Barnabé Petit – TC02
Félix Houdebert - TC02
Groupe de TD du jeudi 14h00 – 16h00

NF05
A2020

Projet : « Simulation d'un service d'urgences »

printf(" Développement d'un programme en langage C \n") ;

30/12/2020

UTT UNIVERSITÉ DE TECHNOLOGIE TROYES

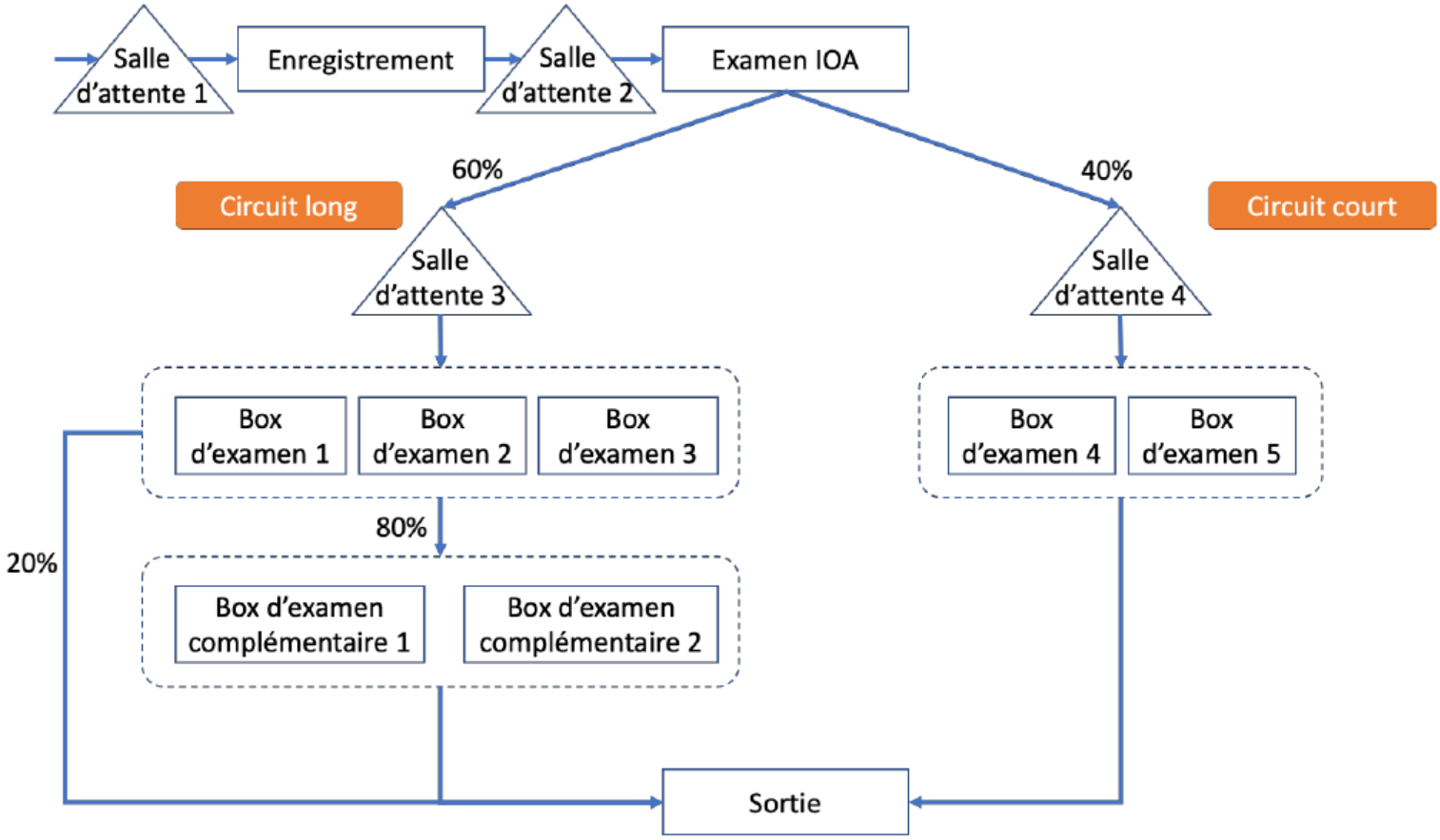
Sommaire

I - Présentation

1.1 - Objectifs

L'objectif était de concevoir et de développer un programme permettant de simuler le flux de patients d'un service d'urgences en **langage C**.
L'utilisateur devait être en mesure **de contrôler les différents paramètres de la simulation** (nombre de patients, heures d'arrivée, durée des tâches ...) avant de l'exécuter.
Une fois la simulation terminée, il fallait mettre à la disposition de l'utilisateur différents outils lui permettant de **consulter** les étapes du flux de patients et d'**afficher des statistiques** établies à partir des données du modèle.

Le flux de **patients** est conçu en adéquation avec le schéma suivant :



On considère que les salles d'attentes peuvent accueillir **un nombre illimité de patients** et que **l'ordre de priorité** des patients correspond à leur **ordre d'arrivée** dans la salle d'attente.

On établit également que les salles et box d'examen ne peuvent contenir **qu'un patient à la fois**.

Enfin, on distingue **trois différents parcours** pour les patients : un circuit court, un circuit long sans examen complémentaire et un circuit long avec examen complémentaire.
On considère que **40%** des patients sont orientés en circuit court et les **60%** restants en circuit long.
Parmi ces derniers, 80% doivent également subir un examen complémentaire avant de quitter l'hôpital.

Notre choix s'est porté sur ce sujet pour différentes raisons :

- C'est un projet concret et appliqué. Nous serons peut-être amenés à concevoir d'autres simulations de ce type à l'avenir, dans nos études comme dans le cadre de notre future profession.
- Les différentes notions à aborder (*utilisation de lois probabilistes, gestion du temps, gestion de différentes structures de données, manipulation de pointeur, fonctions de saisie et d'affichage ...*) sont variées et approfondies : ainsi ce projet nous paraissait-il constituer un bon apprentissage du langage C.
- Enfin, il représentait pour nous un certain défi et semblait être, des trois sujets, celui qui laissait le plus de place à la créativité.

1.2 – Principe du programme réalisé

Le programme permet d'exécuter **plusieurs simulations indépendantes** les unes des autres.
On décomposera les différentes étapes d'une simulation comme suit :

- **Choix des paramètres** : au lancement d'une nouvelle simulation, l'utilisateur peut utiliser les **paramètres par défaut** ou les modifier à partir du **menu des options.**

- **Instanciation des patients** : l'utilisateur doit ensuite créer les patients intervenant dans la simulation. Il peut les générer **automatiquement** (il faut alors préciser leur nombre et leur plage horaire d'arrivée), ou **manuellement**, en précisant à chaque fois leur nom, leur date et leur heure d'arrivée, ainsi que d'autres informations spécifiques à chaque patient selon les options choisies.

- **Flux de patients** : le programme procède ensuite à la simulation du flux de patients. Il affiche dans la console son déroulement, tout en le sauvegardant dans un fichier texte.

- **Menu de fin de simulation** : l'utilisateur peut ensuite saisir différentes commandes lui permettant d'afficher l'**historique** de l'ensemble de la simulation ou d'un patient unique, ainsi que différentes **statistiques** établies sur la base des données de la simulation.
L'utilisateur peut alors mettre fin à la simulation et en lancer une nouvelle, ou quitter le programme.

