Les MinimesEnUnClic

Master de Bioinformatique - Parcours Biologie Computationnelle

Cornier Alexandre

A rendre pour : Mme. Beurton

8 Avril 2020

Résumé

Dans le cadre de notre projet en programmation orientée objet, nous devons réaliser un programme permettant la gestion du port de plaisance des Minimes à La Rochelle et permettre ainsi une comptabilité exemplaire. Pour cela le port dispose d'une liste d'abonnés, d'une liste de places pour les bateaux et une liste de tous les différents services qui sont proposés aux clients.

Les clients du ports se distinguent en deux catégories :

— **Les abonnés** : possédant une place permanente pour leur bateau et donc qui règlent à la fin de chaque mois sur la base d'une cotisation annuelle

— **Les clients** : qui sont de passage seulement pour quelques jours et qui règlent une taxe par jour en fonction de la durée de leur séjour

Les tarifs que le port propose dépendent également de la taille du bateau de chaque usager. En effet on distingue 3 catégories de bateaux :

— **Les voiliers non habitables** : ce sont des bateaux dont la taille est inférieure à 10 mètres de long ne possédant pas de cabine. Ces bateaux n'utilisent pas les services du port tel que l'électricité ou l'eau

— **Les voiliers de type 1** : ce sont des bateaux dont la taille est comprise entre 10m et 25m de long avec une ou plusieurs cabines, ils peuvent utiliser les services de branchement à l'électricité et à l'eau

— **Les voiliers de type 2** : ce sont des bateaux identiques aux voiliers de type 1 mais avec une taille supérieure à 25m. Ce type de bateau nécessite une place spécifique, de grande taille, et celles-ci sont en nombre limité

Le but du programme LesMinimesEnUnClic est de pouvoir proposer des places disponibles et d'automatiser la gestion du paiement de chaque client. Pour cela on doit être capable d'enregistrer un nouvel usager en renseignant toutes les informations que nous avons vues précédemment. Il doit permettre également l'enregistrement d'un nouveau paiement et enfin d'éditer une facture, celle-ci, bien sûr devant être conservée dans un fichier historique.

Sommaire

Chapitre 1

Analyse du sujet

1.1 Matériel et logiciels utilisés

Dans le cadre de ce projet, nous avons simplement besoin d'un ordinateur sur lequel est installé un éditeur de code. Nous avons choisie Visual Studio Code développé par Microsoft pour Windows, Linux et macOS car il possède de nombreux avantages :

- Il est gratuit et simple d'utilisation
- Il possède des milliers d'extensions qui sont développées par la communauté
- Il est relativement peu exigeant en ressources machine
- Il supporte la majorité des langages de programmation

Concernant le langage utilisé pour ce projet, ce sera le C++. C'est un langage de programmation compilé très populaire dit de « bas niveau », c'est à dire un langage plus proche du fonctionnement de la machine permettant plus de contrôle sur les différentes actions que l'on va mener. Ce type de langage peut alors sembler complexe au premier abord mais avec de la pratique, c'est celui qui allie le mieux puissance et rapidité.

Il permet également d'utiliser le langage objet, ce qui permet d'offrir une vision plus proche de la réalité physique et des concepts humains. Cela permet de représenter des objets ainsi que leurs relations, dans un logiciel, comme des entités à part entière et ensuite de les faire interagir entre elles. Dans notre cas la création d'un objet « port » prend tout son sens.

1.2 Hypothèses

Après lecture et analyse du sujet, nous avons choisie de définir un certain nombre d'hypothèses afin de construire notre programme :

- Pour cet exercice, le choix proposé est d'utiliser des containers map qui représentent les objets usager, bateau et place de port, et de définir une classe de gestion du port qui va à la fois contenir l'enregistrement de chaque séjour, et assurer les relations avec les objets map
- Aucun nom ni numéro en double (utilisateur, bateau, place), ce qui permet d'utiliser le nom d'usager, de bateau et le numéro de place en tant que clé pour les map. Il est

toujours possible d'ajouter un caractère ou un numéro pour gérer d'éventuels doublons

— Un bateau peut être utilisé par des usagers différents (location)

— Un usager peut utiliser plusieurs bateaux (en propre ou en location), par contre un usager ne peut avoir qu'un seul abonnement. Si un abonné possède plusieurs bateaux avec abonnement, il faudra créer des noms d'usagers différents

— Aucun nom avec espace, par contre les tirets sont acceptés pour les noms composés

— On ne supprime jamais un usager ni un bateau

— Les places sont gérées directement dans le fichier texte (configuration)

— Étant donné que les noms et n° de place sont les clés, et que les données sont stockées dans des fichiers textes lorsque le programme ne tourne pas, nous dupliquons les valeurs au lieu d'utiliser des pointeurs vers le contenu des containers map

— Le nombre de séjours, correspondant à des instances de la classe GestionPort n'est pas connu à l'avance et peut varier pendant la durée de l'utilisation du programme. Le choix d'un map permet de disposer des fonctions de recherche sur la clé

— Nos trois objets usager, bateau, place sont des couples (clé, valeur) pour lesquels il est possible d'utiliser les mêmes types (string, int). Cela permet d'utiliser les mêmes fonctions de lecture, écriture des fichiers texte et des map, sans surcharge ni dédoublement de ces fonctions

— Dans un souci d'intégrité, les données qui peuvent être calculées ne sont pas stockées, par exemple : La disponibilité d'une place de port. A contrario les données de facturation sont enregistrées pour conserver un historique

— Ce programme ne gère pas les dispositifs d'impression, les factures sont générées sous forme d'une liste dans un fichier texte au format CSV qui peut ensuite être utilisé pour un publipostage dans une application de type LibreOffice par exemple

— Nous utilisons le moins possible de fonction inline pour accéder aux attributs d'une instance de la classe GestionPort pour réduire l'empreinte mémoire globale de tous les enregistrements de séjours

— Le supplément de 5 euros pour chaque service est le tarif par jour. En appliquant le même coefficient de réduction pour les abonnés (environ 0,07), cela correspond à 10 euros par mois. Le système de facturation mis en place permet de choisir les services indépendamment, pour le séjour ou par mois

