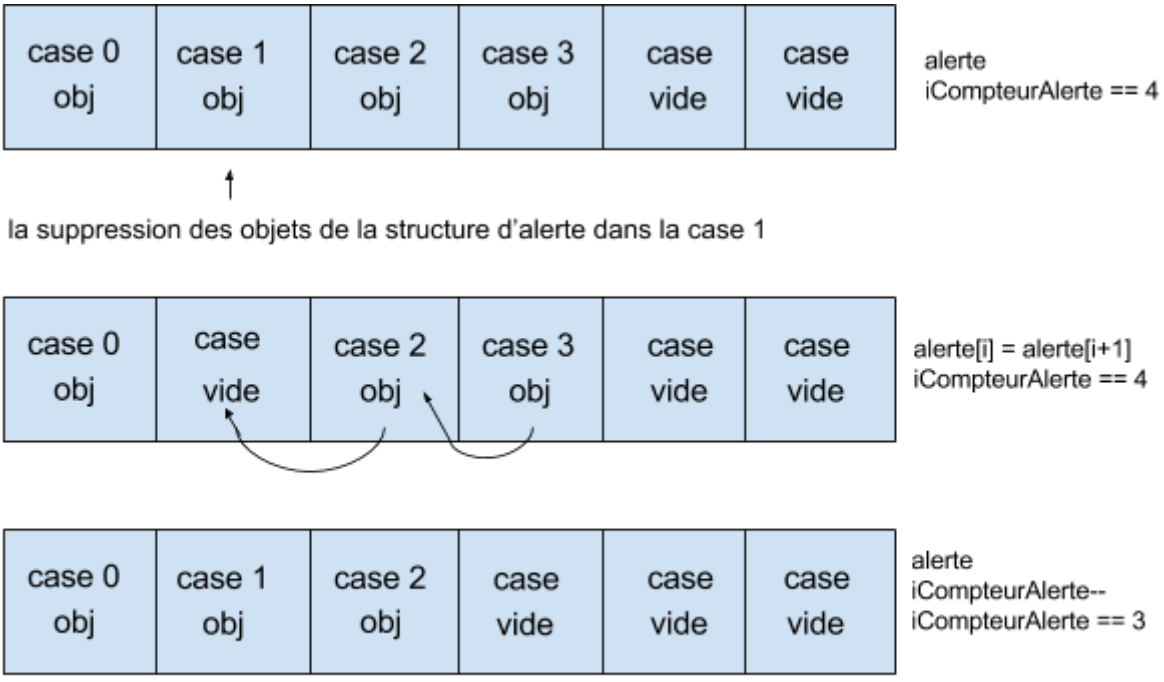
```
void afficherUneAlerte(Alerte **alerte, int *iCompteurAlerte);
void afficherToutesAlertes(Alerte **alerte, int *iCompteurAlerte);
```

servent pour l'affichage des alertes. En utilisant un entier `int *iCompteurAlerte,` ces fonctions parcourent le tableau de la structure *Alerte* et affichent les résultats trouvés ou un message correspondant s'il n'y a aucune alerte enregistrée. Elle ne retournent pas de valeurs et elles prennent en paramètre un pointeur de pointeur de la structure *Alerte* et un entier `int *iCompteurAlerte` qui donne le nombre de cases

dans le tableau dynamique. Dans la première fonctions la recherche est basée sur le code donné par l'utilisateur alors que la deuxième fonction affiche toutes les alertes du tableau.

La fonction `void modifierAlerte(Alerte **alerte, int *compteurAlerte);` ne retourne aucune valeur et prend en paramètre un pointeur de pointeur de la structure *Alerte* `Alerte **alerte` et un pointeur d'entier `int *compteurAlerte` de compteur qui correspond au nombre de cases du tableau. Cette fonction parcourt le tableau de la structure alerte et affiche une alerte selon le code qu'il a saisi. Si le code est faux ou que l'alerte n'existe pas, le message correspondante sera affiché à l'écran. Sinon, l'utilisateur modifie l'information de l'alerte trouvée ; excepté le code qui a été donnée à la création de cette alerte.