1.3 - Mode d'emploi

1.3.a – Déroulement d'une simulation :

Lancement d'une simulation :

```
Bienvenue dans ce logiciel de simulation de service des urgences.

Souhaitez-vous demarrer une simulation ? (o/n) :
O
```

Saisir 'O' pour **lancer une première simulation.** Saisir 'N' pour **mettre fin au programme.**

Choix des paramètres :

```
Souhaitez-vous utiliser les parametres par defaut ? (o/n) :
O
--------------------------------------------------------

CONFIGURATION : DEFAUT
        Generation des patients : AUTO
        Generation des durees : AUTO
        Determination du circuit : MANUELLE
        Limitations temporelles : JOUR
        Bip sonore horripilant : AVEC
```

Saisir 'O' pour **conserver les paramètres par défaut.** Saisir 'N' pour **éditer les options de la simulation.**

Génération des patients :

```
GENERATION DES PATIENTS :
--------------------------------------------------------
```

Mode **automatique** :

Il faut indiquer une plage horaire (**début** et **fin**) d'arrivée des patients, ainsi qu'un **nombre de patients.**

```
Date d'arrivee des patients :
        Entrez la date au format jj/mm/aaaa :  21/12/2020
Heure de debut de la plage horaire d'arrivee des patients :
        Entrez l'heure au format hh:mm :  10
Fin de la plage horaire d'arrivee des patients :
        Entrez l'heure au format hh:mm :  12:05
```

Les informations demandées sont différentes en fonction des **paramètres temporels** de la simulation.
Dans la figure ci-dessus, l'option « jour fixé » est activée.

Il faut donc d'abord saisir le jour, puis l'heure de début, puis l'heure de fin de la plage horaire durant laquelle les patients arrivent. La fin doit être postérieure au début.
La saisie est sécurisée et le programme prend en charge les erreurs.

```
Nombre de patients a simuler (recommande : 5 < nb < 58) : 21
```

Un nombre de patients dépendant de la plage horaire est conseillé – mais la simulation marchera aussi s'il n'est pas respecté – il suffit qu'il soit entier et strictement positif.

```
19 - 11:57 : Patient  19 (Eugene Poubelle)
20 - 12:05 : Patient  20 (Edward Kasner)
21 - 12:14 : Patient  21 (Celine Evitable)
```

Les patients se verront attribuer automatiquement un id, une heure d'arrivée, un prénom et un nom.

Mode **manuel** :

Disponible après l'activation d'une **option** (*commande 'p' du menu des options*).
L'utilisateur doit saisir les informations demandées pour chaque patient. (*dépendantes des options activées – au minimum, prénom, nom, et heure d'arrivée*).
L'utilisateur peut créer autant de patients qu'il le souhaite : après l'ajout de chaque patient, il est invité à en **saisir un nouveau** ou à **exécuter la simulation**.

```
Mois et annee d'arrivee des patients :
        Entrez le mois et l'annee auxquels se limitera la simulation au format mm/aaaa :  12/2020

Veuillez entrer a la suite les informations demandees pour chaque patient


        Nouveau patient (0 au total) :
Prenom : Toto
Nom : Toulmonde
Jour et heure d'arrivee au service des urgences :
        Entrez le jour au format jj (/12/2020) :  21
        Entrez l'heure au format hh:mm :  10:05
        Circuit emprunte par le patient (c/l) : l
        Le patient necessite-t-il un examen complementaire ? (o/n) :
o
        Duree de l'examen complementaire :
        Entrez la duree de la tache au format hh:mm ou mm :  10

Voulez-vous ajouter un nouveau patient ? (o/n) :
o
```

Sur la figure ci-dessus, les options « patient généré manuellement », « mois fixé », « choix manuel du circuit » et « choix manuel des durées » sont actives.

Flux de patients :

Par défaut, le flux de patients est géré de manière 100% **automatique.**

```
DEBUT DE LA SIMULATION :
________________________________________________________________________________

        Le : 21/12/2020
10:00 : Le patient   1 (Oceane Picard) arrive aux urgences et entre en salle d'attente 1.
      : Le patient   1 (Oceane Picard) entre en salle d'enregistrement.

10:03 : Le patient   1 (Oceane Picard) entre dans la salle d'attente 2.
      : Le patient   1 (Oceane Picard) entre en salle d'examen IOA.
```

(L'activation de l'option « *durées des tâches saisies par l'utilisateur* » (commande 'd' du menu des options) imposera cependant à l'utilisateur de **saisir la durée** de chaque événements.)

```
10:00 : Le patient   1 (Romeo Debuduphilm) arrive aux urgences et entre en salle d'attente 1.
      : Le patient   1 (Romeo Debuduphilm) entre en salle d'enregistrement.
        Entrez la duree de la tache au format hh:mm ou mm :  3
```

Menu de fin de simulation :

Une fois tous les patients sortis du service des urgences, le **menu principal de fin de simulation** s'affiche.

```
FIN DU FLUX DE PATIENTS



DONNEES DE LA SIMULATION

Saisissez l'une des commandes suivantes :
        - 's' pour afficher les statistiques de la simulation
        - 'h' pour afficher l'historique des evenements
        - 'p' pour afficher les horaires de passage d'un patient dans les differents services
        - 'q' pour mettre fin a la simulation
s
```

Saisir 's' permet d'afficher un **menu secondaire** proposant **de nombreuses commandes.**

Saisir q permet de **mettre fin à la simulation** pour en relancer une nouvelle, ou quitter le programme.

1.3.b – <u>Saisies :</u>

A plusieurs reprises, des saisies sont attendues de l'utilisateur. La plupart des saisies sont sécurisées.

Lorsqu'un format spécifique est attendu de l'utilisateur, il est généralement indiqué.

Voici un exemple du fonctionnement des différentes catégories de saisies que vous pourrez rencontrer :

Yes/No questions :

Ce type de saisie permet d'obtenir une **réponse positive ou négative** de l'utilisateur.

Elle n'est **pas sensible à la casse.**

Seuls les caractères 'o'/'O' et 'n'/'N' – correspondant respectivement à « *oui* » et « *non* » sont acceptés.

```
Souhaitez-vous demarrer une simulation ? (o/n) :
c'est la mer noire
Entree non valide. Veuillez entrer uniquement 'o' ou 'n'.
O
```

Saisie d'une durée :

Selon les paramètres temporels sélectionnés, vous serez amenés à saisir des dates complètes ou partielles ainsi que des heures.

Dates ***:***

```
Date d'arrivee des patients :
        Entrez la date au format jj/mm/aaaa :  32/12/2020  ✘
        Jour incorrect. Il n'y a que 31 jours dans ce mois.
        Entrez la date au format jj/mm/aaaa :  29/02/2019  ✘
        Jour incorrect. Fevrier n'a que 28 jours les annees non bissextiles.
        Entrez la date au format jj/mm/aaaa :  3 ✘
        Format incorrect.
```

Seules des dates valides sont acceptées. En cas de saisie invalide, le programme réitèrera la demande.

```
        Entrez la date au format jj/mm/aaaa :  21/12/2020
```

Les mêmes critères de validité s'appliquent lorsque le programme vous demande de saisir un jour ou un mois uniquement.

Heures et durées :

Vous pouvez saisir des heures et des durées allant de **00:00** à **23:59.**

```
Entrez l'heure au format hh:mm :  24:30
Erreur : le nombre d'heures ne peut exceder 23.
Entrez l'heure au format hh:mm :  23:60
Erreur : le nombre de minutes ne peut exceder 59.
```

Le **format** pour la **saisie d'une heure** est *hh:mm,* mais les formats *h:mm / h:m / hh / h* sont aussi acceptés.

`Entrez l'heure au format hh:mm :  01:00` / `1:00` / `1` …

Pour les **durées**, le **nombre de minutes pourra être indiqué directement** si le nombre d'heures est nul :

`Entrez la duree de la tache au format hh:mm ou mm :  30`

Noms et prénoms :

Vous pouvez écrire les prénoms et les noms choisis manuellement comme vous le souhaitez :
Les espaces, signes de ponctuation et chiffres sont autorisés.
Notez que la limite est de 39 caractères et les accents du français ne seront pas affichés correctement dans le shell par défaut sous Windows.

```
Prenom : Les-prenoms-composes-c'est-pas-pratique
Nom : xX_d4rk_Sh4d0w_Xx
```

Commandes :

Les commandes disponibles pour chaque menu sont indiquées avant chaque saisie.
La saisie attendue consiste en un **caractère unique.** Elle n'est **pas sensible à la casse.**
Si vous vous trompez, c'est vraiment que vous l'avez cherché !

```
Saisissez l'une des commandes suivantes :
        - 's' pour afficher les statistiques de la simulation
        - 'h' pour afficher l'historique des evenements
        - 'p' pour afficher les horaires de passage d'un patient dans les differents services
        - 'q' pour mettre fin a la simulation
r
Saisie incorrecte. Veuillez reessayer :
Q
Etes-vous sur(e) de vouloir quitter la simulation ? (o/n) :
```

```
Saisissez l'une des commandes suivantes :
        - 's' pour afficher les statistiques de la simulation
        - 'h' pour afficher l'historique des evenements
        - 'p' pour afficher les horaires de passage d'un patient dans les differents services
        - 'q' pour mettre fin a la simulation
r
Saisie incorrecte. Veuillez reessayer :
Q
```

