

FISE 1: Algorithmes et structures de données

Une introduction ludique aux structures de données et aux langages C/C++

Christophe Gravier

Table des matières

1	Introduction	3
1.1	Préambule	3
1.2	Objectifs applicatifs	3
1.3	Contenu des séances	3
1.4	Note importante	4
2	Séance 1 : des pokemons	5
2.1	Menu interactif	5
2.2	Structures de base du programme	5
2.3	Fonctions evolve et powerup	6
2.3.1	Evolve	6
2.3.2	Powerup	6
2.4	Affichage d'un pokemon	7
2.5	Tests	7
3	Séance 2 : Bestiaire	8
3.1	Représentation du bestiaire	8
3.1.1	Structure <code>struct EspecePokemon</code>	8
3.1.2	Tableau de 150 <code>struct EspecePokemon</code>	8
3.2	Affichage du bestiaire	9
3.3	Tri du bestiaire	9
3.4	Type de pokemon	9
4	Séance 3 : Mon Pokedex 1/2	11
4.1	Gestion des ressources	11
4.2	Mon Pokedex	11
5	Séance 4 : Mon Pokedex 2/2	14
5.1	Capture d'un pokemon	14
5.2	Gain de ressources	17
5.3	Faire progresser son pokemon	18
6	Séance 5 : Journalisation des évolutions	20
7	Séance 6 : Baston !	22
7.1	Objectifs en terme de fonctionnalités	22
7.2	Structure <code>fp CombatEspece</code>	22
7.3	Table de hachage	23
7.3.1	Taille de la table de hachage	23
7.3.2	Structure pour la table de hachage	23
7.3.3	Fonction de hachage	23
7.3.4	Primitives pour la table de hachage	25
7.3.5	Initialisation des données	25
7.4	Ajout du combat au menu du jeu	25
8	Séance 7 : Récapitulons, débogons, consolidons	27
A	Sortie standard du programme de test du pokedex	28


1 Introduction

1.1 Préambule

Durant nos séances de travaux dirigés, nous allons réaliser de bout en bout un jeu de Pokémon. Alors que j'ai souhaité trouver une application « fil rouge », susceptible de vous intéresser, mais si comme moi vous n'êtes pas un maître ès pokemon, voici un lien pour découvrir l'univers pokemon :

<http://www.pokemon.com/us/parents-guide/>

Nous nous inspirerons particulièrement du fonctionnement du jeu Pokemon Go.



Informations sur l'univers des pokemon

Lorsque nécessaire, des boîtes de dialogue comme celle-ci viendront apporter des informations complémentaires au fonctionnement du jeu et de l'univers des pokemon en général à destination du néophyte. Je ne suis pas expert et quelques raccourcis ont été pris par contrainte d'implémentation. N'hésitez cependant pas à me pointer les incohérences.

1.2 Objectifs applicatifs

Nous souhaitons réaliser notre propre jeu dans l'univers Pokemon en ligne de commandes. Ce jeu se présentera sous la forme d'un menu interactif comme suit (en gras le menu affiché et en police standard une explication associée) :

1. **Index des pokemons** : bestiaire du jeu.
2. **Mon Pokedex** : mes pokemons attrapés, mon nombre de poussières d'étoiles (stardust) et mes bonbons (candies). Ce sont les informations relatives au joueur.
3. **Attraper un pokemon** : attraper un pokemon (nous verrons plus tard comment attraper un pokemon dans notre jeu).
4. **Power-up et évolution** : Dépenser ses poussières d'étoiles (stardust) et bonbons (candies) afin de monter en niveau (power-up) ou pour faire évoluer un de ses pokemons (évolution).
5. **Combat** : Réaliser un combat entre deux de ses pokemons afin de tester leur force.
6. **Quitter** : Terminer le jeu et fermer l'application.

1.3 Contenu des séances

L'aspect ludique de l'application réalisée ne doit pas vous faire perdre de vue les différents objectifs pédagogiques que nous nous fixons dans ce module. Cela inclut la découverte de la syntaxe du C, le maniement des pointeurs, l'environnement de développement, et l'implémentation des structures de données.

Sur ce dernier point, sachez que chaque séance gravite autour de l'implémentation d'une structure de données que nous aurons abordée en cours :

- séance 1 : type `struct` et déclaration de fonctions.
- séance 2 : tableaux statiques,
- séance 3 : tableaux dynamiques,
- séance 4 : tableaux dynamiques suite,
- séance 5 : listes chaînées,
- séance 6 : tables de hachage,
- séance 7 : séance de consolidation.

1.4 Note importante

Une erreur peut survenir à la compilation. C'est normal, personne ne code parfaitement du premier coup ! Il est cependant anormal de poser une question sans avoir lu le message d'erreur et avoir tenté de le comprendre.

Une erreur peut survenir à l'exécution. Il est cependant anormal, dès lors que vous avez compris le fonctionnement du débogueur lors des séances de BCD, de ne pas utiliser le débogueur avant de poser une question.

Nous ne répondrons donc pas aux questions des élèves n'ayant pas pris le soin de lire les messages d'erreur et n'ayant pas pris le soin de lancer le débogueur. Il ne s'agit pas là d'une volonté de vous laisser « pa-ta-ger », bien au contraire. Manier le débogueur et décrypter les messages d'erreur fait partie des objectifs pédagogiques. Si vous faites toujours appel à voz « débogueurs sur pattes » que peuvent être vos enseignants de TDs, vous ne serez jamais autonome en programmation. Programmer s'apprend en tapant mais aussi en réfléchissant !

Cela étant dit, si vous ne comprenez pas les messages d'erreur ou que le débogueur ne vous aide pas, ne restez pas bloqué non plus ...

2 Séance 1 : des pokemons


2.1 Menu interactif

Dans un premier temps, nous allons réaliser notre menu interactif, même si chaque entrée du menu ne sera pas encore réalisée. Pour cela vous utiliserez les instructions d'entrée/sortie au clavier et écran. Nous souhaitons :

1. Afficher le menu tel que présenté en introduction (texte en gras).
2. Contrôler l'entrée utilisateur :
 - Si l'utilisateur a entré un numéro valide de menu, afficher le label du menu associé pour le moment (chaque séance vise ensuite à implémenter les fonctionnalités derrière chaque menu).
 - Si l'utilisateur a entré un numéro invalide, affichage d'un message d'erreur.
3. Finalement, affichage à nouveau du menu, en l'attente d'une nouvelle saisie utilisateur.

Il vous est conseillé de considérer la création d'une fonction *ad hoc* pour afficher le menu afin de l'appeler autant que nécessaire. Afin de réaliser cette fonction, vous pouvez utiliser l'instruction **switch** qui vous sera présentée.

2.2 Structures de base du programme



Attrapez-les tous!

Pokemon est un terme porte-manteau pour "Poket Monster". Nous nous basons sur un bestiaire dans ce cours des 150 premiers pokemon de Pokemon Go :

https://www.pokebip.com/page__jeuxvideo__pokemon_go__pokemon.html.

Chaque pokemon possède des caractéristiques propres (points de vie, aptitude au combat, expérience, ...). Un pokemon peut aussi évoluer (se transformer) une à deux fois – le rendant plus fort, résistant et expérimenté.