La fonction `void supprimerAlerte(Alerte **alerte, int *compteurAlerte);` enlève une alerte du tableau dynamique selon le code saisi par l'utilisateur. Elle ne retourne aucun valeur et elle prend en paramètre un pointeur de pointeur de la structure *Alerte* `Alerte **alerte` et un pointeur d'entier `int *compteurAlerte` qui indique le nombre de cases du tableau et qui sert à parcourir ce-dernière. Après la suppression, la mémoire est réallouée à nouveau et tous les éléments du tableau sont déplacés afin d'éviter des éléments manquants.



le tableau de la structure alerte après la suppression

La fonction `int alertesStatiques(Alerte **alerte, int *iCompteurAlerte, int *iLesAlertesDefinis);` ne retourne aucun valeur et prend en paramètre un pointeur de pointeur de la structure *Alerte* et un pointeur d'entier `int *iCompteurAlerte` ainsi qu'un booléen. Le but de cette fonction est d'initialiser quelques alertes statiques afin de faciliter les tests du logiciel. De cette façon, en lançant le logiciel, il y aura déjà quelques données prédéfinies.

La fonction `void imprimerLesAlertes(Alerte **alerte, int *iCompteurAlerte);` parcours le tableau de la structure d'*Alerte* et écrit dans le fichier .txt toutes les alertes qui sont enregistrés dans le programme. Elle ne retourne pas de valeur et elle prend en paramètres un pointeur de pointeur de la structure *Alerte* `Alerte **alerte` et un entier `int *iCompteurAlerte` qui sert pour le parcours des cases du tableau dynamique.

La fonction `void chargerLesAlertes(Alerte **alerte, int *iCompteurAlerte);` charge d'un fichier externe .txt les alertes saisi par l'utilisateur. Les alertes ne sont pas enregistrées dans le programme, la

fonction ne fait qu'afficher les données. Elle ne retourne pas de valeur, elle prend en paramètres un pointeur de pointeur de la structure *Alerte* `Alerte **alerte` et un entier `int *iCompteurAlerte` qui sert pour le parcours des cases du tableau dynamique.

Une autre fonction `libererMemoireAlerte(Alerte **unite);` sert pour la libération de la mémoire de la structure *Alerte*. Elle ne retourne pas de valeurs et elle prend un pointeur de pointeur de la structure Unite en paramètre.

La fonction `void lesAlertesTraitee(Alerte **alerte, int *iCompteurAlerte);` sert à définir si les alertes statiques ont été intiialisées. Elle ne retourne aucune valeur et elle prend en paramètre un pointeur de pointeur de la structure Alerte et un pointeur du booléen.

Le fichier d'interface unite.h

Nous avons protégé ce fichier contre les inclusions multiples en utilisant l'approche suivante :

```
#ifndef UNITE_H
#define UNITE_H
#include <stdio.h>

/* corps du fichier interface unite.h */

#endif
```

Ensuite, nous avons défini la structure *Unite*. Elle a pour but de faire des traitements sur les unités dans le logiciel de secours. La représentation de cette structure est la suivante :

```
typedef struct {
  int iCode;
  char cNom[TAILLE_BUFFER];
  char cMoyenDeplacement[TAILLE_BUFFER];
  char cNiveauDisponibilite[TAILLE_BUFFER];
  char cStatut[TAILLE_BUFFER];
  char cBase[TAILLE_BUFFER];
  int iCompteurRepos;
  int iEstEnRepos;
  int iUniteDisponible;
  int iDeployeeSurAlerte;
} Unites;
```

Nous avons déclaré au total 10 attributs de la structure *Unites.* Selon le cahier des charges, chaque unité doit contenir un code individuel généré par le logiciel. Le premier membre est donc `int iCode` qui sert au stockage du code aléatoire généré par le logiciel. Les attributs suivants :

```
char cNom[TAILLE_BUFFER];
char cMoyenDeplacement[TAILLE_BUFFER];
char cNiveauDisponibilite[TAILLE_BUFFER];
char cStatut[TAILLE_BUFFER];
char cBase[TAILLE_BUFFER];
```

ont pour but de stocker l'information contenant le nom, le moyen de déplacement, le niveau de disponibilité, le statut et la base d'unité. Ils ont une `TAILLE_BUFFER` qui contient 150 cases vides pour les chaînes de caractères. Ensuite, il y a un entier `int iCompteurRepos` qui stocke le nombre de missions effectuées et un autre entier :

`int iDeployeeSurAlerte` qui stocke le code d'alerte que traite cette unité.