— Comme pour la plupart des pays, des taxes d'appliquent… nous avons ajouté la TVA à 20%

Chapitre 2

Conception du programme

2.1 Structure des données et implémentation en C++

Avant d'aborder le code lui-même, il est important de comprendre comment la structure des données a été conçue. Le schéma ci-dessous est une représentation similaire à ce qu'on utilise en gestion de base de donnée. Il est basé sur un modèle entité/relation :

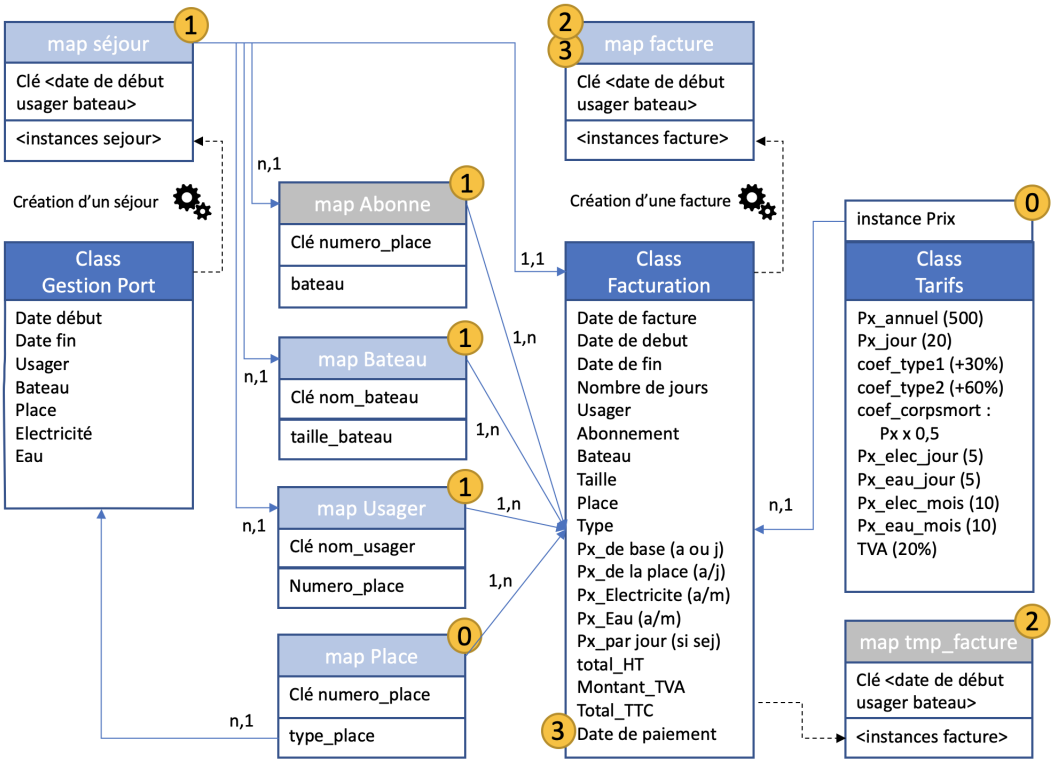

Figure 1 : Schéma représentant la structure du code

Les classe sont utilisées pour définir les tables complexes contenant plusieurs attributs de différents types. Chaque enregistrement dans cette table correspond à une instance de la classe. Par exemple : une facture est une instance de la classe facturation. Pour gérer les multiples instances, nous utilisons un conteneur C++ de type map qui permet à la fois de créer les instances nécessaires en tant qu'attributs et d'accélérer les recherches d'une instance particulière en utilisant sa clé au lieu de faire une boucle sur la totalité des instances de la classe.

Ensuite, pour les tables de correspondance utilisant uniquement une clé et un attribut on utilise un conteneur c++ de type map. Dans ce cas chaque enregistrement de la table correspond à un couple (clé, attribut). La clé doit être unique évidemment.

Pour cet exercice, nous aurions pu utiliser des conteneurs unordered map qui sont moins gourmands en ressources car ils ne trient pas systématiquement les données sur la clé. Cependant le conteneur map permet d'enregistrer les données de manière ordonnée dans les fichiers texte ce qui facilite la recherche si besoin, par exemple pour les séjours ou les factures dont la clé commence par la date inversée.

Les maps décrits ci-dessus sont enregistrés dans des fichiers texte pour garder les données lorsque le programme est arrêté, notamment les usagers, les bateaux, les places du port, les séjours, les factures et les tarifs.

Pour simplifier le code lors de certaines recherches et pour accélérer le programme nous avons ajouté des maps qui servent d'index ou de table temporaire, par exemple :

- **map abonné** : il sert dans notre structure à créer une relation entre une place de port et un bateau pour un abonné. La clé place permet de trouver très rapidement le bateau sans interroger la totalité des instances de la classe gestion port. Cette table est construite au chargement du programme et mise à jour à chaque création d'un nouvel abonnement
- **map tmp facture** : le map principal facture contient toutes les instances de la classe facturation c'est à dire toute les factures. Lorsque l'on crée une ou plusieurs nouvelles factures et que l'on souhaite les exporter en réutilisant le module d'enregistrement existant, il faut disposer d'une liste contenant uniquement les nouvelles factures. C'est le propos de cette table temporaire

Les numéros sur le schéma précèdent correspondent aux différentes étapes au cours desquelles ces objets sont crées. Le processus standard d'utilisation du programme comporte les étapes suivantes :

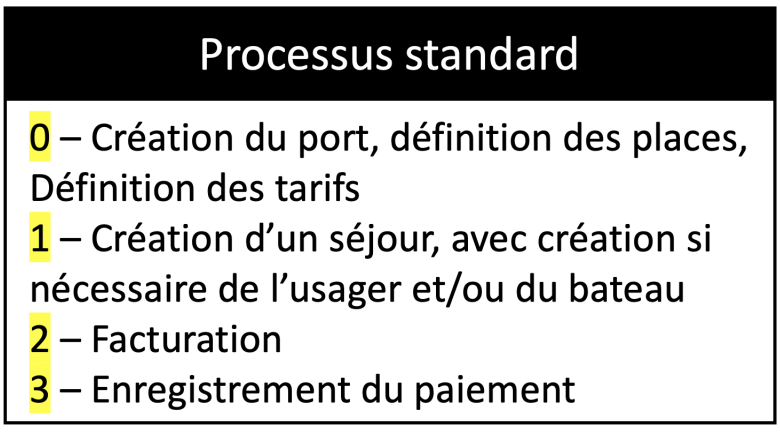

Figure 2 : Schéma représentant le processus standard

2.2 Utilisation d'un map avec les instances de classe en attribut

N'ayant pas de système de gestion de base de données pour stocker et manipuler les différents objets, nous en avons reproduit le principe au moyen du conteneur map pour les tables simples (couple clé/attribut) mais nous avons besoin d'un modèle plus complexes lorsqu'il faut gérer plusieurs attributs pour un même objet. Pour cela, nous nous appuyons sur la définition d'une classe. Un attribut du conteneur map correspond à un pointeur vers une instance de classe.