1.3.b

1.3.c – <u>Utilisation des commandes :</u>

Menu des options :

Pour afficher le **menu des options,** refusez l'utilisation des paramètres par défaut au lancement de la simulation.

```
Souhaitez-vous utiliser les parametres par defaut ? (o/n) :
N
```

Vous obtiendrez le menu suivant :

```
MENU DES OPTIONS :
_____________________________________________________________________________________
Saisissez l'une des commandes suivantes :
        - 't' pour modifier les parametres temporels de la simulation
        - 'p' pour modifier le mode de generation des patients
        - 'd' pour modifier le mode de generation des durees des taches
        - 'c' pour modifier le mode de determination des circuits empruntes par les patients
        - 'b' pour activer ou desactiver les tonalites sonores
        - 's' pour lancer la simulation
```

- La commande '*t*' permet de **fixer ou non le mois ou le jour d'arrivée** des patients.
- La commande '*p*' permet de **créer manuellement ou automatiquement les patients.**
- La commande '*d*' permet de **générer automatiquement les durées des tâches**, ou de **les choisir vous-même** à chaque étape du flux de patients.
- La commande '*c*' vous permet de choisir entre **détermination automatique** et **manuelle** du **circuit emprunté** par chaque patient.
- La commande '*b*' vous permet de désactiver la tonalité sonore qui a vite fait de devenir casse-bonbons.

Vous pouvez modifier à volonté les paramètres de la simulation et revenir sur vos choix si vous le souhaitez.
Utilisez la commande '*s*' pour valider vos options.

Menu principal de fin de simulation :
Une fois le flux de patients terminés, vous aurez accès à ce menu :

```
DONNEES DE LA SIMULATION
_____________________________________________________________________________
Saisissez l'une des commandes suivantes :
        - 's' pour afficher les statistiques de la simulation
        - 'h' pour afficher l'historique des evenements
        - 'p' pour afficher les horaires de passage d'un patient dans les differents services
        - 'q' pour mettre fin a la simulation
```

- '*h*' affiche un **historique complet** des événements de la simulation.

- '*p*' affiche l'**historique du patient unique** de votre choix.

- '*q*' **met fin à la simulation** et vous invite à en redémarrer une nouvelle, ou à quitter le programme.

```
Voulez-vous commencer une nouvelle simulation ? (o/n) :
n
```

- '*s*' vous donne accès à un **menu secondaire** comportant de **nombreuses commandes de traitement de données :**

```
STATISTIQUES :
_____________________________________________________________________________
Saisissez l'une des commandes suivantes :
        - 'm' pour retourner au menu principal.
        - 'a' pour afficher l'ordre d'arrivee et de sortie des patients ainsi que le temps passe aux urgences.
        - 's' pour afficher le temps d'attente moyen pour chaque salle d'attente
        - 'o' pour afficher l'ordre de passage des patients dans chaque service
        - 't' pour calculer le temps total, moyen et median passe aux urgences
        - 'p' pour afficher le pourcentage et la liste de patients ayant emprunte chaque circuit
        - 'd' pour afficher des statistiques relatives a la duree de chaque tache
```

- 'a' affiche **tous les patients** de la simulation par **ordre d'arrivée** au service d'urgences, puis **par ordre sortie** avec leur **temps de séjour à l'hôpital**. Précise si l'ordre de sortie et d'arrivée sont différents ou non.

- '*o*' affiche **tous les patients passés par chaque salle** autre que les salles d'attente **dans l'ordre** où ils y sont entrés.

- 's' affiche **le temps d'attente moyen** dans chaque **salle d'attente.**
  Selon le nombre de patients, la répartition de leurs heures d'arrivée, et le circuit dans lequel elles se trouvent, les salles d'attente peuvent avoir un temps d'attente gigantesque ou insignifiant, voire nul.

- 't' affiche le **temps total**, **moyen** et **médian** passé aux urgences. Il calcule également l'**écart-type** des durées de séjour à l'hôpital en minutes.
  Montre également les **durées minimales et maximales de séjour** et propose à l'utilisateur d'**afficher celles de tous les patients.**

- 'd' affiche, **pour chaque salle** autre que les salles d'attente, les **durées totales**, **moyennes**, **médianes**, **minimales** et **maximale** d'utilisation, la **liste des patients entrés** par ordre croissant de durée de séjour et l'**écart-type des durées de séjour** dans chaque salle.

- 'm' permet **de retourner au menu principal.**

L'affichage des résultats peut prendre beaucoup de place, et les menus se réaffichent juste après.
Il faut donc parfois remonter dans la console jusqu'à la commande saisie pour consulter ses résultats.

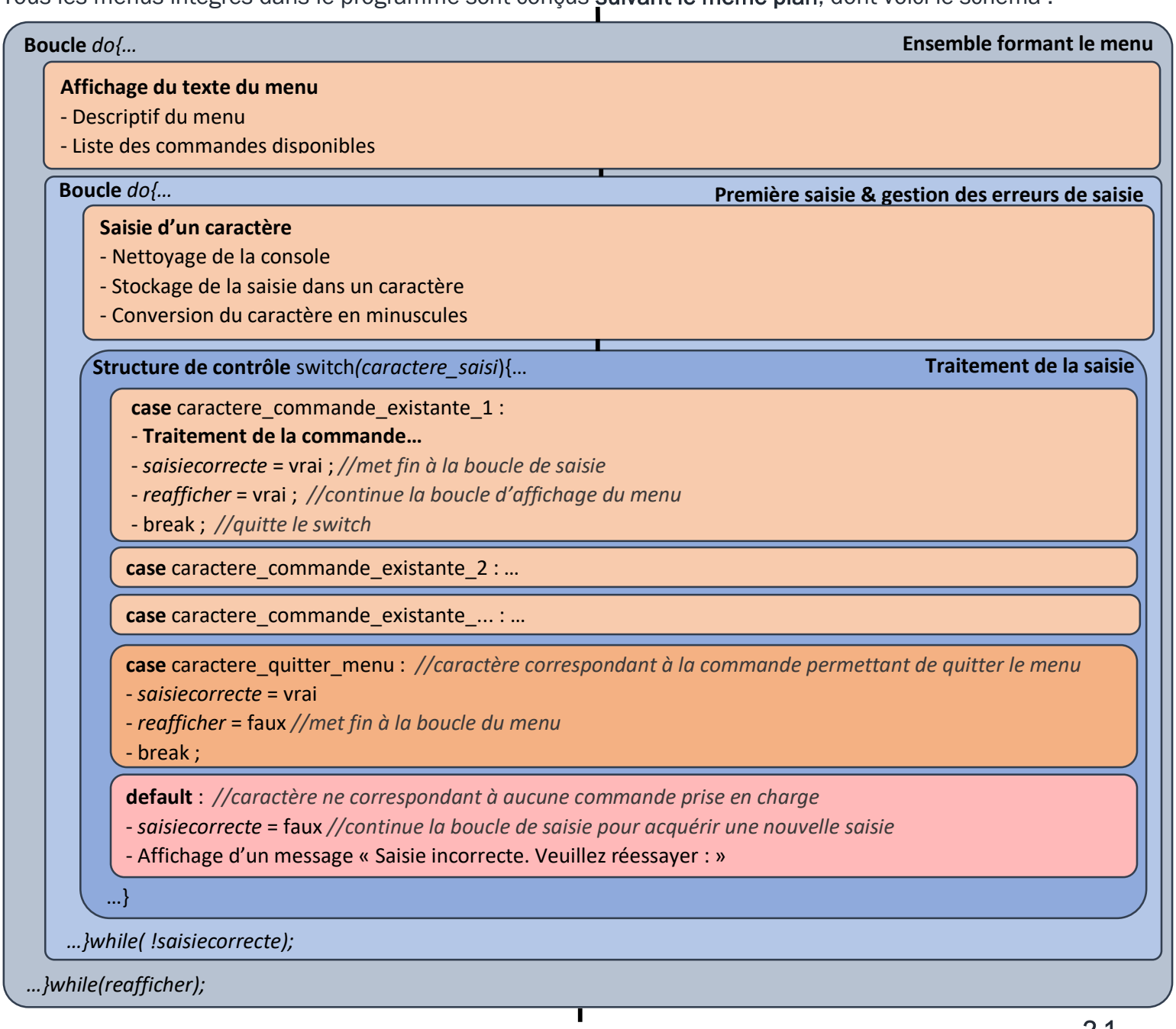
Ces menus s'affichent de nouveau après chaque commande tant que vous n'utilisez pas la commande de sortie.