Puisque nous n'avons pas vraiment le type `boolean` dans le langage C et afin d'éviter d'utiliser d'autres librairies qui contiennent ce type, nous avons décidé de simplifier les choses et de ne pas surcharger le logiciel. Nous utilisons une variable de type entier qui va définir avec les valeurs 0 (faux) et 1 (vrai) si une unité est en repos `int iEstEnRepos` et si elle est disponible `int iUniteDisponible`.

La liste de fonctions

Nous avons déclaré 14 fonctions au total qui permettent de traiter une unité :

```
void creerUnite(Unites **unite, int *compteurUnite);
void afficherUneUnite(Unites **unite, int *compteurUnite);
void afficherToutesUnites(Unites **unite, int *compteurUnite);
void modifierUnite(Unites **unite, int *compteurUnite);
void supprimerUnite(Unites **unite, int *compteurUnite);
char * moyenDeplacement();
char * niveauDisponibilite();
char * statutUnite();
int genererCodeUnite();
void mettreUniteEnRepos(Unites **unite, int *iCompteurUnite);
```

```
void unitesStatiques(Unites **unite, int *iCompteurUnite, int
*iLesUnitesDefinis);
void consulterUnitesDeployees(Unites **unite,int *iCompteurUnite);
void imprimerLesUnites(Unites **unite, int *iCompteurUnite);
void chargerLesUnites(Unites **unite, int *iCompteurUnite);
```

La fonction `void creerUnite(Unites **unite, int *compteurUnite);` sert pour la création d'une nouvelle unité. Elle est de type `void` et donc elle ne retourne pas de valeurs. Elle prend en paramètre un pointeur de pointeur de la structure *Unites* `Unites **unite` et un pointeur de type entier `int *compteurUnite` qui donne le nombre de cases total dans le tableau dynamique d'*Unites* et qui sert également pour le parcours du tableau et le traitement des cases. Cette fonction fait aussi appel aux fonctions supplémentaires qui contiennent des informations par rapport aux unités. Ce sont les fonctions utilisées à l'intérieur de la fonction :

```
char * moyenDeplacement();
char * niveauDisponibilite();
char * statutUnite();
int genererCodeUnite();
```

Elles ne prennent pas de paramètres mais elles retournent un pointeur de type `char`, ces pointeurs contiennent une chaîne de caractères selon le choix que l'utilisateur a saisi et un entier `int` qui est le code aléatoire de l'unité qui vient d'être créée. L'ensemble de déroulement de ces fonctions ont une forme schématique suivante.

La fonction `int alertesStatiques(Unites **unite, int *iCompteurAlerte, int *iLesUnitesDefinis);` ne retourne aucun valeur et prend en paramètre un pointeur de pointeur de la structure *Unite* et un pointeur d'entier `int *iCompteurUnite` ainsi qu'un booléen. Le but de cette fonction est d'initialiser quelques unités statiques afin de faciliter les teste du logiciel. De cette façon, en lançant le logiciel il y aura déjà quelques données prédéfinies.

La fonction `void imprimerLesUnites(Unites **unite, int *iCompteurUnite);` parcourt le tableau de la structure d'*Unite* et écrit dans le fichier .txt toutes les alertes qui sont enregistrés dans le programme. Elle ne retourne pas de valeur et elle prend en paramètre un pointeur de pointeur de la structure *Unite* `Unites **unite` et un entier `int *iCompteurUnite` qui sert pour le parcours des cases du tableau dynamique.

La fonction `void chargerLesUnites(Unites **unite, int *iCompteurUnite);` charge d'un fichier externe .txt les unités saisi par l'utilisateur. Les unités ne sont pas enregistrée dans le programme, la fonction ne fait qu'afficher les données. Elle ne retourne pas de valeur, elle prend en paramètres un pointeur de pointeur de la structure *Unite* `Unites **unite` et un entier `int *iCompteurUnite` qui sert pour le parcours des cases du tableau dynamique.