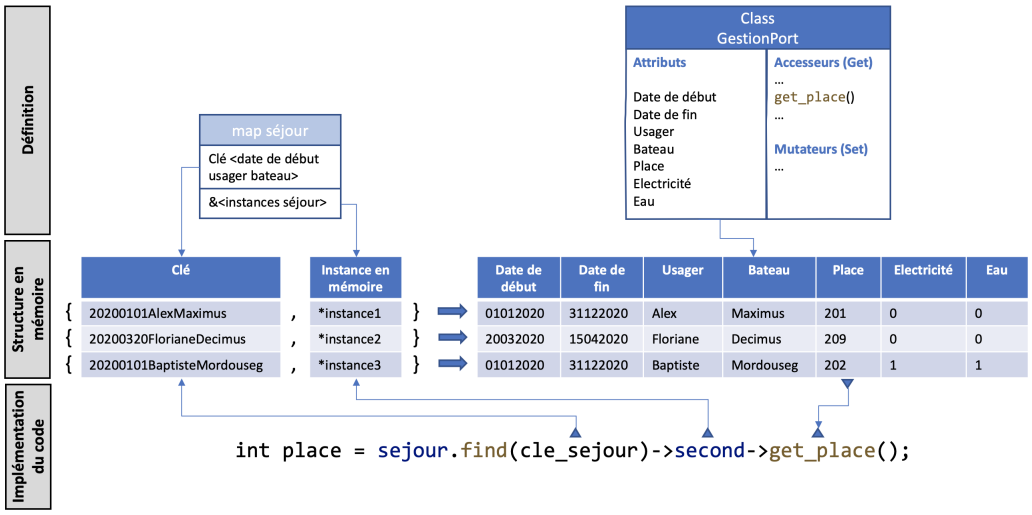

Figure 3 : Schéma représentant l'utilisation de map avec une instance de classe en attribut

2.3 Fonctionnalités du logiciel et structure des menus

Pour cette première version du logiciel nous avons retenu les fonctionnalités suivantes, basées sur notre compréhension du fonctionnement d'un port de plaisance :

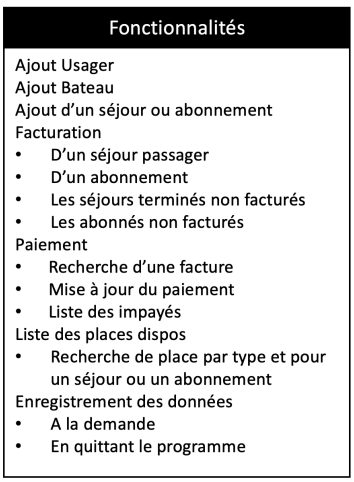

Figure 4 : Schéma représentant l'ensemble des fonctionnalités

Pour présenter ces fonctionnalités de manière logique à l'utilisateur, voici le menu que nous avons conçu :

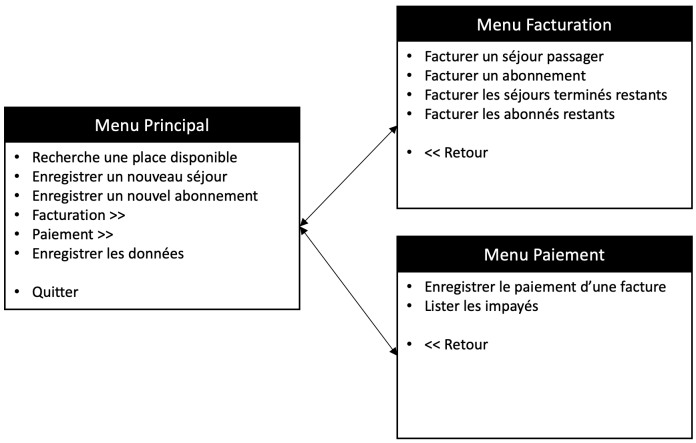

Figure 5 : Schéma représentant le menu du programme

2.4 Architecture du code en C++

Nous avons adopté les pratiques usuelles d'une architecture de programme en C++ :

— Un fichier main.cpp, le plus léger possible qui intégre les menus

— Un couple header *.hpp et code implémentation*.cpp pour chaque classe

— Un couple header *.hpp et code implementation*.cpp pour les ensembles de fonctions correspondant à un groupe fonctionnel, par exemple toutes les fonctions permettant de gérer les factures