Voici les types de pokemon que nous prendrons en compte : Normal, Fighting, Flying, Poison, Ground, Rock, Bug, Ghost, Steel, Fire, Grass, Water, Electric, Psychic, Ice, Dragon, Dark, Fairy.

Nous allons créer les deux premiers types de données :

1. `PokemonType` : un type énuméré, représentant une espèce de pokemon (comme par exemple les pokemons de type Bulbizarre). On décrit ici les caractéristiques de cette famille de pokemon, leur différentes évolutions, etc.
2. `Pokemon` : représentant un pokemon existant dans notre monde virtuel (comme par exemple un pokemon Bulbizarre, ayant ses points de vie, d'expérience, etc.).

La **struct** `PokemonType` sera créée dans un fichier appelé **`pokemonTypes.h`**.

La **struct** `Pokemon` sera placée dans un fichier appelé **`pokemon.h`**.

Le premier type – énuméré –, `PokemonType`, vous est donné au Listing 1.

La seconde structure `Pokemon` est à écrire, sachant que l'on a besoin de représenter pour chaque pokemon :

- Son niveau d'expérience – que l'on peut appeler **XP** – (un entier),
- Son numéro d'**évolution** (1, 2, 3, ...) – nous gérerons les noms de chaque type d'évolution pour chaque pokemon dans d'autres séances,
- Ses points de vie actuels – que l'on peut appeler **PV** – (un entier),

- Ses points de combat actuels – que l’on peut appeler **CP** – (un entier).

Listing 1: Type énuméré *PokemonType*

```
// le type énuméré décrivant le type de Pokemon parmi les 18 existants

typedef enum {
    Normal, Fighting, Flying, Poison, Ground, Rock, Bug,
5    Ghost, Steel, Fire, Grass, Water, Electric, Psychic,
    Ice, Dragon, Dark, Fairy
} PokemonType;
```

2.3 Fonctions evolve et powerup

Evolution et Powerup d'un pokemon

- **Power-up** : utiliser ses ressources (**bonbons** et **poussières** afin d'augmenter les capacités de combat d'un pokemon. Cela se traduit par l'augmentation des points de combat (CP) du pokemon.
- **Evolution** : utiliser ses **bonbons** afin de transformer son pokemon dans une version plus évoluée
 - le pokemon change de nom, d'apparence et ses caractéristiques sont augmentées, notamment points d'expérience et points de vie.

Nous souhaitons ajouter à notre programme la capacité de faire évoluer (*evolve*) ou progresser (*powerup*) un pokemon donné.

2.3.1 Evolve

Soit une fonction appelée *evolve*, prenant un *Pokemon* en paramètre et ne retournant rien. Cette fonction fait évoluer un pokemon (on incrémente son champ *evolution* de un). Son prototype est le suivant :

```
void evolve(Pokemon& p);
```

Le pokemon concerné est passé sous la forme d'une référence (spécifique au langage C++). Il s'agit d'une donnée dite d'entrée-sortie : les modifications faites sur ce pokemon *p* seront visibles sur la variable passée en paramètre à l'issue de l'appel à la fonction. Nous avons besoin de cette fonction mais nous reviendrons sur ce signe bizarre '&' (esperluette – ou encore « et » commercial) devant le nom de l'argument dans une prochaine séance de cours.

2.3.2 Powerup

La fonction *powerup* comporte le même argument et dispose du même type de retour que la fonction précédente. Elle permet de faire progresser un pokemon en terme d'aptitude au combat. Nous considérerons que faire progresser (*to powerup*) un pokemon, revient à faire progresser de 25% ses points de combat (Combat Points – CP).

Il vous appartient d'écrire le prototype et la définition de cette fonction.

Vous penserez à écrire la signature de chaque fonction dans le fichier header (*pokemon.h*), et la définition de ces fonctions dans le fichier d'implémentation (*pokemon.cpp*) – ce dernier fichier est à créer, il faut y inclure le fichier *pokemon.h*.

2.4 Affichage d'un pokemon

Écrire une fonction `void display(const Pokemon p);` permettant d'afficher les informations d'un pokemon sur la sortie standard. Penser à inclure la signature de la fonction dans le fichier header et sa définition dans le fichier `pokemon.cpp`.

2.5 Tests

Implémenter l'utilisation de ces fonctions et structures : créer une variable de type `pokemontype`, initialiser ses champs, créer une variable de type `Pokemon`, initialisez ses champs, afficher un pokemon, utiliser les fonctions `powerup` et `evolve` avant d'afficher à nouveau le pokemon et vérifier que ses champs sont bien mis à jour.

3 Séance 2 : Bestiaire

Dans cette section, nous souhaitons constituer le bestiaire de notre univers Pokemon. Nous utiliserons les 150 premiers pokemon dans Pokemon Go. La liste complète et les besoins en évolution est accessible à la page : <http://www.listchallenges.com/list-of-pokemon-in-pokemon-go> ou encore <https://recombu.com/mobile/article/pokedex-for-pokemon-go-complete-list-of-wild-pokemon-evolved-guide>.

Nous représentons le bestiaire (l'ensemble des types de pokemons attrapables dans le jeu), sous la forme d'un tableau de taille fixe (150), contenant pour chaque case une structure de données, `EspecPokemon`, décrivant :

- Le nom de l'espèce de pokemon.
- Le type de pokemon (en utilisant le type énuméré `PokemonType` réalisé à la séance 1).
- Le nom du pokemon vers lequel cette espèce peut évoluer. Ce champs est vide (`NULL`) sinon.
- Le nombre de bonbons nécessaires pour permettre son évolution. Ce champs vaut 0 lorsque ce pokemon ne peut plus évoluer.

Dans cette séance nous nous intéressons donc à développer la fonctionnalité derrière le menu 1. Nous allons successivement réaliser :

1. La représentation du bestiaire en C.
2. L'affichage du bestiaire sur la sortie standard.
3. Le tri du bestiaire.

3.1 Représentation du bestiaire

3.1.1 Structure `struct EspecPokemon`

Dans un premier temps, vous réaliserez la déclaration de la structure `struct EspecPokemon` telle que décrite précédemment (nom de l'espèce, nom de l'espèce vers laquelle cette espèce peut évoluer, et nombre de bonbons nécessaires à l'évolution).

Cette structure sera décrite dans le fichier `typesPokemon.h` (celui contenant déjà le type énuméré).

3.1.2 Tableau de 150 `struct EspecPokemon`

Nous appellerons dès le début du programme `main` une fonction dont le prototype est le suivant :

```
void initBestiaire(EspecPokemon bestiaire[]);
```

Elle prend en argument un tableau vide de 150 cases permettant de stocker des `EspecPokemon` et le remplit par les 150 espèces de pokemon disponibles dans le jeu. Son prototype sera déclaré dans le fichier `bestiaire.h` et son implémentation dans `bestiaire.cpp`.

Dans l'implémentation de la fonction, nous saisisons « à la main » toutes les infos pour les 150 pokemons. Vous pourrez vous rendre compte que cela est une tâche fastidieuse mais je l'ai faite pour vous par avance : <https://gist.github.com/anonymous/8838a11c2330938d50d16634daf85da6>.