Les fonctions neutres dans le fichier menu.c

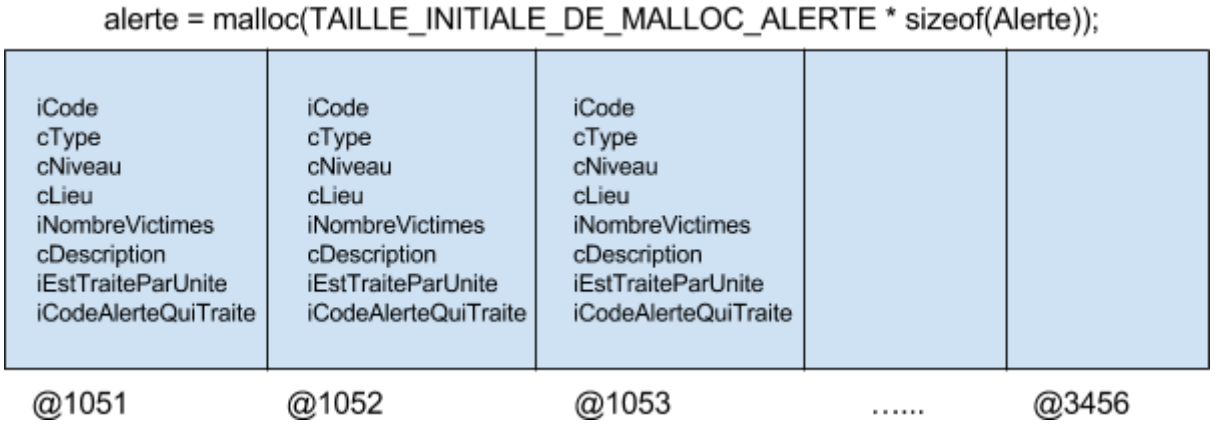
```
void afficherAlerte(Alerte alerte);
void afficherUnite(Unites unite);
void libererMemoireAlerte(Alerte **alerte);
void libererMemoireUnite(Unite **unite);
```

Ne retournent pas de valeurs, prennent en paramètre les tableaux d'Alerte et d'Unites et qui ne contiennent que les textes d'affichage pendant le parcours d'une boucle dans une autre fonction.

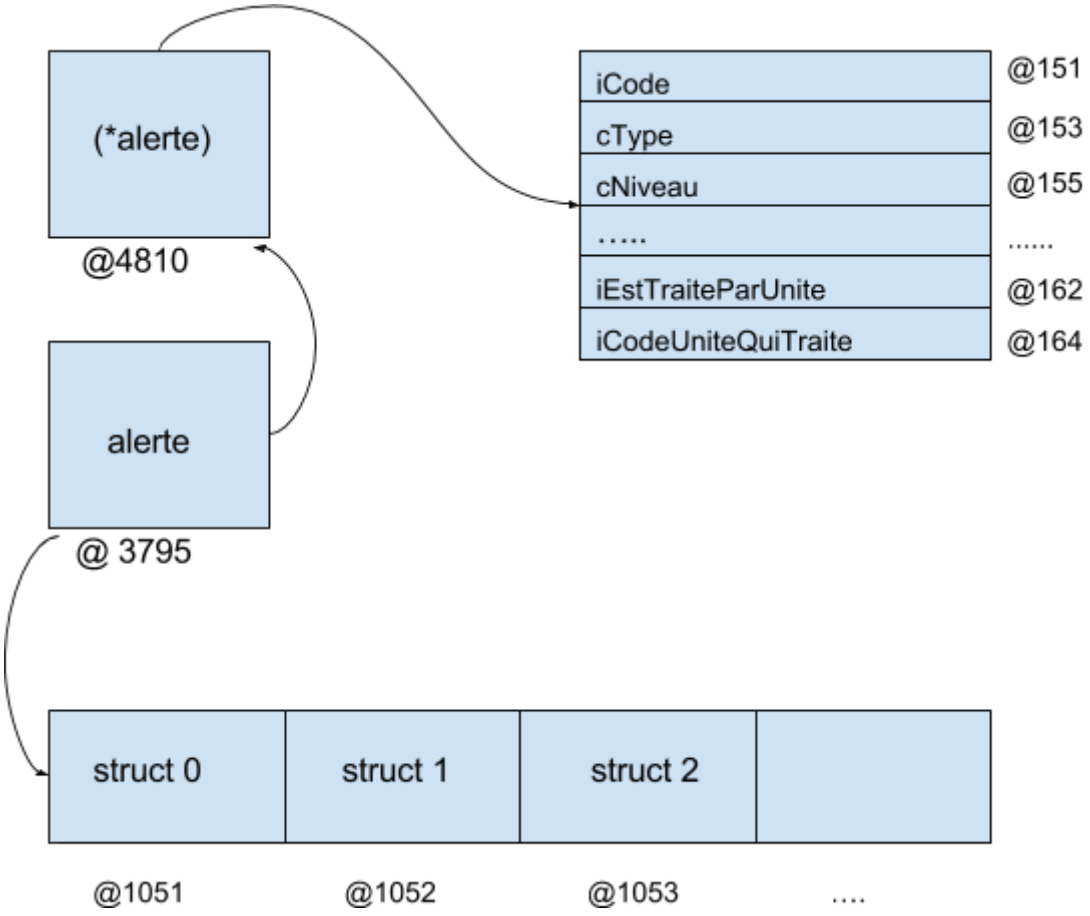
Les autres fonctions `libererMemoireAlerte(Alerte **alerte);` et `libererMemoireUnite(Unite **unite);` servent pour la libération de la mémoire de la structure Alerte et Unite. Elles ne retournent pas de valeurs et elles prennent un pointeur de pointeur de la structure Unite en paramètre et de la structure Alertes.