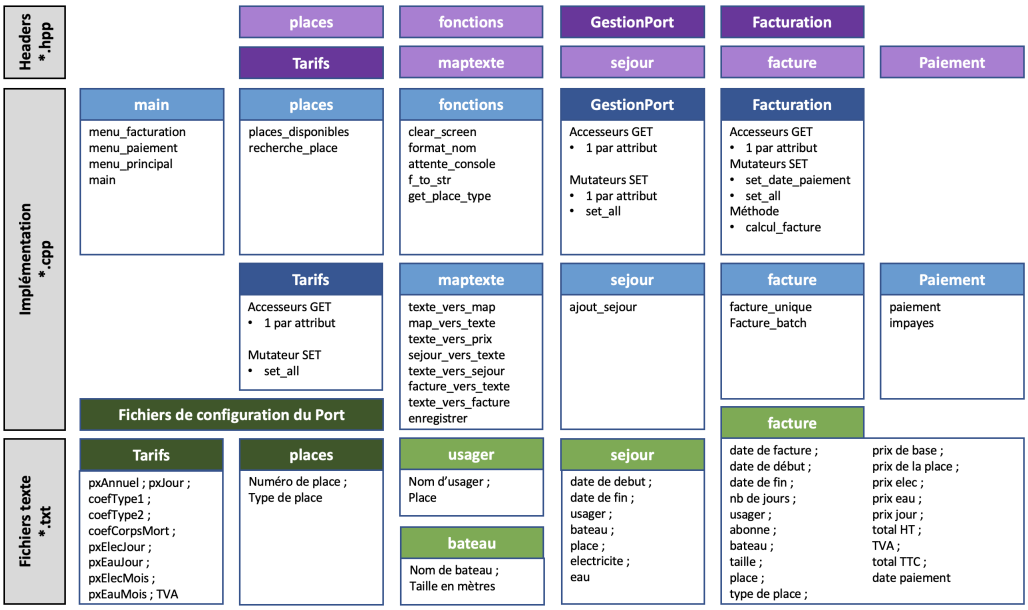

Figure 6 : Schéma représentant l'architecture du code avec ses différentes classes et fichiers

Parmi les bonnes pratiques que nous avons adoptées :

- Ne pas utiliser namesspace using std pour éviter tout risque d'erreur sur le namespace
- Utiliser le moins possible de variables globales et aucune globale externe. Cela a un impact fort sur le code et la quantité d'arguments à passer aux fonctions mais cette pratique garantit la sécurité du code. Un corollaire est l'usage du passage de paramètres par références pour ne pas dupliquer les tables ni gérer des valeurs en retour
- Les variables sont définies de préférence au moment ou elles sont nécessaires et non en entête de fonction pour réduire leur scope au strict minimum. C'est notamment le cas pour les stringstream dont nous réduisons le plus possible la portée car la purge du buffer s'avère compliquée à réaliser
- Pour les accesseurs, mutateurs et méthodes, nous avons choisi de sortir de la classe leur implémentation pour qu'ils ne soit pas inline, réduisant ainsi l'occupation mémoire compte tenue de la quantité potentielle d'instances pour les séjours et les factures. Dans ce programme nous sommes amenés à manipuler beaucoup d'instances mais peu souvent
- Dans le même esprit, nous avons limité au strict minimum les mutateurs par exemple : pour les tarifs et les séjours un seul mutateur suffit pour mettre a jour les valeurs d'une instances en un seul appel de méthode
- Règles d'écritures : nous avons adopté pour l'initialisation des variables la règle définie à partir du c++11 qui utilise les , par exemple "int i;"
- De même, les points virgules qui sont des punctuators ne sont employés que si nécessaire. Ce n'est pas le cas par exemple pour les statements (sauf pour le do while) ni pour les fonctions. Par contre c'est nécessaire pour les structures et classes

2.5 Structure des répertoires pour les fichiers texte

Pour organiser les fichiers selon leur usage, nous avons défini la structure de répertoires suivante. Les Répertoires config et database et les fichiers qu'ils contiennent doivent exister pour accéder au menu du programme.

```
LesMinimesEnUnClic
├── config
│   ├── places.txt
│   └── tarifs.txt
├── database
│   ├── bateaux.txt
│   ├── factures.txt
│   ├── sejours.txt
│   └── usagers.txt
├── doc
│   └── LesMinimesEnUnClic.pdf
├── facture
│   └── Fichiers texte contenant les données de facturation ou les impayes
└── publipostage
    └── Facture_template.odt
```

Chapitre 3

Solutions retenues pour le code

3.1 Gestion d'entrée/sortie console

Nous avons utilisé massivement std : :cout, std : :cin, avec les opérateurs » et « mais nous avons recherché un comportement particulier : créer un prompt avec l'entrée sur la même ligne. Nous devons également vider le buffer pour éviter de propager un surplus de caractères entrés par l'utilisateur. Pour résoudre cela, nous utilisons par exemple le code que nous avons implémenté dans la fonction d'attente d'une entrée utilisateur avant de revenir à un menu pour lui donner le temps de lire le résultat de l'opération réalisée (par exemple la création d'un séjour) :

```
1  int attente_console ()
2  {
3      // Attendre une entree console pour revenir au menu
4      std::cout << "\nEntrez 0 pour revenir au menu)\n";
5      std::cout << "> ";
6      int retour;
7      std::cin >> retour;
8      std::cin.clear();
9      std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
10
11     return retour;
12 }
```

En ligne 5, il n'y a pas de std : :endl ni de caractère de retour a la ligne (
n) donc le std : :cin qui suit est à la suite du dernier caractère affiché. Ensuite les deux lignes clear et ignore permettent de vider le buffer de l'entrée et de ne pas générer de fin de ligne non souhaitée pour éviter de valider par erreur la prochaine saisie console.

3.2 Gestion des fichiers

Nous avons implémenté dans le programme plusieurs fonctions permettant de lire et d'écrire dans des fichiers textes. Cela offre la possibilité de conserver les informations lorsque le programme est arrêté, et aussi de transférer des données à un programme externe, en différé, par exemple les données de facturation pour une impression en batch depuis un logiciel adapté à la présentation et à l'impression comme un tableur ou un traitement de texte.

Cela nécessite de mettre en œuvre deux type d'opérations : lecture et écriture. Au même titre que les entrées/sorties console l'usage en C++ est de s'appuyer sur des flux (streams) et les opérateurs chevrons (« et »). C'est une pratique C++ courante dont nous verrons d'autres usages notamment pour des conversions de type ou des mises en forme (float et date). L'opérateur chevrons donne le sens du flux.