3.2 Affichage du bestiaire

Nous allons tout d'abord modifier le menu 1 – Index des pokemons (pour le moment la sélection de ce menu renvoie « unimplemented yet ».). Dans ce menu, le programme affichera la liste des pokemons telle que représentée par le tableau `bestiaire`. Le style d'affichage est libre. Il est pertinent de créer une nouvelle fonction dans les fichiers relatifs au bestiaire (`bestiaire.h` et `bestiaire.cpp`), et d'appeler celle-ci au bon moment dans la gestion du menu.

3.3 Tri du bestiaire

Les pokemons s'affichent par ordre d'entrée dans le bestiaire. Nous souhaitons ici mettre en place la possibilité de trier le bestiaire suivant le nom des pokemons.

Pour cela, nous ajouterons une fonction de tri dans les fichiers `bestiaire.h` et `bestiaire.cpp` pour implémenter ce tri.

L'algorithme de tri choisi sera le tri par sélection que nous avons déjà réalisé en pseudo-code lors des séances de BCD algorithmique. Pour rappel, cet algorithme trie un tableau `T` de taille n avec la stratégie suivante :

1. Initialise un entier `i` à 0
2. Tant que `i` est inférieur ou égale à $n - 1$:
 - (a) Parcours de `T[i]` à `T[n-1]` à la recherche de la plus petite valeur,
 - (b) Un fois trouvée (soit l'indice k de cette plus petite valeur), permuter `T[i]` et `T[k]`,
 - (c) Incrémenter `i` de 1.

Il vous faudra comparer des chaînes de caractères. Pour cela, vous utiliserez la fonction du module `string` (à inclure sous la forme `#include <string>`) permettant de comparer des chaînes :

```
int strcmp(const char *str1, const char *str2)
```

Cette fonction renvoie :

- une valeur négative si `str1` est **avant** `str2` dans l'ordre alphabétique.
- une valeur positive si `str1` est **après** `str2` dans l'ordre alphabétique.
- la valeur 0 si les chaînes sont identiques.

Une fois cette fonction réalisée, vous la testerez en intégrant un choix dans le menu de tri : affichage par ordre naturel – c'est-à-dire tel que saisi manuellement dans `bestiaire.cpp` – ou affichage trié par nom).

3.4 Type de pokemon

Pour le moment, le bestiaire contient les pokemons, et pour chacun, entre autres, le type de pokemon (Fire, Grass, etc.) parmi les 18 disponibles. En parallèle, la fonction `void display(const Pokemon p);` affiche les informations sur un pokemon, mais pas encore son type.

Dans cette partie, nous allons ajouter l'affichage du type de pokemon lors de l'affichage d'un pokemon. Pour cela, il nous faut, pour un pokemon à afficher, aller chercher l'espèce correspondante dans le bestiaire (i.e. trouver la case du tableau dont le champs `nom` a la même valeur que le nom du pokemon). Une fois l'espèce trouvée (une variable de type `EspecPokemon`), le champs énuméré `PokemonType` nous renseigne.

Cependant, un type énuméré en C est en fait codé comme un entier (le premier label vaut 0, le second label vaut 1, etc.). Si nous affichons directement la valeur `EspecPokemon.Normal`, nous n'aurons pas la chaîne

de caractère « Normal » à la sortie standard, mais l'entier 0. Nous allons donc mettre en place un nouveau tableau statique établissant cette correspondance entre l'entier dénotant l'espèce et la chaîne de caractères associée.

Pour résumer et vous guider, afin d'ajouter l'affichage correct du type de chaque pokemon lorsque nous affichons un pokemon à l'aide de la fonction `void display(const Pokemon p);`, nous allons :

1. Ajouter un tableau statique de 18 chaînes de caractères (`char*`) que nous nommerons `typesLabel`. Ce tableau est renseigné manuellement et contenant à l'indice 0 la chaîne « Normal », à l'indice 1 la chaîne « Fighting », etc. Ce tableau est créé au tout début de la fonction `main`, au même moment où l'on initialise le bestiaire. Cette variable peut alors être passée aux fonctions appelées plus tard.
2. Modifier la fonction `void display(const Pokemon p);` afin d'aller rechercher dans le bestiaire l'entier correspondant à l'espèce de ce pokemon (le champs `EspecePokemon.type`). Le bestiaire devra être passé en paramètre de la fonction. Il est probable que nous ayons plusieurs fois à chercher des pokemons dans le bestiaire (ajouter une fonction dans le bestiaire plutôt que mettre le code directement dans la fonction `void display(const Pokemon p);` pourrait être astucieux par avance).
3. Retrouver dans le tableau `typesLabel` la chaîne associée afin de l'afficher.

Vous devez également tester ces fonctions dans le programme principal, avant l'appel au menu (créer un pokemon et essayer de l'afficher avec la nouvelle fonction – son type doit être maintenant affiché sous forme de chaîne de caractères).

4 Séance 3 : Mon Pokedex 1/2

Dans cette séance nous nous intéresserons aux second et troisième menus : la gestion du pokedex et la capture de pokemon. Pour cela, nous allons maintenir le nombre de ressources qu'un utilisateur obtient lors de la capture d'un pokemon, la capture en elle-même d'un pokemon tiré aléatoirement (comme ces caractéristiques), ainsi que l'utilisation de ressources (poussières et bonbons) afin de faire progresser ses pokemons.

La séance est principalement axée autour de la notion de tableaux dynamiques – un tel tableau sera utilisé afin de gérer l'ensemble des pokemons attrapés par un utilisateur.

Informations sur la capture d'un pokemon

Dans notre jeu, lorsqu'un utilisateur capture un pokemon, il obtient :

- 100 poussières d'étoiles (*stardust* dans le jeu original),
- 3 bonbons (*candies*).

4.1 Gestion des ressources

Afin de gérer les ressources de l'utilisateur, nous allons créer une nouvelle structure (`struct Ressource`). Cette structure comprendra :

- un entier `poussieres`, représentant le nombre de poussières dont dispose l'utilisateur,
- un entier représentant le nombre de bonbons dont dispose l'utilisateur.

Vous réaliserez cette structure dans une nouvelle paire de fichiers `.h` et `.cpp` nommés `Pokedex`.

4.2 Mon Pokedex

Avant de capturer des pokemons, il nous faut disposer d'une structure de données permettant de stocker notre collection de pokemons. Comme le nombre de pokemons dans la collection n'est pas connu par avance et qu'il ne cesse de croître, nous utiliserons un tableau dynamique de pokemons pour cela.

Nous gérerons ce tableau avec les contraintes suivantes :

- Nous souhaitons stocker nos pokemons en les triant par ordre croissant de noms. Nous allons réaliser un tri par insertion : à chaque nouvelle insertion d'un pokemon dans ce tableau dynamique, le pokemon sera inséré à sa place, au regard de son nom.
- Si le tableau ne permet plus d'entrer de nouveau pokemon car il a atteint sa capacité maximale, nous devons faire croître le tableau dynamique de 50% de sa capacité courante.

Travail à réaliser :

1. Réaliser une fonction `Pokemon* initPokedex(const int size);`, permettant d'initialiser un tableau dynamique de pokemons sur le tas de de taille `size` et renvoie le pointeur vers ce nouveau tableau.
2. Réaliser une fonction `void insertPokemon(Pokedex* monPokedex, Pokemon p);` permettant d'insérer un pokemon `p` dans la pokedex.
Il conviendra de gérer le cas où `monPokedex->nbPokemons == monPokedex->capacity`, en faisant l'allocation d'un nouveau tableau de `Pokemon` sur le tas 50% plus grand – cela revient à modifier le pointeur `monPokedex->mesPokemons`.