ORGANISATION DES STRUCTURES EN MÉMOIRE

Pour la réalisation de ce projet, nous avons décidé d'utiliser les tableaux dynamiques pour la gestion des structures. Nous avons déclaré des variables correspondantes à la taille des tableaux dynamiques pour la gestion des cases de chaque structure. La représentation schématique des structures est la suivante :



Nous avons alloué un bloc de mémoire de taille 3 pour le tableau dynamique de la structure Alerte. Pour manipuler tous les éléments de la structure dans les autres fonctions, nous avons utilisé un pointeur de pointeur de la structure Alerte. Voici une représentation schématique de cela :



Selon ce schéma, nous pouvons donc dire que le pointeur *alerte*

pointe sur l'adresse de la structure du tableau dynamique et le pointeur de pointeur de *Alerte (*alerte)* sert dans les fonctions pour accéder dans l'adresse des cases du tableau.

ORGANISATION DE TRAVAIL

Pour faciliter le développement du logiciel et pour un bon avancement, nous avons décidé de partager le travail également selon le cahier de charges. Vu que les structures d'Alerte et d'Unites se ressemblent, Mykola GOLOVAN a développé la structure Alerte avec toutes les fonctions nécessaires et Rémi LACROIX s'est lancé dans la conception de la partie Unités. Nous avons structuré notre travail de façon suivant :

Première étape :

1) Développement des structures
2) Conception des fonctions pour chaque structure
3) Collaboration des fichiers
4) Testes des ensembles de fonctions

Deuxième étape :

5) Conception des fonctions de la gestion globalisée

Dans cette partie, Mykola GOLOVAN a travaillé sur le développement de la fonction de déclenchement d'une alerte et Rémi LACROIX a pris la fonction de la gestion des unités déployées sur une alerte.

6) Collaboration des fonctions et les tests généraux
7) Tests des logiciel, corrections des bugs trouvés, optimisation du code

Dans ce dernière étape nous avons consacré la plupart de temps au tests, à la corrections des bugs et à l'optimisation du code. Dans quelques parties le code peut toujours paraître un peu lourd mais nous avons fait de notre mieux pour pouvoir représenter et écrire le code le

plus clair, lisible et optimisé possible.