II – Analyse des algorithmes

Fonctionnement des principaux algorithmes utilisés

2.1 - Fonctionnement des menus

Tous les menus intégrés dans le programme sont conçus **suivant le même plan**, dont voici le schéma :



2.1

2.2 - Gestion du temps

La gestion du temps occupe une place importante dans le programme : la **date et l'heure d'arrivée** des patients, la **durée** des différentes tâches accomplies, l'écoulement **temps virtuel** à l'intérieur de la simulation, le calcul du **temps d'attente moyen** dans chaque salle d'attente … sont autant d'éléments qui nécessitent d'avoir une **structure représentant des valeurs temporelles** et d'être capables **de saisir, comparer et traiter** des éléments de ce type.

Nous avons fait le choix de concevoir **notre propre système de gestion du temps**, adapté à nos besoins, plutôt que d'utiliser la biliothèque native <*time.h*>.

<u>Structure :</u>

Le fichier header *''temps.h''* définit une **structure de type** *temps* décomposant le temps en **5 unités** :

Elle possède le champ *année* : entier non signé

Et les champs *mois*, *jour*, *h*, *m* : caractères non signés (car valeurs extrêmes >= 0 et <255.)

Il sera fait **abstraction des secondes** pour simplifier la saisie et l'affichage du temps – excepté lors du calcul des **statistiques** où les **secondes deviennent significatives** (*exemple : temps moyen entre 2 et 3 minutes : 2min30sec*).
Dans ce cas, on **convertira** le temps concerné en un nombre de minutes **réel** pour effectuer différents calculs (divisions par exemple) – à l'issue desquels, les chiffres après la virgule représenteront un nombre de secondes exprimées en minutes, que l'on affichera si cela est pertinent.
(*exemple : 2min + 3min = 5min. 5/2 = 2,5 min → 2 min et 60*(2,5-2) sec = 2min et 30sec*).

Fonctionnalités :

Saisie :

Nous avons défini plusieurs **fonctions de saisie de valeurs temporelles** dans la source *temps.c*.
Elles fonctionnent toutes selon le même principe.
Exemple avec le fonctionnement de la **saisie d'une heure** avec *temps demander_heure()* :

- Dans un premier temps, on réalise la saisie **sécurisée** d'une **chaîne de caractères**.

- On effectue ensuite des **tests** sur la chaîne de caractères pour vérifier qu'elle contient une heure ou une date écrite sous forme de **l'un des formats acceptés** : par exemple, pour une heure, on essayera de convertir directement la chaîne en un entier à l'aide de la fonction *atoi(const char *str)* pour récupérer une heure aux *formats " hh "* ou *" h "*.
Si cela n'est pas possible, nous essayons de **localiser le caractère** ':' dans la chaîne à l'aide de la fonction *strchr(const char *str)* de *<string.h>*, et regardons si les caractères avant et après le ':' sont des chiffres – en distinguant les cas *" hh:mm "*,*" h:mm "*...

- On stocke les données acquises dans des **variables temporaires** *heures* et *minutes*.

- On réalise des tests sur ces variables pour **vérifier que les valeurs acquises sont valides** :
heures doit être compris entre 0 et 23, *minutes* entre 0 et 59 ...

- Si toutes ces étapes sont validées avec succès, on créée une variable *t* de type *(temps)*.
Ici, les champs *t.annee, t.mois* et *t.jour* sont initialisés à 0 car l'utilisateur ne saisit que l'heure et les minutes.
t.h prend la valeur d'*heures* et *t.m* la valeur de *minutes*. **On retourne *t*.**
Si une de ces étapes n'a pas été validée, on recommence la saisie en indiquant si possible le problème détecté à l'utilisateur.

Pour la saisie d'une date jj/mm/a..., le fonctionnement est analogue : on cherchera le premier caractère '/' puis le deuxième caractère '/', deux chiffres avant et entre chaque '/' et tous les chiffres après le deuxième '/' pour l'année.

Tests :

Pour mener à bien les saisies et les opérations d'addition et de soustraction de valeurs temporelles, il faut être en mesure d'effectuer certains **tests** sur ces dernières.

- **Test de nullité** : *bool estNul(temps t)* renverra **1 (*vrai*)** si tous les champs du *temps t* valent 0, et **0 (*faux*)** sinon.

On aura notamment besoin de cette fonction pour additionner deux temps *t1* et *t2* en ajoutant une minute à t1 et en enlevant une minute à t2 **tant que t2 n'est pas nul** – à l'issue de cette opération, t1 contiendra la somme des deux temps.

- **Test de bissextilité d'une année** : *bool estBissextile(unsigned short int annee)* renverra **1 (*vrai*)** si une année est divisible par 400 ou par 4 et non par 100, et **0 (*faux*)** dans tous les autres cas.

On aura besoin de cette fonction pour déterminer la **validité d'une date** : en effet, le 29 février est un jour valide seulement si l'année est bissextile.

- **Tests de validité / valeurs maximales :** On utilisera par exemple la fonction *bool jmSontCorrects(temps t)* pour vérifier si une date saisie est valide.

Elle initialise deux **booléens** *jour_correct* et *mois_correct* à *vrai,* et effectuera une **série de tests** pour détecter une éventuelle invalidité du **jour** ou du **mois**. Si le mois n'est pas compris entre 1 et 12, il est considéré **invalide.**

Pour le jour, il faudra distinguer plusieurs cas : tout d'abord, si le jour est égal à 0 (il ne peut pas être inférieur à 0 car il est de type *caractère non signé*), il est considéré **invalide.**

Si *t.mois == 2* , le jour est invalide s'il dépasse 28 et si *!estBissextile(t.annee)*, ou s'il dépasse 29 et si *estBissextile(t.annee)*.

Pour tous les autres mois, on détermine si le jour choisi est correct connaissant le **nombre de jour maximal** (30 ou 31) **en fonction de la parité de chaque mois** (de mars à juillet et en janvier, les mois impairs ont 31 jours et les mois pairs 30. D'août à décembre, c'est le contraire).

A la fin des tests, la fonction retourne **l'intersection des deux booléens** (*jour_correct && mois_correct*).

Cette fonction est notamment utilisée pour **vérifier la validité de la saisie d'une date.** On l'utilise également pour les opérations d'incrémentation et de décrémentation du temps (on considère qu'on incrémente un temps en lui ajoutant une minute.)