En plus de ces deux fonctions, nous aurons également besoin d'une fonction d'affichage du contenu du pokédex. Le listing 2 vous donne une possible implémentation. Cette fonction reprend les variables `bestiaire` et `typeLabel` en paramètres : cela est utile car la fonction réutilise celle permettant d'afficher un pokemon (ces paramètres sont nécessaires pour trouver le type de pokemon et le label associé pour l'affichage, c.f. séance précédente).

Listing 2: *Fonction d'affichage du contenu d'un pokédex*

```
void displayPokedex(Pokedex* pokedex, EspecePokemon bestiaire[150],
                   char* typesLabel[18])
{
5   cout << endl << "Contenu du pokédex (Actuellement ";
   cout << pokedex->nbPokemons << " pokemons capturés)" << endl;

   for (int i = 0; i < pokedex->nbPokemons; i++) {
       display(pokedex->mesPokemons[i], bestiaire, typesLabel);
10  }
}
```

Tester vos deux fonctions dans un programme principal à part pour le moment (réaliser une fonction `main` dans un fichier `main2.cpp` par exemple).

Le listing 3 donne le contenu d'un tel programme `main`.

L'annexe A présente ce que l'on escompte à la sortie standard (remarquez que le nombre de pokemons croît et qu'ils sont rangés au fur et à mesure par ordre croissant de nom).

Listing 3: *Programme principal de test du pokédex*

```
EspecePokemon bestiaire[150];
initBestiaire(bestiaire);

char* typesLabel[18] = {"Normal", "Fighting", "Flying", "Poison", "Ground",
5  "Rock", "Bug", "Ghost", "Steel", "Fire", "Grass", "Water", "Electric",
  "Psychic", "Ice", "Dragon", "Dark", "Fairy"};

Pokedex* pokedex = initPokedex(3);
Pokemon a;
10 a.nom = "Seadra";
   a.xp = 1;
   a.cp = 4;
   a.evolution = 1;
   a.pv = 23;
15
   Pokemon b;
   b.nom = "Arbok";
   b.xp = 0;
   b.cp = 123;
20 b.evolution = 2;
   b.pv = 134;

   Pokemon c;
```

```
25     c.nom = "Zubat";
    c.xp = 40;
    c.cp = 300;
    c.evolution = 1;
    c.pv = 223;

30     Pokemon d;
    d.nom = "Paras";
    d.xp = 0;
    d.cp = 23;
    d.evolution = 1;
35     d.pv = 90;

    Pokemon e;
    e.nom = "Arkanine";
    e.xp = 0;
40     e.cp = 23;
    e.evolution = 2;
    e.pv = 290;

    insertPokemon(&pokedex, a);
45     cout << "***** Pokedex after inserting pokemon a *****" << endl;
    displayPokedex(pokedex, bestiaire, typesLabel);
    cout << "*****" << endl;
    cout << endl << endl;

50     insertPokemon(&pokedex, b);
    cout << "***** Pokedex after inserting pokemon b *****" << endl;
    displayPokedex(pokedex, bestiaire, typesLabel);
    cout << "*****" << endl;
    cout << endl << endl;

55     insertPokemon(&pokedex, c);
    cout << "***** Pokedex after inserting pokemon c *****" << endl;
    displayPokedex(pokedex, bestiaire, typesLabel);
    cout << "*****" << endl;
60     cout << endl << endl;

    insertPokemon(&pokedex, d);
    cout << "***** Pokedex after inserting pokemon d *****" << endl;
    displayPokedex(pokedex, bestiaire, typesLabel);
65     cout << "*****" << endl;
    cout << endl << endl;

    insertPokemon(&pokedex, e);
    cout << "***** Pokedex after inserting pokemon e *****" << endl;
70     displayPokedex(pokedex, bestiaire, typesLabel);
    cout << "*****" << endl;
    cout << endl << endl;
```

5 Séance 4 : Mon Pokedex 2/2

Suite de la séance précédente avec de nouvelles fonctionnalités, basées sur la manipulation de tableaux dynamiques.

5.1 Capture d'un pokemon

Lorsque l'on entre dans le menu 3, nous tentons de capturer un pokemon. À chaque tentative de capture d'un pokemon, un nouveau pokemon est aléatoirement généré. Ce pokemon sauvage est alors affiché à l'écran et nous tentons de l'attraper par un jet de dés (virtuel) : si nous générons une valeur comprise entre 0 et 100 qui est supérieure à 60, le pokemon est capturé. Sinon le pokemon s'enfuit et nous ne pourrions plus l'attraper (la nouvelle tentative suivante exige la création d'un nouveau pokemon aléatoire). Nous avons donc 40% de chance de capture.

Des pokemons sauvages

Des pokemons sauvages sont des pokemons qui n'appartiennent à personne, et que nous essayons donc de capturer pour augmenter notre collection. Evidemment, tous les pokemons ne se valent pas : un Bulbizarre est moins puissant qu'un Mew. Mais deux bulbizarres ne se valent pas non plus : un bulbizarre A nait potentiellement avec plus ou moins de points de combat et de points de vie. Concernant les points de combat, la page suivante fournit des informations sur les valeurs minimum et maximum qu'un pokemon peut naturellement avoir : <http://www.serebii.net/pokemongo/combatpoints.shtml>. De plus, nous ne pouvons d'ordinaire capturer que des pokemon non évolués (i.e. des pokemon d'évolution 1). Enfin, certains pokemons sont plus rares !

Il est évident que cela complexifie notre jeu, et nous souhaitons réaliser une première version relativement simple à implémenter dans le temps qui nous est imparti. Ainsi, dans notre « Pokemon TSE Edition », nous fixons les conditions suivantes quant aux caractéristiques d'un pokemon sauvage :

- **Seuls les pokemons de première évolution peuvent apparaître.**
- **Les pokemons de première évolution ont tous la même probabilité d'apparition - il n'y en a pas de plus rares que d'autres.**
- **Tous les pokemons sont aléatoirement créés. Un pokemon d'évolution 1, quel qu'il soit, aura un nombre aléatoire de points de combat tiré dans l'intervalle [200,1000] ainsi qu'un nombre de points de vie (PV) tiré uniformément dans l'intervalle [0; 500].**

Si vous êtes passionné(e)s et désireux(se) de coller le plus possible à l'esprit du jeu, libre à vous d'imposer des contraintes supplémentaires comme par exemple un tirage aléatoire en fonction de la rareté des pokemons, ou encore un interval de points de vie ou de combat suivant le type de pokemon. Cela pose un challenge intéressant aux étudiant(e)s ayant déjà réalisé la version attendue et désireux d'acquérir une maîtrise plus grande du langage C à travers ce sujet ludique. Sentez-vous libre de le faire le cas échéant. La solution ne comprendra pas ces aspects avancés afin de rester accessible au plus grand nombre.