3.2.1 Enregistrement dans un fichier : écriture

Nous avons choisi de créer des fichiers dans un format de type CSV (Comma-seperated values) qui est facile à écrire et à utiliser depuis de nombreux logiciels tiers. Pour le créer il suffit de définir un stream correspondant au fichier texte, dans le sens output, puis d'utiliser les chevrons « « » pour alimenter le flux.
Dans notre cas, les données étant issues d'un un conteneur map, nous avons construit une boucle sur les objets de la map pour enregistrer chaque couple (clé ; attribut) mot à mot avec une fin de ligne. L'un des avantages du stream est que l'opérateur chevrons effectue la conversion du type automatiquement. Par exemple dans l'exemple ci-dessous que nous utilisons pour les tables usagers, place, et bateau, l'attribut est lu comme texte et stocké dans un integer.

```
int map_vers_texte(std::string fichier_texte, std::map<std::string, int> &
    mapptr)
{
    // creation du stream en ecriture du fichier texte
    std::ofstream wr_texte {fichier_texte.c_str()};

    // creation de l'iterateur du map
    std::map<std::string, int>::iterator it_mapptr {};

    if (wr_texte)
    {
        // ligne d'entete de colonnes
        wr_texte << "cle ; attribut" << std::endl;

        for (it_mapptr = mapptr.begin(); it_mapptr != mapptr.end(); ++
    it_mapptr)
        {
            wr_texte << it_mapptr->first << " ; " << it_mapptr->second <<
    std::endl;
        }

        wr_texte.close();

        return 0;
    }
    else {
        std::cout << "ERREUR: fichier" << fichier_texte << " impossible a
    ouvrir en ecriture" << std::endl;

        return 1;
    }
}
```

3.2.2 Lecture d'un fichier et chargement dans un map

Suivant le même principe il faut définir un flux de type input à partir du fichier et utiliser l'opérateur « » ». La difficulté dans le contexte d'une lecture est d'interpréter la structure du fichier pour sélectionner les bonnes données à charger dans le conteneur cible. L'avantage du fichier CSV que nous créons est que nous maîtrisons sa structure. Pour éviter d'entrer dans une boucle infinie nous privilégions une lecture ligne à ligne (getline) qui nous assure de sortir de la boucle après la dernière ligne. Le contenu de la ligne lue par getline est envoyé dans une variable intermédiaire de type string qui est-elle même associée à un flux de type input string (istringstream). Ce flux permet de lire la ligne mot à mot en identifiant la clé, le séparateur (« ; ») et l'attribut. On appelle cette opération « parser ».

```
int texte_vers_map(std::string fichier_texte, std::map<std::string, int> &
    mapptr)
{
    // creation du stream en lecture du fichier texte
    std::ifstream rd_texte {fichier_texte.c_str()};

    if (rd_texte)
    {
        std::string ligne {};
        std::string key_texte {};
        std::string separateur {};
        int attribut_texte {};

        // saut de la premiere ligne
        std::getline(rd_texte, ligne);

        while ( std::getline(rd_texte, ligne) )
        {
            // creation d'un flux string pour charger les donnees de la
    ligne
            std::istringstream iss {ligne};
            iss >> key_texte >> separateur >> attribut_texte;

            iss.clear();

            mapptr[key_texte] = attribut_texte;
        }

        rd_texte.close();
        return 0;
    }
    else {
        std::cout << "ERREUR: fichier " << fichier_texte << " introuvable"
    << std::endl;

        return 1;
    }
}
```

3.3 Gestion des dates

Cet exercice nécessite de gérer plusieurs dates et de les comparer entre elle, par exemple pour calculer le nombre de jours pour un séjour de passage ou encore si une place est occupée entre deux dates.

Une solution serait de manipuler des dates au format string mais cela impose des algorithmes complexes et des manipulations de chaîne de caractère. La solution que nous avons retenue, plus cohérente avec le C++ moderne depuis 2011 est d'utiliser la structure standard « tm » (std : :tm) qui permet de s'appuyer sur les fonctions natives de calcul et de conversion.

3.3.1 Formatage d'une date (console ou fichier texte)

Une des opérations fréquentes est de passer du format tm a un format de type string ou inversement. Pour cela les streams sont encore une fois mis en œuvre, associés aux deux fonctions « get time » et « put time ». L'exemple ci-dessous montre comment récupérer dans une variable de type tm une entrée console au format jj/mm/aaaa.

```
// Entree de date a la console
    std::tm date4 {};
    std::cout << "Entrez une date au format jj/mm/aaaa" << std::endl;
    std::cin >> std::get_time(&date4, "%d/%m/%Y");
    std::cout << "date capturee par la console : " << std::put_time(&date4,
    "%d/%m/%Y") << std::endl;
```

3.3.2 Date courante (local system)

Une autre complexité est de récupérer l'heure et la date système car il faut passer par un type intermédiaire « time_t » dont nous devons limiter la portée car il présente des risques de compatibilité suivant le système. Nous l'utilisons uniquement pour la conversion du local time vers la structures tm, c'est ce que montre l'exemple ci-dessous.

```
// Date courante, locale du systeme
    std::time_t t3 = std::time(nullptr);
    std::tm date_courante = *std::localtime(&t3);
    std::cout << std::put_time(&date_courante, "%c %Z") << std::endl;
    date_courante.tm_hour = 0;
    date_courante.tm_min = 0;
    date_courante.tm_sec = 0;
    std::cout << std::put_time(&date_courante, "%d/%m/%Y") << std::endl;
```

3.3.3 Comparaison date

Il s'agit d'une opération fréquente pour évaluer si une date est avant ou après une autre chronologiquement, ou pour calculer un écart entre deux dates. Pour cela nous nous appuyons sur la fonction « difftime » qui utilise elle aussi des variables de types « time_t » par lequel nous devrons passer pour effectuer l'opération.

time_t étant une valeur en seconde depuis une date fixée par le système (en général le 1er Janvier 1970), le résultat sera un integer en nombre de secondes entre les deux dates, positif ou négatif suivant la chronologie des deux dates.

```
// definition de 2 dates : 20/03/2020 et 25/03/2020
    std::tm date1 = {0,0,0,20,3,120};
    std::tm date2 = {0,0,0,25,3,120};

    // difference en nb de jours
    std::time_t t1 = std::mktime(&date1);
    std::time_t t2 = std::mktime(&date2);

    // chronologie : t1...t2, difftime(t2, t1) effectue t2-t1,
    //ce qui dans ce cas donne un resultat positif (25 mars - 20 mars)
    if ( t1 != (std::time_t)(-1) && t2 != (std::time_t)(-1) )
    {
        int difference = std::difftime(t2, t1) / (60 * 60 * 24);
        std::cout << difference << std::endl;
    }

    // conversion *tm en jj/mm/aaaa
    std::cout << std::put_time(&date2, "%d/%m/%Y") << std::endl;
```