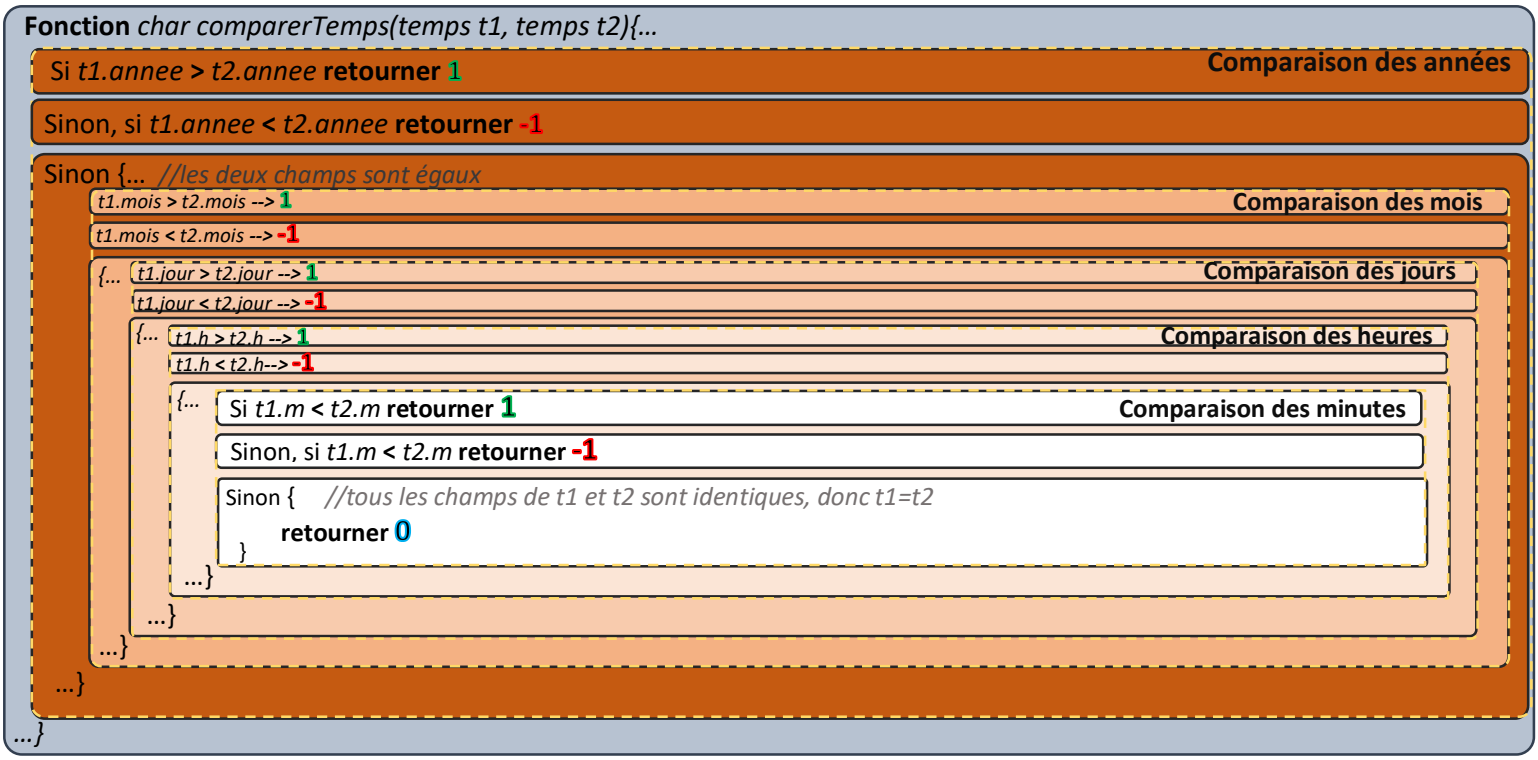
En effet, si le champ *m* d'un temps a atteint la valeur 59, et son champ *h* n'est pas encore à *23*, ajouter une minute nécessite d'**incrémenter** le champ *h* et de **remettre le champ** *m* **à 0**.

De même, si le jour, les heures et les minutes ont atteint leur valeur maximale, mais pas le mois, on incrémentera le compteur de mois, on remettra le compteur de jours à 1 et ceux des heures et des minutes à 0 : (exemple : 23 :59 le 31/10/2020 + 1minute -> 00 :00 le 01/11/2020).

Les tests de valeurs maximales sont donc particulièrement **importants pour une bonne gestion du temps.**

Comparaison : A plusieurs reprises, nous avons besoin d'une fonction permettant d'indiquer si un *temps t1* est « supérieur », « inférieur » ou « égal » à un *temps t2*.
Par exemple, pour libérer un patient d'une salle, il faut que son instant de libération soit égal ou inférieur au temps de la simulation.

On utilisera pour ce faire la fonction *char comparerTemps(temps t1, temps t2)* : elle compare **deux à deux** les champs de *t1* et *t2* (en allant **des plus grandes unités vers les plus petites**).
La comparaison s'arrête dès que les valeurs de *t1* et *t2* **pour un même champ sont différentes** : on retourne alors **1** si c'est le champ de *t1* qui est le plus grand, **-1** si c'est le plus petit.
Si aucune différence n'est détectée, càd. si tous les champs de *t1* et *t2* sont **égaux**, on retourne ***0***.



Incrémentation : On définit la procédure *void incrementerTemps(temps *t1)* permettant d'**ajouter une minute** à un élément de type temps dont on connaît **l'adresse en mémoire.**

Le fonctionnement d'une telle procédure est plus complexe qu'il n'y paraît : en effet, **pour que le temps reste exprimé suivant un format valide,** il faut tenir compte des **limites maximales** et **minimales de chaque unité.** Ainsi, on utilisera, avant d'incrémenter **t1*, des **fonctions de test** retournant un booléen indiquant pour chaque champ s'il a atteint la valeur maximale autorisée : par exemple, pour regarder si un jour est le dernier du mois, on appellera *estDernierJourMois(*t) ;* on procèdera de la même manière pour tous les champs **exceptée l'année.**

On teste les valeurs maximales **de la plus petite unité** (minute) **à la plus grande** (mois, car ne considère pas de maximum pour les années). Par exemple :

Si *t->m* == 59, et *t->h* < 23, on incrémente *t->h* et on remet *t->m* à sa valeur minimale, 0.

Sinon, on incrémente le compteur de minutes *t->m*.

En procédant ainsi en considérant toutes les unités utilisées dans la structure *temps*, on pourra **passer par incrémentation du 31/12/2020 à 23:59 au 01/01/2021 à 00:00.**

Dans la pratique, nous imbriquons des conditions if (*valeurMax(t->unité))* … avec l'unité allant de mois à minute, en gérant dans le bloc d'instruction du dernier *if* le **cas particulier** où on doit remettre tous les champs à leur valeur minimale sauf année.

A chaque bloc *if*, nous associons un bloc *else* correspondant au **cas général** où il faut incrémenter le champ correspondant à l'unité choisie et remettre au minimum autorisé les champs correspondant à de plus petites unités.

L'incrémentation est notamment utilisée pour **faire avancer le temps virtuel de la simulation** afin de faire évoluer le flux de patients.

On définit également une fonction pour **décrémenter le temps**, selon le même principe mais en considérant cette fois comme cas particulier le cas où l'unité a atteint sa **valeur minimale** – on décrémente alors le compteur de l'unité de rang supérieur et on remet à leur **valeur maximale** les unités de rang inférieur.

L'incrémentation, la décrémentation et le test de nullité peuvent alors être **utilisées pour faire la somme et la différence de valeurs temporelles.**

Somme : On utilise la procédure *void ajouterTemps(temps *t1,temps t2)* pour **ajouter** un temps *t2* à un temps *t1* d'adresse en mémoire connue.

L'algorithme utilisé se base sur des fonctions vues précédemment :

t2 est copiée dans une variable *temps reste* ;

On entre dans une boucle indéterminée appelant à chaque itération le **test de nullité** :
« Tant que(*estNul(reste) est faux) … ».* A chaque itération, on **décrémente reste** et on **incrémente **t1*.**

Une fois *reste* devenu un « temps nul » à force de décrémentation, on sort de la boucle : **t1* vaut alors sa valeur initiale additionnée de *t2*.

Cette procédure est notamment utilisée pour **ajouter la durée d'une tâche au temps virtuel de la simulation pour obtenir l'heure de libération d'une salle,** et pour **sommer des durées** afin d'établir des statistiques d'utilisation des salles.

Différence : On procède d'une manière similaire pour **calculer la différence entre deux instants.** La fonction *temps diffTemps(temps t1,temps t2)* renvoie un temps correspondant à la **distance entre le plus grand et le plus petit des temps *t1* et *t2*.**

Il faut pour cela d'abord utiliser la **fonction de comparaison** pour comparer *t1* et *t2* : on stocke le plus petit temps dans une variable *ppt* et le plus grand dans une variable *pgt*.

On crée ensuite une variable *sortie* de type temps initialisée avec tous ses champs nuls.