Afin de fournir les sous-fonctions nécessaires à la réalisation de ce menu, le travail à réaliser est le suivant :

1. Modifier la structure `EspecPokemon` afin d'ajouter un champs boolean appele `estEvolue`. Si ce champs est à VRAI (`true`), cela signifie que le pokemon n'est pas de première évolution (c'est une évolution d'un autre pokemon du bestiaire). Une fois cette petite modification réalisée, il convient de modifier la création du bestiaire dans `bestiaire.cpp` afin de préciser la valeur de ce champs pour tous les pokemons.

2. Créer une fonction nommée `genererPokemon` dont le prototype est le suivant :

```
Pokemon genererPokemon(EspecePokemon bestiaire[150], char* typesLabel[18])
```

dans le fichier Pokedex (fichier d'entête et fichier `.cpp`). Cette fonction a pour algorithme le suivant :

- Tirage aléatoire d'un pokemon dans le bestiaire (une `EspecPokemon`) tant que l'espèce tirée n'est pas de première évolution. À l'issu de cette itération, l'espèce tirée détermine l'espèce du pokemon sauvage qui va être soumis à l'utilisateur.
- Un pokemon est initialisé : son nom est celui de l'espèce de pokemon tirée aléatoirement précédemment, ses points d'expérience sont à zéro, ses points de combat tirés uniformément dans l'intervall `[200;1000]` et ses points de vie tirés uniformément aléatoirement dans l'intervall `[30;500]`.
- L'utilisateur voit le pokemon affiché avec ses caractéristiques. On lui demande s'il veut tenter de i) capturer le pokemon, ii) essayer d'en capturer un autre ou ii) revenir au menu principal.
- S'il choisit d'en tirer un autre, retour à l'item 1 (nouveau pokemon sauvage à générer).
- S'il choisit de capturer ce pokemon, il a 40% de chance de réussir à partir d'un tirage aléatoire (par exemple dans l'intervall `[0;100]`). En cas de succès, le pokemon est ajouté à son pokedex (et l'on peut vérifier cela en accédant au pokedex via l'écran principal!).

Listing 4: Ajout du booléen permettant de savoir si un pokemon est de première évolution ou non

```
// rajout des infos d'évolution
for (int i = 0; i < 150; i++) {
    bestiaire[i].estEvolue = false;
}

// set all evolved pokemon to true
bestiaire[0].estEvolue = 0;
bestiaire[3].estEvolue = 0;
bestiaire[6].estEvolue = 0;
bestiaire[9].estEvolue = 0;
bestiaire[12].estEvolue = 0;
bestiaire[15].estEvolue = 0;
bestiaire[18].estEvolue = 0;
bestiaire[20].estEvolue = 0;
bestiaire[22].estEvolue = 0;
bestiaire[24].estEvolue = 0;
bestiaire[26].estEvolue = 0;
bestiaire[28].estEvolue = 0;
bestiaire[31].estEvolue = 0;
bestiaire[34].estEvolue = 0;
bestiaire[36].estEvolue = 0;
bestiaire[38].estEvolue = 0;
bestiaire[40].estEvolue = 0;
bestiaire[42].estEvolue = 0;
bestiaire[45].estEvolue = 0;
bestiaire[47].estEvolue = 0;
bestiaire[49].estEvolue = 0;
bestiaire[51].estEvolue = 0;
bestiaire[53].estEvolue = 0;
bestiaire[55].estEvolue = 0;
bestiaire[57].estEvolue = 0;
bestiaire[59].estEvolue = 0;
bestiaire[62].estEvolue = 0;
bestiaire[65].estEvolue = 0;
```

```
35 | bestiaire[68].estEvolue = 0;
    | bestiaire[71].estEvolue = 0;
    | bestiaire[73].estEvolue = 0;
    | bestiaire[76].estEvolue = 0;
    | bestiaire[78].estEvolue = 0;
40 | bestiaire[80].estEvolue = 0;
    | bestiaire[82].estEvolue = 0;
    | bestiaire[83].estEvolue = 0;
    | bestiaire[107].estEvolue = 0;
    | bestiaire[108].estEvolue = 0;
45 | bestiaire[110].estEvolue = 0;
    | bestiaire[112].estEvolue = 0;
    | bestiaire[113].estEvolue = 0;
    | bestiaire[114].estEvolue = 0;
    | bestiaire[115].estEvolue = 0;
50 | bestiaire[117].estEvolue = 0;
    | bestiaire[119].estEvolue = 0;
    | bestiaire[121].estEvolue = 0;
    | bestiaire[122].estEvolue = 0;
    | bestiaire[123].estEvolue = 0;
55 | bestiaire[124].estEvolue = 0;
    | bestiaire[125].estEvolue = 0;
    | bestiaire[126].estEvolue = 0;
    | bestiaire[127].estEvolue = 0;
    | bestiaire[128].estEvolue = 0;
60 | bestiaire[130].estEvolue = 0;
    | bestiaire[131].estEvolue = 0;
    | bestiaire[132].estEvolue = 0;
    | bestiaire[136].estEvolue = 0;
    | bestiaire[137].estEvolue = 0;
65 | bestiaire[139].estEvolue = 0;
    | bestiaire[141].estEvolue = 0;
    | bestiaire[142].estEvolue = 0;
    | bestiaire[143].estEvolue = 0;
    | bestiaire[144].estEvolue = 0;
70 | bestiaire[145].estEvolue = 0;
    | bestiaire[146].estEvolue = 0;
    | bestiaire[149].estEvolue = 0;
```

Voici quelques aides afin de réaliser ce travail.

Tout d’abord, ajouter si un pokemon est de première évolution ou non pour tout pokemon du bestiaire est une opération fastidieuse. Soyez heureux : cela a déjà été réalisé par votre vénérable enseignant qui vous offre ce code au Listing 4. À ajouter au bon endroit !

Aussi, sachez que générer un nombre (pseudo-)aléatoire tiré uniformément dans un interval n’est pas si trivial. Sans rentrer dans les détails dans ce cours, nous nous inspirerons du post fourni sur le site <http://stackoverflow.com/questions/2509679/how-to-generate-a-random-number-from-within-a-range> nous fournissant une fonction toute prête pour réaliser ce travail. Cette fonction est fournie au Listing 5. Cette fonction pourra être définie dans un couple de fichiers .h et .cpp appelés utils (nous regrouperons dans ces fichiers des fonctions « utilitaires »).

Listing 5: Génération d'un nombre aléatoire compris entre 0 et max

```
#include <stdlib.h> // For random(), RAND_MAX

// Assumes 0 <= max <= RAND_MAX
// Returns in the closed interval [0, max]
5 long random_at_most(long max) {
    unsigned long
        // max <= RAND_MAX < ULONG_MAX, so this is okay.
        num_bins = (unsigned long) max + 1,
        num_rand = (unsigned long) RAND_MAX + 1,
10    bin_size = num_rand / num_bins,
        defect = num_rand % num_bins;

    long x;
    do {
15        x = random();
    }
    // This is carefully written not to overflow
    while (num_rand - defect <= (unsigned long)x);

20    // Truncated division is intentional
    return x/bin_size;
}
```

5.2 Gain de ressources

Nous avons déjà défini que nous disposions de bonbons et de poussière d'étoiles comme ressource dans le jeu. Les bonbons et les poussières sont acquises lors de la capture d'un pokemon. Les poussières sont des ressources génériques, mais les bonbons sont propres à chaque pokemon dans la version original du jeu ! Nous relâcherons cette règle du jeu en considérant dans notre version qu'il n'existe qu'une seule sorte de bonbons, commune à tous les pokemons.