3.4 Calcul du dernier jour du mois

Pour la facturation d'un abonné nous voulons calculer le dernier jour du mois pour présenter la période correspondant au montant mensuel à régler. Pour cela nous devons déterminer si le dernier jour du mois est le 28, 29, 30 ou 31. On trouve sur internet de nombreux algorithmes s'appuyant sur des modulos. Nous avons préféré une solution plus courte qui consiste à s'appuyer sur les algorithmes standards associés à la structure « std : :tm ». En résumé nous positionnons le jour au premier du mois, nous ajoutons un mois puis nous retirons un jour en passant encore une fois par l'intermédiaire d'un type « time_t » pour retrancher le nombre de secondes contenues dans un jour.

```
// Dernier jour du mois en calculant le 1er du mois suivant -1 jour
    m_date_fin = {0,0,0,20,3,120};

    if (m_date_fin.tm_mon == 11)
    {
        m_date_fin.tm_mon = 0;
        m_date_fin.tm_year += 1;
    }
    else
    {
        m_date_fin.tm_mon += 1;
    }
    // valeur d'un jour en nombre de secondes
    const time_t oneDay = 24 *60 *60;
    time_t t_dernier_jour = mktime(&m_date_fin) + (-1 * oneDay);
    m_date_fin = *std::localtime(&t_dernier_jour);
```

3.5 Détection du système d'exploitation (CLS/Clear)

Une autre solution employée dans ce logiciel est l'identification du système tournant le programme par le pré-processeur pour compiler les commandes correspondantes. La problématique décrite ci-dessous est l'usage de la commande system « cls » sur un machine Windows ou « clear » sur une machine Mac ou Linux pour effacer la console.

```
void clear_screen()
{
#if defined(WIN32)||defined(_WIN32) // Windows
    system("cls");
#else
    system("clear"); // ux, mac
#endif
}
```

Chapitre 4

Exemple de manipulation du programme (expérience utilisateur)

Dans cette partie nous allons simuler une expérience utilisateur. Quand on lance le programme, nous arrivons directement sur le menu principal comme on peut le voir ci-dessous. On demande a l'utilisateur de faire un choix suivant l'action qu'il souhaite réaliser.

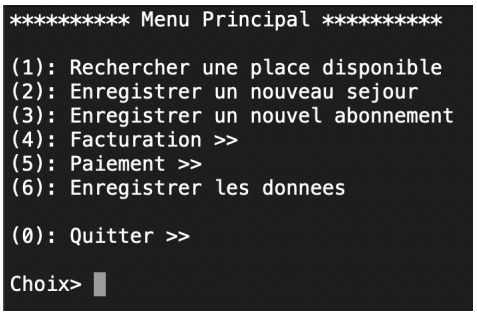

Figure 7 : menu principal

4.1 Recherche de places disponibles

Pour rechercher les places disponibles, il suffit d'entrer le chiffre 1 a la console. On va accéder à une nouvelle interface, nous demandant de renseigner différentes informations : s'il s'agit d'une recherche pour un abonné ou non, la date du début et fin de séjour en faisant bien attention au format demandé (jj/mm/aaaa) et au type de place. En fonction des différentes informations, le programme renvoie la liste des places disponibles.

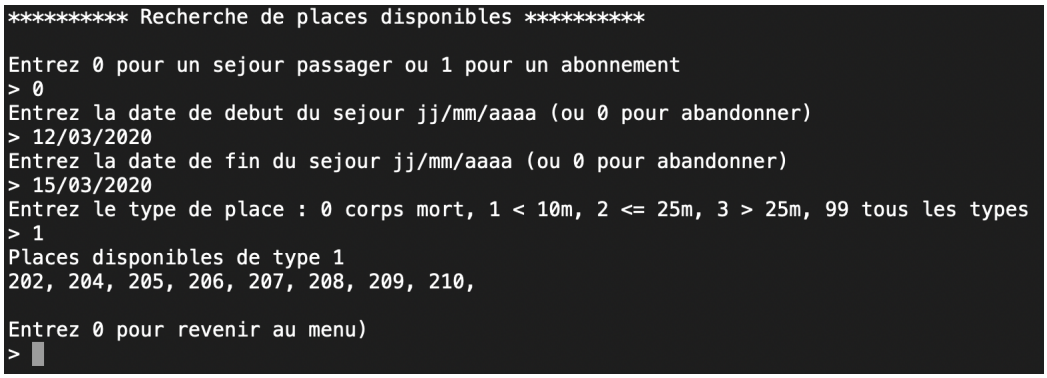

Figure 8 : Recherche de places disponibles

4.2 Cas d'un séjour

4.2.1 Création d'un séjour

Pour créer un nouveau séjour, il faut choisir le choix numéro 2. Il va falloir entrer les dates du début et fin de séjour, puis le nom de l'usager et le nom du bateau avec sa taille, le genre de place souhaité (corps mort ou une place à quai en fonction de la taille du bateau). Le programme liste les différentes places disponibles. Ensuite le programme demande si les services du port seront utilisés, à savoir l'eau et l'électricité, s'il s'agit d'une place à quai de type 2 ou 3.