Tant que *ppt* < *pgt*, on **incrémente** *ppt* et *sortie* à l'aide de la procédure *incrementerTemps*. A l'issue de la boucle, *sortie* vaut la valeur absolue de la différence entre les deux instants. On retourne alors *sortie*.

Cette procédure est notamment utilisée pour **calculer des plages temporelles** de la forme : *instant_fin - instant_début + 1*, une fonctionnalité requise pour la **génération aléatoire des heures d'arrivée** des patients.

2.3 - Génération des patients

Type *patient* :

id (unsigned int) *// Selon le rang*

nom (char[40])

prenom (char[40])

circuit (unsigned char) *// 3 val. possibles*

arrivee (temps)

sortie (temps) *// attrib. en fin de parcours*

duree_examen_complementaire (temps)

NB : Dans la programme, on manipulera principalement des ***pointeurs*** *vers des patients plutôt que des patients directement.*

Représentation des patients :

Les patients sont représentés par un type *patient*, une structure contenant les champs suivants :

- Un **numéro** *id* (*entier non signé*) **le désignant**. Deux patients ne peuvent pas avoir le même identifiant. L'identifiant **correspond à l'ordre d'arrivée des patients** au service des urgences, il leur est attribué avant le début de la gestion du flux.

- Un **nom** *nom* et un **prénom** *prenom* (*tableaux de 40 caractères*).

- Un **numéro de circuit** *circuit* (*caractère non signé*) désignant le circuit emprunté par le patient.

- Un **instant d'arrivée** *arrivee* et un **instant de sortie** *sortie* (*temps*). *arrivee* est déterminé avant le début de la gestion du flux, et influe sur son déroulement. *sortie* est déterminée par son déroulement.

- Une **durée d'examen complémentaire** *duree_examen_complentaire* (*temps*). Elle est déterminée soit avant la gestion du flux (en mode choix manuel des durées d'examen), soit durant celle-ci (en mode génération automatique des durées d'examen).

La consigne du sujet stipulait en effet qu'il fallait demander la durée d'examen complémentaire de chaque patient au début de la simulation – l'associer directement à un patient semblait donc nécessaire pour le mode « génération manuelles des durées »

Les champs à déterminer à la création des patients sont donc (obligatoirement) le **nom**, le **prénom**, et **l'instant d'arrivée**. (Avec certaines options, il faut ajouter le **numéro de circuit** et la **durée d'examen complémentaire**)

Le **nombre total de patients créés** est compté par un entier ***nombre_patients*** (initialisé à 0).

L'adresse mémoire de tous les patients de la simulation est répertoriée dans un même tableau ***liste_patients***.
On considère qu'on lui a initialement **alloué dynamiquement** un espace mémoire de taille *(1*sizeof(patient*))*.

On pourra utiliser une fonction *patient* creerPatient()* pour instancier un nouveau patient en réservant un espace mémoire de taille *(sizeof(patient))* et en renvoyant un pointeur vers le début de cet espace.

2.3.a - Génération manuelle des patients :

Les informations requises sont saisies par l'utilisateur pour chaque patient.
Si l'option « génération manuelle des patients » est choisie, on entre dans le bloc permettant la génération manuelle d'un nombre indéterminé de patients.
Si un paramètre temporel est fixé, on demande à l'utilisateur de le saisir (*ex : jour fixé --> saisir la date complète*).
Ensuite, **une première fois**, et **tant que l'utilisateur veut ajouter un nouveau patient** :

1 - On **réalloue dynamiquement** un espace mémoire de taille *(**nombre_patients***sizeof(patient*)) au tableau liste_patients.*

2 - On instancie un nouveau patient avec *creerPatient()* et on récupère son adresse dans une variable *patient_temporaire*.

3 - On demande ensuite à l'utilisateur de **saisir le prénom**, puis **le nom** du nouveau patient.
Sa saisie est enregistrée dans les champs *patient_temporaire -> prenom* et *patient_temporaire -> nom*.

4 - On invite l'utilisateur à saisir les informations manquantes pour compléter **l'instant d'arrivée** du patient : ce peut être l'heure si le jour est fixé, le jour et l'heure si le mois est fixé, ou encore la date complète et l'heure. Sa saisie est enregistrée dans le champs *patient_temporaire -> arrivee.*

- Si l'option « *circuit choisi par l'utilisateur* » est activée, on réalise également la saisie d'un caractère 'c' ou 'l' correspondant respectivement aux circuits « *court* » ou « *long* ».
Si l'utilisateur a choisi le circuit long, on lui demande s'il veut ou nom que le patient subisse un examen complémentaire.

- Si l'option « *durées choisies par l'utilisateur* » est activée, on invitera l'utilisateur à saisir la durée de l'examen complémentaire du nouveau patient. En mode « *circuit choisi par l'utilisateur* », on ne demande cette information que si le patient a emprunté le circuit long avec examen complémentaire.

5 - On **insère** la valeur de *patient_temporaire* (l'adresse mémoire du nouveau patient) **à la position *nombre_patients* du tableau *liste_patients*.**

6 - On **incrémente** *nombre_patients* pour qu'il corresponde au nouveau nombre total de patients.

7 - On demande à l'utilisateur s'il veut ajouter ou non un nouveau patient à la simulation. S'il répond oui, la génération des patients se poursuit et retourne à l'étape 1. Sinon, la génération des patients est finie.

2.3.b - Génération automatique des patients :

L'utilisateur définit une plage horaire d'arrivée et indique un nombre de patients à générer.
Sur la base de ces données, les patients sont tous instanciés et se voient attribuer une heure d'arrivée.
Ils sont ensuite nommés à l'aide de noms et prénoms contenus dans un fichier texte.

Génération des heures d'arrivée :

Référence :
http://www.ressources-actuarielles.net/EXT/ISFA/fp-isfa.nsf/0/1430ad6748ce3affc1256f130067b88e/$FILE/Seance2.pdf
Ce document nous a permis de comprendre comment utiliser le processus de poisson pour modéliser les heures d'arrivée d'une file d'attente.
(Nous avons cependant mis au point notre propre technique de génération des heures d'arrivée des patients sans avoir recours à des exemples ou à une aide extérieure).

L'heure d'arrivée des patients est générée à l'aide d'un **processus poissonnien** :

Notons ***d*** l'étendue non nulle, en minutes, de la plage horaire considérée.
Notons ***nb*** le nombre de patients simulés.
Considérons l'événement : « *un patient arrive au service d'urgences à l'instant t* ». Notons **λ** la probabilité d'occurrence de l'événement. En supposant qu'un patient a la même chance d'arriver à chaque instant (pas d'heures de pointe), on a : $\boldsymbol{\lambda = nb / d}$

Notons ***N(t)*** le nombre de patients arrivés à un instant **t**. D'après la loi de Poisson, la probabilité ***P*** qu'un nombre ***k*** de patients soient arrivés à un instant **t** est donnée par :

$$\mathbf{P} = \mathbb{P}[\mathbf{N(t) = k}] = e^{-\lambda t} \times \frac{(\boldsymbol{\lambda t})^{k}}{\boldsymbol{k!}}$$

La technique que nous avons mise en place pour générer à l'aide de cette loi de probabilité l'heure d'arrivée d'un patient de rang d'arrivée **k** (les rangs allant de 1 à ***nb***), consiste à calculer **P** pour chaque instant de la plage horaire (donc pour ***t*** allant de 0 à ***d***, l'écart de la plage horaire).

On retiendra alors, pour chaque rang d'arrivée ***k***, l'instant ***tmax*** pour lequel on a obtenu la plus grande valeur de **P. tmax** correspond alors à la minute de la plage horaire pendant laquelle il est le plus probable qu'un ***k***-ième *patient* arrive à l'hôpital.
Enfin, en ajoutant ***tmax*** minutes à l'instant de début de la plage horaire, on obtient *instant_arrivee,* la date et l'heure la plus probable d'arrivée de ce patient.