Sachez qu'à chaque capture, nous gagnons dans notre jeu :

- 3 bonbons,
- 100 poussières.

Nous souhaitons ici ajouter cette fonctionnalités au jeu.

Pour cela, nous allons utiliser la structure `Ressources` réalisée à la séance 3. Celle-ci est rappelée au Listing `Ressources`. Une variable de type `Ressources` sera créée au début du programme `main` et utilisée tout au long du programme pour gérer les poussières d'étoiles et les bonbons du joueur.

Le travail à réaliser ici est :

1. Réaliser la mise à jour des ressources collectées au fur et à mesure de capture de pokemon, i.e. mettre à jour les champs de la variable de type `Ressources` à chaque capture.
2. Afficher le nombre de poussières et bonbons disponibles à chaque affichage du pokedex de l'utilisateur. La fonction `displayPokedex` prendra donc un nouvel argument de type `Ressources` afin de compléter l'affichage du pokedex en cours.

Listing 6: *Fichier ressources.h contenant la description de la structure Ressources*

```
#ifndef Ressource_h
#define Ressource_h

typedef struct {
5     int stardust;
    int candies;

} Ressources;
10
#endif /* Ressource_h */
```

5.3 Faire progresser son pokemon

Cette partie du travail concerne le menu 4: « Power-up et évolution ».

Après que l'utilisateur ait choisi entre power-up et évolution dans un sous-menu à créer, nous listons les pokemons de son pokedex (appeler la fonction d'affichage du pokedex). L'utilisateur peut alors choisir le pokemon à faire évoluer ou progresser.

Progression (power-up) Nous considérerons qu'il faut 10 bonbons et 500 poussières pour faire progresser un pokemon. Nous avons déjà réalisé la fonction power-up lors d'une séance précédente!

Evolution De par le bestiaire, nous connaissons pour chaque pokemon le nombre de bonbons pour évoluer (cette quantité de bonbons diffère pour chaque pokemon). Lors d'une évolution, il convient de :

1. Vérifier que l'utilisateur dispose d'assez de ressources et que le pokemon peut encore évoluer (le champs `evolvesTo` dans le bestiaire pour cette espèce de pokemon n'est pas un pointeur `null`).
2. Modifier le pokemon (changer son évolution vers celle prévue dans le bestiaire), incrémenter son numéro d'évolution. L'incrémentation est déjà réalisée dans la fonction `void evolve(Pokemon& p)` du fichier `pokemon.cpp`. Il convient de faire évoluer cette fonction (rajouter le bestiaire en paramètre afin de trouver le pokemon et donc le nom de son évolution suivante).
3. Retrancher les ressources consommées des ressources de l'utilisateur.

Ces implémentations d'algorithmes d'évolution et de progression d'un pokemon sont à implémenter dans le fichier `main.cpp`, à l'endroit pointé au Listing 7. Il vous sera certainement nécessaire de modifier le mode de passage d'un pokemon en paramètre des fonctions `evolve` et `powerup` afin de passer des pokemons par pointeurs.

Listing 7: *Passage à modifier pour ajouter les algorithmes d'évolution et de progression d'un pokemon*

```
(...)  
  
if (newchoice == 1) {  
5     // Insérer ici la progression d'un pokemon  
    cout << "powerup unimplemented yet";  
} else if (newchoice == 2) {  
  
    // Insérer ici l'évolution d'un pokemon  
10    cout << "evolution unimplemented yet";
```

```
}  
  
(...)
```

Pensez-vous qu'écrire une fonction si grande dans le fichier `main.cpp` est une bonne idée? Qu'auriez-vous pu faire autrement?

6 Séance 5 : Journalisation des évolutions

Cette séance s'intéresse à la réalisation et l'implémentation de listes chaînées. Au niveau fonctionnalités, nous souhaitons conserver les évolutions successives que nous avons réalisées avec nos pokemons dans un ordre anté-chronologique (les plus récentes en premier). Nous souhaitons en effet rajouter un menu à notre jeu afin d'afficher l'historique des évolutions réalisées.

Pour cela, nous créerons deux structures, représentant respectivement une liste chaînée (**struct** historique) et les maillons de cette liste chaînée **struct** evolution.

struct Evolution Cette structure représente une évolution ayant eu lieu. Elle est composée de deux champs : la chaîne de caractères représentant le nom du pokémon avant évolution, le nom de l'espèce vers laquelle ce pokemon a évolué, ainsi que le pointeur vers l'évolution suivante, chronologiquement parlant.

struct Historique Cette structure contient le pointeur vers la tête de la liste chaînée, ainsi que le nombre d'évolutions ayant eu lieu.

Prenons l'exemple suivant. Imaginons que nous avons fait les évolutions suivantes dans le jeu au cours du temps (par ordre chronologique de réalisation) :

1. un Charmeleon est devenu un Charizard
2. un Oddish est devenu un Gloom
3. un Charmander est devenu un Charmeleon

Une telle succession d'évolutions peut être représentée par la liste chaînée donnée à la figure 1.

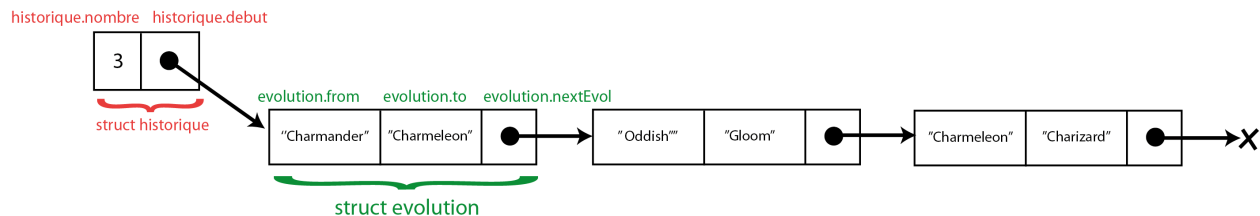


FIG. 1 – Représentation de la liste chaînée de journalisation des évolutions de pokemons.

Le listing 8 donne l'implémentation de la structure **struct** historique.

Listing 8: Implémentation de la structure historique

```
#ifndef historique_h
#define historique_h

typedef struct {
5     int nombre;
    Evolution* debut;
} historique;

#endif /* historique_h */
```

Afin de réaliser la sauvegarde et l'affichage de l'historique des évolutions des pokemons, voici une suite de jalons proposée le long de votre implémentation :

1. Implémenter la structure Evolution.

2. Ajouter une fonction pour insérer une évolution dans la liste chaînée (il s'agit d'une insertion en tête de liste chaînée afin de garder l'ordre antéchronologique).
3. Utiliser cette fonction lors de l'évolution d'un pokemon afin de consigner dans la liste chaînée les évolutions réalisées.
4. Rajouter un menu au jeu afin d'afficher le contenu de cette liste chaînée (il s'agit du parcours d'une liste chaînée).

Une fois ces fonctionnalités ajoutées et testées dans votre jeu, répondez à la question suivante : que faudrait-il modifier pour afficher les évolutions successives non plus par ordre ante-chronologique mais par ordre chronologique ?