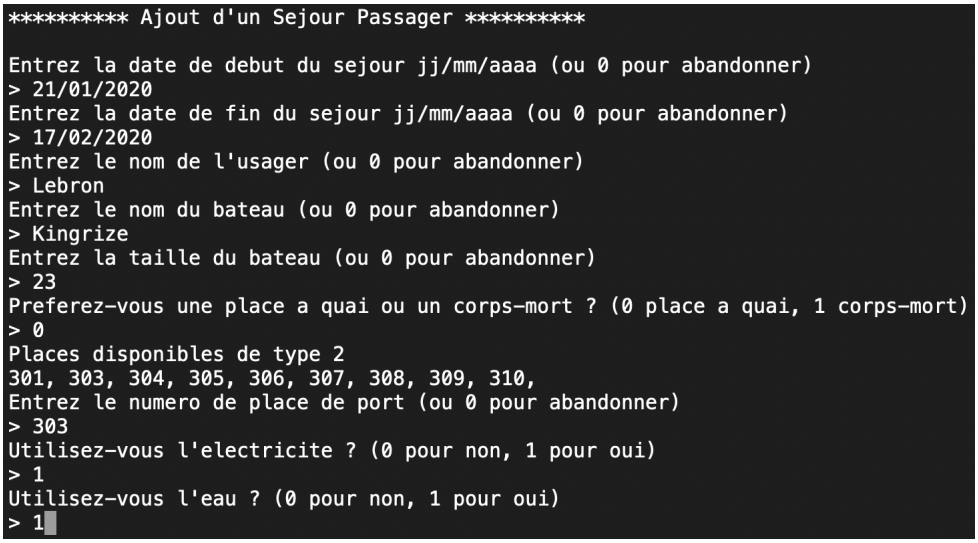

Figure 9 : Enregistrement d'un séjour partie 1

Une fois l'ensemble des données renseignées par l'utilisateur, le terminal affiche un résumé de l'ensemble des informations du séjour comme le montre la figure ci-dessous. Pour revenir au menu principal, il suffit de choisir le choix 0.

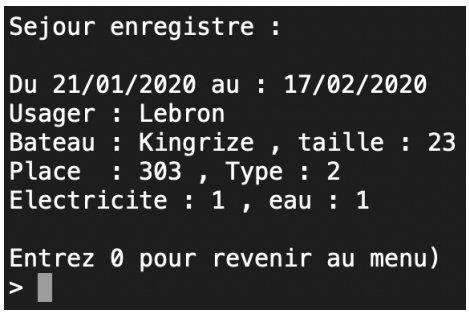

Figure 10 : Enregistrement d'un séjour partie 2

4.2.2 Facturation du séjour

On choisissant l'option numéro 4, on peut accéder à un sous menu permettant de gérer les différentes factures en fonction du type d'usager. Dans notre exemple nous allons facturer un séjour passager.

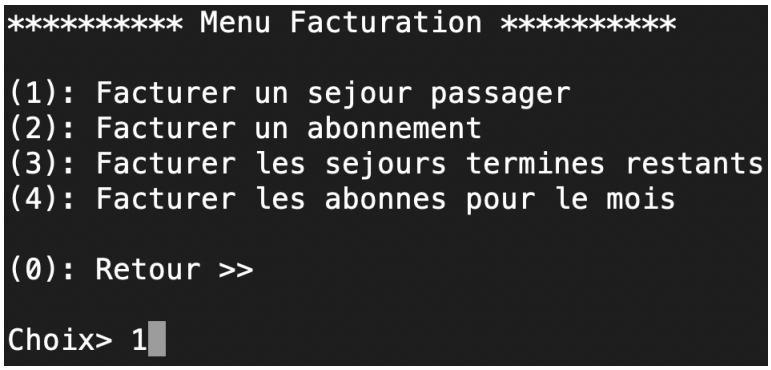

Figure 11 : Sous menu facturation

Pour faire une facture, il faut remplir à nouveau les champs début de séjour, nom du bateau ainsi que celui de l'usager. Ensuite le programme nous affiche un message nous avertissant que la facture est bien enregistrée dans un fichier dont le noms apparaît à l'écran.

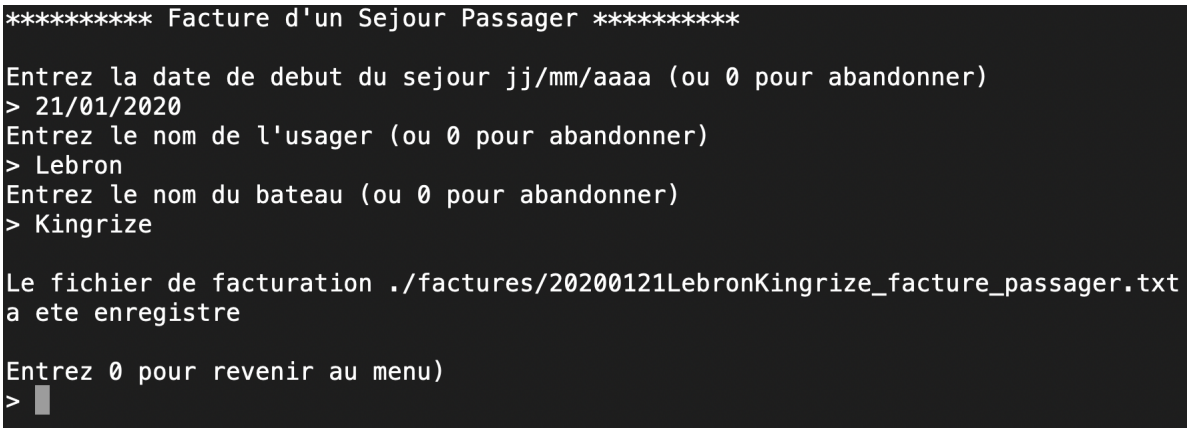

Figure 12 : Enregistrement facture

4.2.3 Recherche des impayés

Pour accéder la liste des impayés, il faut choisir le choix numéros 5 dans le menu principal. On accède ainsi au sous menu paiement, à partir duquel l'option "lister les impayes" est disponible. Si des factures non déjà réglées correspondent à la recherche, le programme affiche le nom du fichier contenant les données de facturation correspondantes.