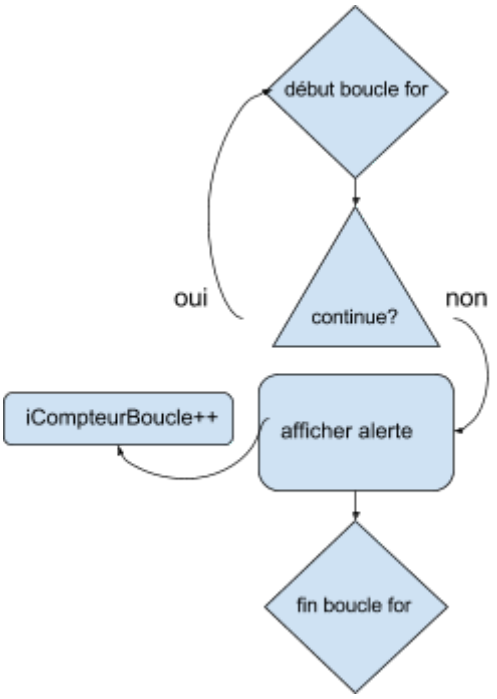
MÉCANISME DE RECHERCHE D'UNE ALERTE QUI EST EN COURS DE TRAITEMENT

Le mécanisme mis en place est assez simple et ne contient qu'une seule boucle. Nous utilisons une boucle `for` pour parcourir le tableau dynamique de la structure *Alerte*. Pendant le parcours nous utilisons une instruction `continue` qui permet de sauter toutes les alertes qui ne sont pas en cours de traitement par les unités. Voici la condition qui nous permet de faire ce traitement :

```
if((*alerte)[i].iEstTraiteParUnite == 0)
      continue;
```

S'il n'y a aucune alerte en traitement, le message correspondant est donc affiché à l'écran. Pour vérifier cela, nous utilisons une autre variable `int iCompteurBoucle` que nous incrémentons à chaque tour de boucle. Si cette variable est égal à 0 à la fin, il n'y a donc aucune alerte qui est en cours de traitement.

Sous la forme schématique cela donne la structure suivante :



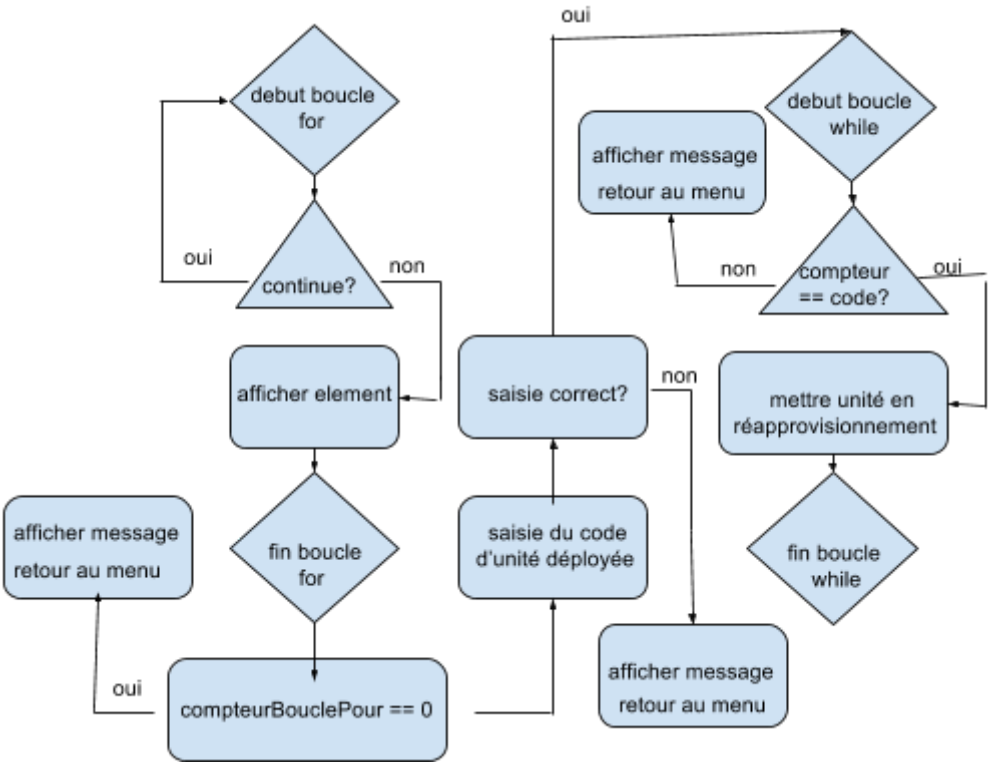
MÉCANISME DE GESTION DES UNITÉS DÉPLOYÉES