Instanciation du patient :

On enregistrera alors cet instant dans une variable *instant_arrivee* de type *temps*.
On pourra ensuite instancier un patient **patient_temporaire* à l'aide de *creerPatient()* et affecter à son champ *patient_temporaire -> arrivee* la valeur d'*instant_arrivee*.
On lui attribuera également l'id ***k***.
On pourra ensuite incrémenter ***k*** pour générer l'heure d'arrivée du prochain patient.

Dans la pratique, le fonctionnement de l'algorithme utilisé est légèrement plus complexe.

- Pour éviter une répartition trop régulière et identique à chaque simulation prenant les mêmes paramètres, nous avons choisi d'ajouter une **composante pseudo-aléatoire** (basée sur l'heure de lancement de l'application) au calcul de ***P***.

Suivant des critères arbitraires, et après de nombreux tests, nous avons décidé de générer à chaque itération de la boucle parcourant la plage horaire, un nombre aléatoire ***aleat*** dépendant du nombre de patients et de l'écart de la plage horaire. La probabilité d'occurrence ***λ*** est recalculée à chaque fois comme **aleat + *nb / d.***
La valeur d'***aleat*** influence alors indirectement le calcul de ***P*** – auquel on ajoute en plus, directement, ***aleat/5***.

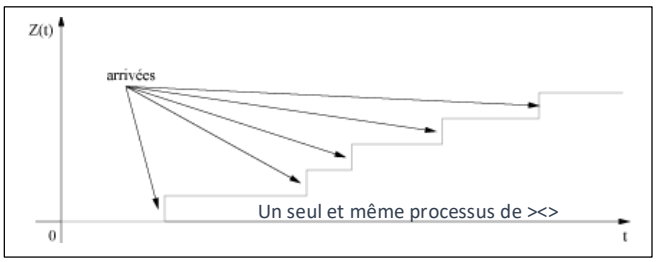
L'influence d'***aleat*** sur le calcul de ***P*** est relativement légère, mais **suffisante pour offrir une chance d'élire un autre instant d'arrivée que celui pour lequel, sans utiliser de composante aléatoire, on aurait obtenu la plus grande valeur de *P***. En procédant ainsi, nous **modifions aléatoirement les écarts entre les instants d'arrivée** des patients, ce qui permet d'avoir des patients d'arriver à des heures très proches (voire égales) ou très éloignées par rapport à l'écart total de la plage horaire.

- Il est possible de forcer l'arrivée d'un patient à la première minute de la plage horaire en prenant ***k*** allant de [0 à ***nb-1***] au lieu de [1 à ***nb***]. Cette méthode permet une utilisation plus intuitive de l'application. Cependant, nous avons choisi de ne pas la mettre en œuvre car elle fausse le calcul de probabilités et augmente la similitude entre deux simulations démarrées avec les mêmes paramètres.

- La technique présente un grand inconvénient : la formule permettant de calculer ***P*** fait intervenir un calcul factoriel. Or, la croissance de la fonction factorielle est telle que ***22*** est le dernier entier dont le langage C peut calculer précisément le factoriel : la manipulation de nombres aussi grands nécessite l'utilisation du type *double*, qui se révèle très imprécis à partir de ***23!*** (du moins, c'est ce que nous avons remarqué en utilisant notre propre fonction de calcul *double factorielle(unsigned int n)*).

Nous avons donc décidé de d'appliquer le processus de Poisson décrit précédemment à des **paquets de 22 patients au maximum** et à des plages horaires correspondantes.
Notre algorithme construit donc des paquets contenant 22 patients (ou moins s'il en reste moins à traiter) et divise la plage horaire principale en sous-plages horaires d'écart (***nb_patients_paquet/nb***)*(***d***).
Il calcule le nombre ***aleat*** en fonction du nombre de patients restant et lui donne moins de poids pour favoriser une meilleure répartition des arrivées pour un nombre de patients inférieur à 22.

Une telle méthode **fausse** certes **la distribution des instants d'arrivée suivant la loi de Poisson** (puisqu'on répèterait alors potentiellement plusieurs fois le même patron de répartition plutôt que d'obtenir une répartition basée sur le nombre total d'individus arrivant au cours de la plage horaire totale),



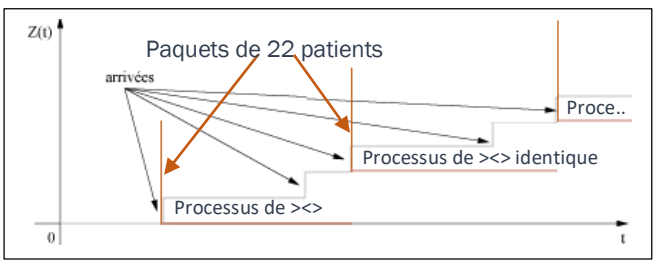


Illustration du problème – *fig.1 : comportement attendu ; fig.2 : la distribution des heures d'arrivée répète le même patron.*

mais **le résultat obtenu reste réaliste**, notamment grâce au concours de la composante aléatoire.
En effet, ***aleat*** influence aléatoirement la distribution obtenue – d'autant plus **que son poids diminue au fur et à mesure** que les paquets de patients sont traités, ce qui permet d'avoir à chaque fois un patron de répartition finalement significativement différent des précédents.

Ainsi, la génération de 20 patients sur une plage horaire de 2 heures donnera un résultat tout aussi satisfaisant pour l'utilisateur que la génération de 200 patients sur une plage horaire de 10 heures.

Voici deux exemples d'utilisation de l'algorithme décrit :

Génération de 30 patients sur une plage horaire 10:00 – 14:00 :

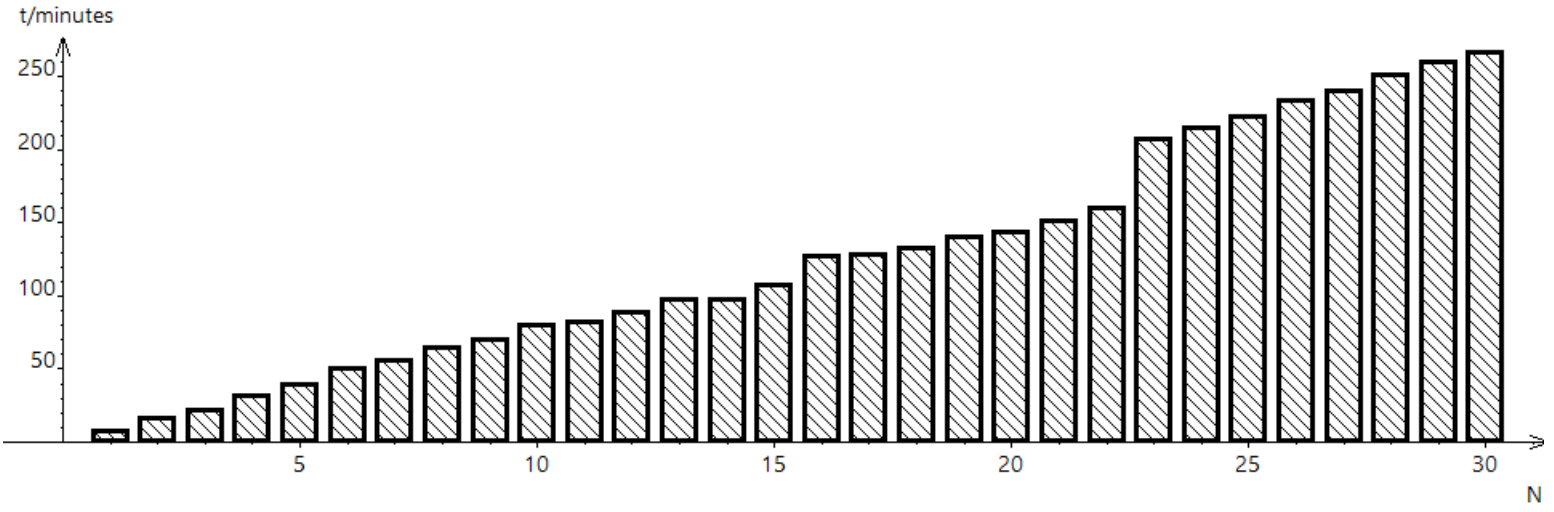
```
Date d'arrivee des patients :
        Entrez la date au format jj/mm/aaaa :  21/12/2020
Heure de debut de la plage horaire d'arrivee des patients :
        Entrez l'heure au format hh:mm :  10
Fin de la plage horaire d'arrivee des patients :
        Entrez l'heure au format hh:mm :  14:30

Nombre de patients a simuler (recommande : 10 < nb < 88) : 30
```