7 Séance 6 : Baston !

7.1 Objectifs en terme de fonctionnalités

Durant cette séance, nous nous intéressons au combat de pokemons.

Combats de pokemons

Dans le monde des pokemons, le type de pokemon influe sur ses capacités d'attaque et de défense. Par exemple, un pokemon de type Feu est efficace contre un pokemon de type Herbe. La liste complète des types de pokemons contre lequel un pokemon est fort et faible est, par exemple, accessible à l'adresse : <https://www.primagames.com/games/pokemon-go/tips/best-pokemon-pokemon-go-strategies-and-battle-type-chart>. Tout pokemon est fort ou faible contre certains autres, mais il peut aussi être « neutre » (ni fort, ni faible) contre certains types de pokemons.

Pour cela nous allons créer une structure contenant les types de pokemons contre lesquels un type de pokemon considéré est fort ou faible. Toutes ces informations sont ensuite stockées dans une table de hachage avec un chaînage linéaire que nous réaliserons, et donc la fonction de hachage se basera sur le nom de l'espèce du pokemon. Nous pourrons ensuite réaliser l'implémentation d'un combat de pokemon : nous exploiterons la table de hachage afin de pondérer les dégâts infligés par un pokemon sur un autre. La figure 2 représente un extrait de la table de hachage que nous réaliserons et exploiterons pour l'implémentation des combats.

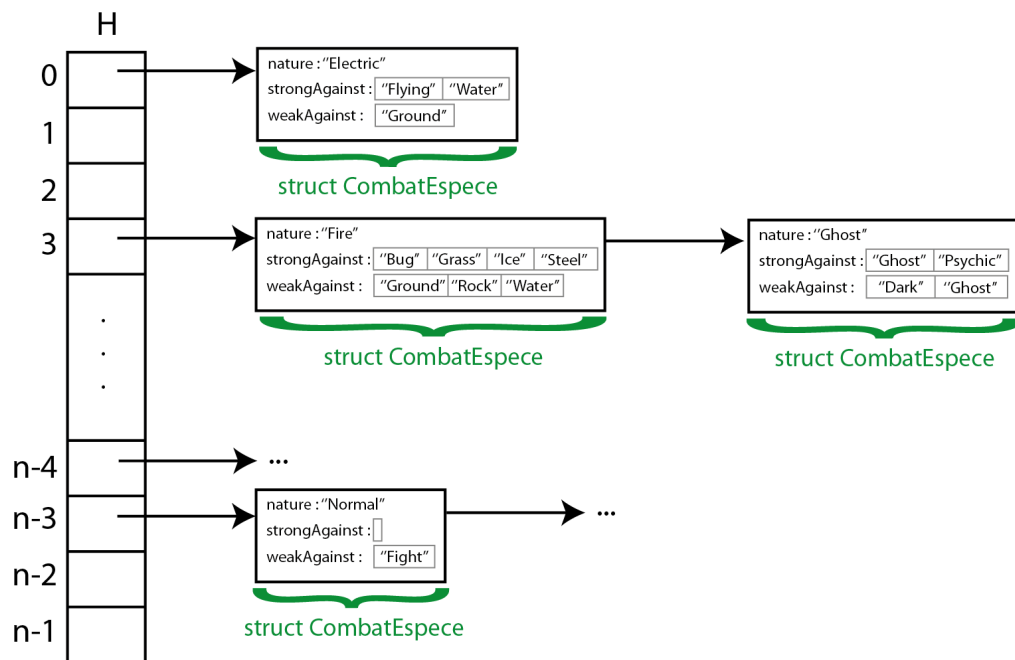


FIG. 2 – Illustration de la table de hachage pour les forces et faiblesses relatives de chaque type de pokemon.

7.2 Structure **CombatEspece**

Chaque élément stocké dans la table de hachage sera de type **CombatEspece**. Cette structure contient comme champs :

- `nature` : la nature du pokemon concerné ("Feu", "Electric", etc.)

- `strongAgainst` : un tableau statique de 5 chaînes de caractères représentant les natures de pokemons contre lesquelles le pokemon de nature `nature` est **fort**.
- `weakAgainst` : un tableau statique de 5 chaînes de caractères représentant les natures de pokemons contre lesquelles le pokemon de nature `nature` est **faible**.
- `next` : un pointeur vers le prochain maillon de la liste chaînée représentant l'alvéole de cette case de la table de hachage – ce sera un pointeur vers une variable de type `CombatEspece`.

Note : Il aurait été plus astucieux de réaliser des tableaux dynamiques que des tableaux statiques pour les champs `strongAgainst` et `weakAgainst`. En effet, si un pokemon n'est fort que contre 2 natures de pokemons, 3 cases resteront vides (initialisées au pointeur `null`). Ce n'est pas optimal, mais le principal dans cette séance n'est pas là – nous souhaitons réaliser une table de hachage et l'utiliser, nous aurions en effet « perdu » du temps à implémenter ces tableaux dynamiques aux dépens de la pratique de l'implémentation d'une table de hachage.

7.3 Table de hachage

Nous nous intéressons maintenant à ajouter une structure dans le fichier `combat.h` représentant la table de hachage illustrée à la figure précédente (Figure 2). Cette structure contiendra deux champs : un entier représentant la taille de la table de hachage (le nombre de cases de la table) et un tableau de taille fixe (tableau statique) de n cases. Chacune des cases de ce tableau est un pointeur vers une variable de type `CombatEspece` : le premier maillon de la liste chaînée pour l'alvéole représentée par cette case. Comme nous disposons de 18 natures de pokemon différentes, notre table de hachage sera de taille fixe n . Cela est par ailleurs plus simple à implémenter pour le temps imparti.

Voici une recommandation d'ordre de travail à réaliser afin de cheminer dans ce travail :

7.3.1 Taille de la table de hachage

Sachant que nous disposons de 18 natures de pokemon différentes, et que nous visons un facteur de charge de 0.7 une fois la table remplie, combien de cases la table de hachage doit-elle avoir ?

7.3.2 Structure pour la table de hachage

Créer la structure `CombatHash` dans le fichier `combat.h`, la table de hachage que nous utiliserons dans la suite de la séance.

7.3.3 Fonction de hachage

Implémenter une fonction de hachage pour nos `CombatEspece`. Il existe déjà de nombreux algorithmes de fonctions de hachage non cryptographiques afin de hacher une chaîne de caractères. Nous choisirons l'algorithme de D. J. Bernstein, qui a le mérite d'être simple et relativement efficace pour notre faible nombre de clés. Un exemple d'implémentation en C est donné au Listing 9. Nous devons simplement modifier cette implémentation afin que la fonction nous renvoie une valeur de hachage modulo la taille de la table de hachage (la table de hachage peut être passée en paramètre et nous pouvons accéder à son champs `size`).