La description textuelle :

La fonction responsable de la gestion des unités déployées est `void consulterUnitesDeployees(Unites **unite, int *iCompteurUnite);`. Elle fait un parcours dans le tableau de la structure Unites en utilisant une boucle `for` . Le parcours lui-même est le suivant : dans la boucle, il y a une condition `if` qui vérifie si l'unité trouvée est disponible ou non. Puisque nous parcourons le tableau entièrement, il faut bien séparer les unites disponibles et les unites qui sont en train de traiter une alerte. Notre solution a été de mettre une opération `continue` qui va sauter chaque unité qui est disponible. De cette façon, il ne sera affiché que les équipes qui sont en alerte. Il faut prendre en compte aussi qu'il peut n'y avoir aucune unité déployée. Pour cela, nous avons mit un compteur de boucle et, si et seulement si, à la fin de la boucle il vaut 0, cela veut dire qu'il n'y a aucune équipe qui a été envoyée. Nous allons donc afficher un message correspondant à l'utilisateur et retourner dans le menu principal. Sinon, l'utilisateur peut mettre l'une des unités choisies en réapprovisionnement. Cela veut dire que l'alerte a été traité et que l'équipe est disponible de nouveau. Sauf dans le cas où c'est leur troisième intervention. Dans ce cas, elle va être mise automatiquement en repos grâce à un compteur qui calcule combien de fois chaque unité

a été déployée. Pour mettre une équipe en réapprovisionnement, l'utilisateur saisi un code d'unité parmis ceux qui sont affichés à l'écran. Si l'utilisateur se trompe avec le code, un message correspondant s'affichera, sinon l'unité choisi sera mis en réapprovisionnement. Pour comparer les unités déployées et le code saisi par l'utilisateur, nous utilisons une autre boucle `while` qui parcourt le tableau des Unites tant que le code saisi par l'utilisateur est différent de ceux trouvés et tant que le compteur est inférieure à la taille du tableau de la structure. Si à la fin de la boucle `while` le compteur est égal à la taille du tableau, cela veut dire que l'unité n'a pas été trouvée, sinon, nous traitons les cases nécessaires du tableau en utilisant le compteur comme la position de la case et nous mettons l'équipe en réapprovisionnement.

La description schematique :



ÉTAT D'AVANCEMENT

Nous avons respecté le cahier des charges et notre avancement n'était pas toujours si rapide mais, malgré tout, nous avons réussi à faire la plupart de fonctionnalités demandées.

Les fonctionnalités du logiciel que nous avons réussi à faire :

- ajout d'une nouvelle alerte et unité
- suppression d'une alerte et unité
- recherche des alertes et unités
- modification des alertes et unités
- affectation d'une unité à une alerte
- affichage des unités et des alertes

Les fonctionnalités du logiciel que nous n'avons pas pu faire :

- suppression d'une alerte après avoir mit en réapprovisionnement une unité et vice-versa

CONCLUSION

Nous avons fait le développement du logiciel de gestion qui est assez compliqué au niveau de la conception. Nous avons travaillé à partir d'un cahier des charges qui contenait toutes les instructions que nous avons respecté. Pendant le développement du logiciel de la gestion des secours, nous avons significativement amélioré notre connaissance dans le domaine de la programmation ainsi que la technique et l'approche de la programmation. Nous avons acquéri l'expérience du travail en équipe ce qui est important car dans le domaine de l'informatique, nous sommes souvent amenés à travailler en groupes. Nous avons également approfondi nos connaissances au niveau des pointeurs, fonctions, structures et de la programmation défensive. La gestion de la mémoire dynamique a été l'un des fondements les plus importants du projet. Nous avons donc acquis de nouvelles compétences au niveau de la gestion de la mémoire et même au niveau de la conception des logiciels en langage C. Ce projet nous a permis de progresser et de grandir au niveau de l'expérience et des connaissances dans notre domaine professionnel.