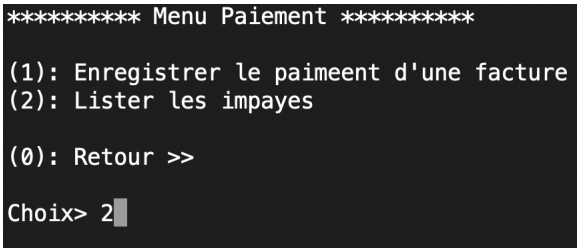

Figure 13 : Liste des impayés partie 1

```
Le fichier des factures impayees ./factures/20200407_factures_impayees.txt
a ete enregistre

Entrez 0 pour revenir au menu)
> 
```

Figure 14 : Liste des impayés partie 2

4.2.4 Paiement du séjour

Enfin pour effectuer le paiement d'un séjour, il faut choisir l'option 5 comme pour la recherche des impayés (figure 13) sauf que cette fois nous réalisons la première action en faisant le choix numéro 1. On renseigne alors les différentes informations demandées puis le programme affiche le montant du séjour et demande à l'utilisateur de confirmer l'enregistrement. Un fois la validation effectuée, la date du jour est enregistrée comme date de paiement de la facture.

```
********** Enregistrement d'un paiement **********

Entrez la date de debut du sejour jj/mm/aaaa (ou 0 pour abandonner)
> 21/01/2020
Entrez le nom de l'usager (ou 0 pour abandonner)
> Lebron
Entrez le nom du bateau (ou 0 pour abandonner)
> Kingrize
Confirmez-vous le montant de 1166.40 TTC ?> oui

Paiement enregistre pour l'usager Lebron
et le bateau Kingrize
pour le sejour commencant le 21/01/2020

Entrez 0 pour revenir au menu)
> 
```

Figure 15 : Liste des impayés partie 2

4.2.5 publipostage des factures à partir du fichier de données

Ce logiciel a été développé pour un usage exclusivement dans une console (ou terminal). Cela le rend très simple et très rapide mais il ne dispose ni d'un gestionnaire graphique pour la composition du modèle de facture, ni d'un gestionnaire d'impression pour les éditer. La solution que nous proposons est de s'appuyer sur la fonctionnalité de publipostage (mailing) que l'on trouve dans la plupart des logiciels de traitement de texte, par exemple LibreOffice ou Word.

Notre logiciel produit pour chaque "édition" un fichier texte au format CSV qui peut être utilisé comme source de données à partir d'un modèle de document pour intégrer des champs qui seront remplacés par les données, pour chaque ligne du fichier, au moment de l'édition. Le schéma ci-dessous illustre la procédure. Il suffit de remplacer le fichier texte pour éditer une nouvelle série de factures. Si le nom du fichier est modifié, une manipulation dans l'outil de publipostage permet de "rebrancher" la base de données interne du modèle, sur le nouveau fichier source.

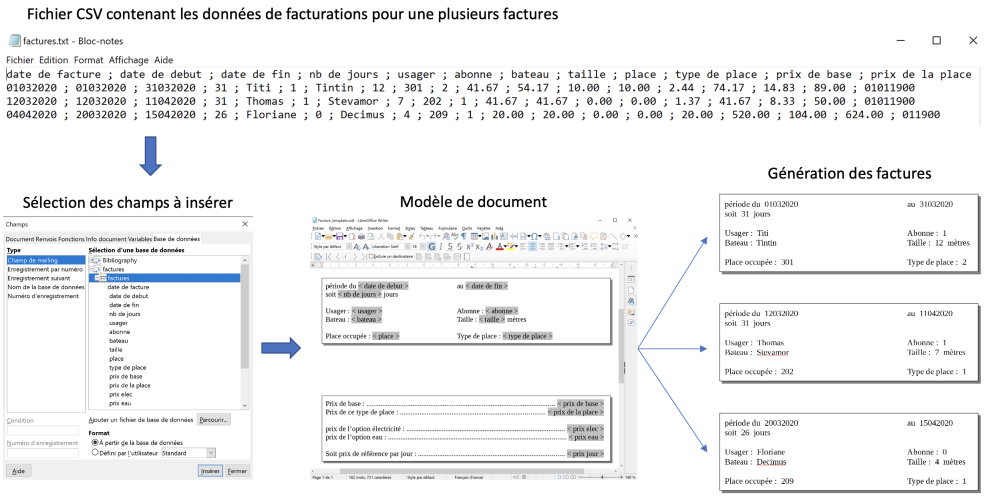

Figure 16 : Publipostage des factures

Chapitre 5

Conclusion et références

Le but de ce projet est d'automatiser la gestion du port Les Minimes de la Rochelle. Pour cela, nous avons implémenté 3 classes : GestionPort, Facturation et tarifs. Ensuite pour construire notre base de données nous avons utilisé des containers std : :map permettant de lier une clé au contenu associé. Dans notre programme nous avons utilisé ces containers à de nombreux endroits comme par exemple le conteneur sejour, bateau etc...(voir figure 1).
Concernant l'utilisation du programme, l'utilisateur peut naviguer dans les menus en fonction

des ses besoins. Tous les enregistrements de séjours ou d'abonnés sont traités en mémoire, pour les stocker dans les différents fichiers il suffit soit de quitter le programme soit de choisir l'option 6 du menu permettant l'enregistrement des données.

Pour ce projet, nous avons pris des hypothèses qui simplifient les objets usagers, bateaux et places de port, en particulier pour les clés qui interdisent les homonymes pour les usagers et les bateaux. Mais dans une version plus élaborée du programme, nous pourrions ajouter :

- Prise en charge des homonymes et ajout d'attributs complémentaires pour les objets usager, bateau, place. L'utilisation de clés plus complexes implique une architecture plus complexe également, ainsi que des fonctions ou méthodes de recherche plus élaborées.

- Prise en charge des noms avec espace. Il en va de même pour la lecture des fichiers texte mot par mot, avec un découpage très simple. Cela interdit par exemple les noms composés avec espace pour les usagers et les bateaux. Un parser plus complexe, à même de prendre en charge un format CSV en lecture par exemple, permettrait de gérer ces noms composés.
- Implémentation d'un Système de Gestion de Bases de Données pour déporter la gestion des données et pour permettre le multi-utilisateurs (exemple : MySQL, SQL Express ou PostgreSQL. Ces solutions simplifient considérablement la gestion des données, en apportant des fonctionnalités performantes de recherche et de tri ainsi qu'un meilleur contrôle de l'intégrité des données.
- Interface Graphique en utilisant les librairies GUI (Graphic User Interface), notamment Qt ou WxWidgets.
- Gestion d'erreurs plus complète. En exploitation il faut se prémunir contre le maximum possible d'erreurs de saisies, volontaires ou non. Cela prend du temps à développer et surtout à tester.

Les références que nous avons utilisées tout au long de ce projet sont les suivantes :

- http://stackoverflow.com
- http://www.cppreference.com
- https://openclassrooms.com/fr
- https://www.geeksforgeeks.org