Listing 9: Implémentation de la fonction de hachage de chaînes de caractères selon l'algorithme de D. J. Bernstein

```
int hash(const char *str)
{
```

```
    unsigned long hash = 5381;
    int c;

5    while (c = *str++)
        hash = ((hash << 5) + hash) + c; /* hash * 33 + c */

    return hash;
10 }
```

Notons que nous nous attendons ici à avoir une clé sous la forme d'une chaîne de caractères : la nature du pokemon (« Feu », « Grass », etc.). Il ne vous a peut-être pas échappé que pour le moment la structure définissant les types de pokemon (`EspecesPokemon`) ne contient qu'un type énuméré nommé `PokemonType`. Chaque élément de ce type n'est pas une chaîne de caractères mais un entier, en partant de zéro ! Nous aurions pu utiliser cet entier et appliquer l'opérateur modulo comme fonction de hachage. Cela nous aurait cependant fourni une fonction de hachage assez biaisée : nous aurions rempli les cases de la première à la dernière avant de recommencer ! C'est la raison pour laquelle nous introduisons une fonction dans les fichiers `combat` permettant de récupérer la chaîne de caractères représentant ce label. Cette fonction est donnée au Listing 10.

Listing 10: *Correspondance entre le type énuméré `PokemonType` et la chaîne de caractères représentant la nature du pokemon*

```
char* pokemonTypeToString(PokemonType t) {

    if (t == Normal) {
        return "Normal";
5    } else if (t == Fighting) {
        return "Fighting";
    } else if (t == Flying) {
        return "Flying";
    } else if (t == Poison) {
10    return "Poison";
    } else if (t == Ground) {
        return "Ground";
    } else if (t == Rock) {
        return "Rock";
15    } else if (t == Bug) {
        return "Bug";
    } else if (t == Ghost) {
        return "Ghost";
    } else if (t == Steel) {
20    return "Steel";
    } else if (t == Fire) {
        return "Fire";
    } else if (t == Grass) {
        return "Grass";
25    } else if (t == Water) {
        return "Water";
    } else if (t == Electric) {
        return "Electric";
    } else if (t == Psychic) {
30    return "Psychic";
    } else if (t == Ice) {
```



```

        return "Ice";
    } else if (t == Dragon) {
        return "Dragon";
35    } else if (t == Dark) {
        return "Dark";
    } else {
        return "Fairy";
    }
40 }

```

7.3.4 Primitives pour la table de hachage

Nous souhaitons disposer de quelques primitives sur notre table de hachage – primitives vues en cours –, dont nous vous donnons les prototypes ci-dessous :

- `CombatEspece* get(CombatHash* hashtable, char* key)` : fonction permettant de trouver le pointeur vers un `CombatEspece` dont la nature vaut `key`.
- `bool contains(CombatHash* hashtable, CombatEspece* c)` : fonction permettant de savoir si un `CombatEspece` pointé par le pointeur `c` appartient à la table de hachage. Cette fonction pourra réutiliser la fonction précédente – son implémentation est donnée au Listing 11.
- `void insert(CombatEspece* c, CombatHash* hashtable)` : fonction permettant d'insérer un nouvel élément à la table de hachage.

Listing 11: *Primitive pour savoir si un élément appartient à la table de hachage*

```

bool contains(CombatHash* hashtable, CombatEspece* c) {
    CombatEspece* element = get(hashtable, c->nature);
    return (element != nullptr);
5 }

```

7.3.5 Initialisation des données

On se propose d'écrire une fonction permettant d'initialiser la table de hachage et de remplir les valeurs pour les 18 natures de pokemons existant dans le jeu.

Cette tâche ingrate et chronophage vous est épargnée : vous pouvez récupérer le code cette fonction à l'adresse : <https://gist.github.com/anonymous/a328aa8ad4469f21abd285557a9a7d87>.

7.4 Ajout du combat au menu du jeu

Lors d'un combat de pokemons, nous considérerons que les dommages de base de tous les pokemons sont de 30 PV. Dans le cas où un pokémon *A* est plus fort par nature qu'un pokémon *B* (par exemple un pokemon de type « Grass » est naturellement plus fort contre les pokemons de type « Rock »), le pokemon *A* inflige 3 fois plus de dégâts.

Le travail à réaliser pour ajouter le combat dans le jeu est le suivant :

1. Modifier le menu pour que dans le cas du sous menu 6 (combats), nous appelions l'affichage du pokedex, puis une fonction appelée `combat` dont la définition sera donnée dans le fichier `combat.cpp`.
2. Cette fonction demande à l'utilisateur les identifiants des deux pokemons à faire combattre. Leur nature permet de déterminer si le premier est fort sur le second et vice-versa.

3. Déterminer ensuite dans cette fonction qui commence (tirage aléatoire),
4. Itérer en appliquant des dégâts (incluant le potentiel bonus dû à la nature des pokemons) jusqu'à ce qu'un pokemon n'ait plus de PV.
5. Afficher le vainqueur.

8 Séance 7 : Récapitulons, déboggons, consolidons

Cette séance est dédiée à un travail personnel vous permettant de terminer votre version du jeu, en empruntant ou non des éléments de solutions. Vous pouvez également vous amuser à ajouter des fonctionnalités à votre jeu (utiliser les points de combats dans les dommages infligés, afficher d'autres informations sur les pokemons, etc.). L'objectif est de prendre le temps de repasser en revue le chemin parcouru depuis la première séance et de lever vos interrogations quant à des aspects qui vous poseraient encore problème. C'est une séance de consolidation de vos acquis!

Quelle joie d'arriver à implémenter un programme complet avec lequel on peut interagir.
Soyez fier/fière de ce que vous avez appris et réalisé!

A Sortie standard du programme de test du pokedex

***** Pokedex after inserting pokemon a *****

Contenu du pokedex (Actuellement 1 pokemons capturés)

Seadra | Water | (Evolution : 1)
PV : 23 | XP : 1 | CP 4

***** Pokedex after inserting pokemon b *****

Contenu du pokedex (Actuellement 2 pokemons capturés)

Arbok | Poison | (Evolution : 2)
PV : 134 | XP : 0 | CP 123

Seadra | Water | (Evolution : 1)
PV : 23 | XP : 1 | CP 4

***** Pokedex after inserting pokemon c *****

Contenu du pokedex (Actuellement 3 pokemons capturés)

Arbok | Poison | (Evolution : 2)
PV : 134 | XP : 0 | CP 123

Seadra | Water | (Evolution : 1)
PV : 23 | XP : 1 | CP 4

Zubat | Poison | (Evolution : 1)
PV : 223 | XP : 40 | CP 300

***** Pokedex after inserting pokemon d *****

Contenu du pokedex (Actuellement 4 pokemons capturés)

Arbok | Poison | (Evolution : 2)
PV : 134 | XP : 0 | CP 123

Paras | Bug | (Evolution : 1)
PV : 90 | XP : 0 | CP 23

Seadra | Water | (Evolution : 1)

PV : 23 | XP : 1 | CP 4

Zubat | Poison | (Evolution : 1)

PV : 223 | XP : 40 | CP 300

***** Pokedex after inserting pokemon e *****

Contenu du pokedex (Actuellement 5 pokemons capturés)

Arbok | Poison | (Evolution : 2)

PV : 134 | XP : 0 | CP 123

Arkanine | Normal | (Evolution : 2)

PV : 290 | XP : 0 | CP 23

Paras | Bug | (Evolution : 1)

PV : 90 | XP : 0 | CP 23

Seadra | Water | (Evolution : 1)

PV : 23 | XP : 1 | CP 4

Zubat | Poison | (Evolution : 1)

PV : 223 | XP : 40 | CP 300