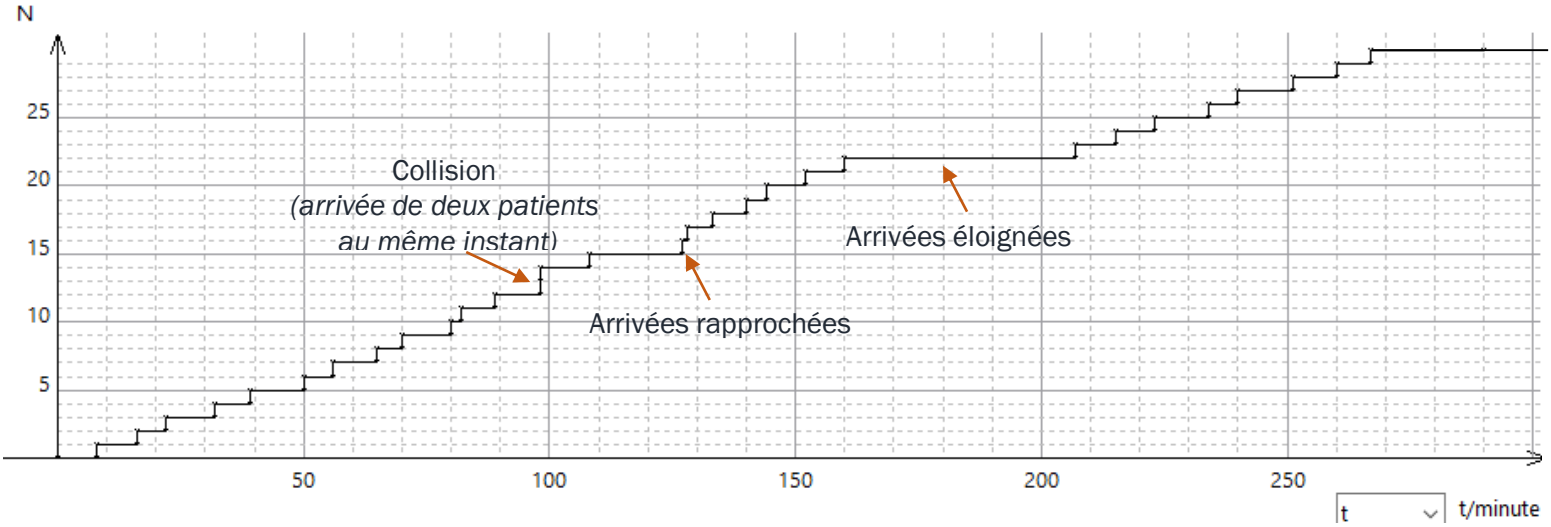
Affichage de l'ordre d'arrivée (commande '*s*' + '*a*') :

```
            Le 21/12/2020 :
         1 - 10:08 : Patient   1 (Monique Leroux)
         2 - 10:16 : Patient   2 (Maria Rubio)
         3 - 10:22 : Patient   3 (Homer Simpson)
         4 - 10:32 : Patient   4 (Gerard Menvussat)
         5 - 10:39 : Patient   5 (Besse Cooper)
         6 - 10:50 : Patient   6 (Koto Okubu)
         7 - 10:56 : Patient   7 (Hester Ford)
         8 - 11:05 : Patient   8 (Jacques Ouzy)
         9 - 11:10 : Patient   9 (Meredith Saperson)
        10 - 11:20 : Patient  10 (Adelina Domingues)
        11 - 11:22 : Patient  11 (Anita Bath)
        12 - 11:29 : Patient  12 (Andrew Chalmers)
        13 - 11:38 : Patient  13 (Aubin Didon)
        14 - 11:38 : Patient  14 (Neva Morris)
        15 - 11:48 : Patient  15 (Akim Embett)
        16 - 12:07 : Patient  16 (Elizabeth Adin)
        17 - 12:08 : Patient  17 (Patt Tachon)
        18 - 12:13 : Patient  18 (Steve Jobs)
        19 - 12:20 : Patient  19 (Deborah Francois)
        20 - 12:24 : Patient  20 (David Beyssac)
        21 - 12:32 : Patient  21 (Marin Daudousse)
        22 - 12:40 : Patient  22 (Emma Carennat)
        23 - 13:27 : Patient  23 (Maria Gomes)
        24 - 13:35 : Patient  24 (Greta Thunberg)
        25 - 13:43 : Patient  25 (Genevieve Letonnelier)
        26 - 13:54 : Patient  26 (Marie-Louise Meille)
        27 - 14:00 : Patient  27 (Patrick-Odile Gillet)
        28 - 14:11 : Patient  28 (Christian Mortensen)
        29 - 14:20 : Patient  29 (Francisca Celsa)
        30 - 14:27 : Patient  30 (Omer Dilpleux)
```

Minutes écoulées depuis le début de la plage pour chaque nombre de patients arrivés :



Courbe en escalier représentative du processus de Poisson influencé par la composante aléatoire :



On observe une répartition **régulière dans l'ensemble,** avec quelques **disparités par instants** (arrivée simultanée, très rapprochée ou très éloignée de deux patients).

Détermination automatique du nom et du prénom de chaque patient :
Une fois tous les patients générés et leurs adresses insérées dans un même tableau, une procédure *nommerPatients* est appelée pour attribuer à chacun d'entre eux un prénom et un nom.

1 - Elle réalise dans un premier temps **l'ouverture du fichier** « *liste_noms.txt* » en mode lecture. Si cette étape échoue, tous les patients seront nommés « John Doe ».

2 - Elle lit ensuite, dans une variable ***nb_noms_dispo*** le nombre total de combinaisons nom+prenom, inscrit au début du fichier. Un tableau d'entiers de cette taille, ***tab_noms_dispo***, est alors créé, puis rempli par de numéros allant de 1 à ***nb_noms_dispo***.
tab_noms_dispo sera utilisé pour éviter l'attribution de noms déjà utilisés (tant que la liste de noms n'est pas épuisée).
En effet, chaque combinaison nom+prénom est associée à un numéro écrit en début de ligne dans le fichier.
Chaque fois que l'on choisira une ligne numérotée *id*, ***tab_noms_dispo[id-1]*** (valant initialement *id*) sera remplacé par *0*. On ne pourra alors plus choisir une ligne pour laquelle *tab_noms_dispo[id-1]* **== 0.** (*voir schéma explicatif p 17*).

3 - Pour choisir une ligne du fichier au hasard, on calcule un nombre pseudo aléatoire ***indice_random*** compris entre 0 et *nb_noms_dispo*-1.
On entre ensuite dans un bloc d'instructions qui parcourt le tableau ***tab_noms_dispo*** à partir d'*indice_random*, vers les cases d'indice supérieur. Dès que la valeur d'une case du tableau est différente de 0, on cesse la recherche, et on assigne à un entier ***id_nom_choisi*** son contenu avant de le **remplacer par 0** pour rendre l'*id* inutilisable par d'autres patients.
Si toutes les cases du tableau contiennent 0, on **balaye le tableau dans l'autre sens** (à partir d'*indice_random*, vers les indices inférieurs).
Si encore une fois, aucun *id* disponible n'a pu être trouvé, **la liste est complètement épuisée.**
On remplit alors à nouveau *tab_noms_dispo* et on incrémente une variable ***num_doublon*** (initialisée à 0), qui sera, à partir de 1, ajoutée aux noms donnés pour **différencier les doublons**.

4 - Une fois un *id* choisi, on parcourt le fichier jusqu'à rencontrer le même numéro au début d'une ligne . **On lit** alors **le prénom et le nom qui suivent** et on les copie dans les champs ***nom*** et ***prenom*** d'un *patient de *liste_patients*. On recommence à partir de l'étape 2 pour chaque autre patient de *liste_patient*.
Une fois tous les patients nommés, on **ferme le fichier.**

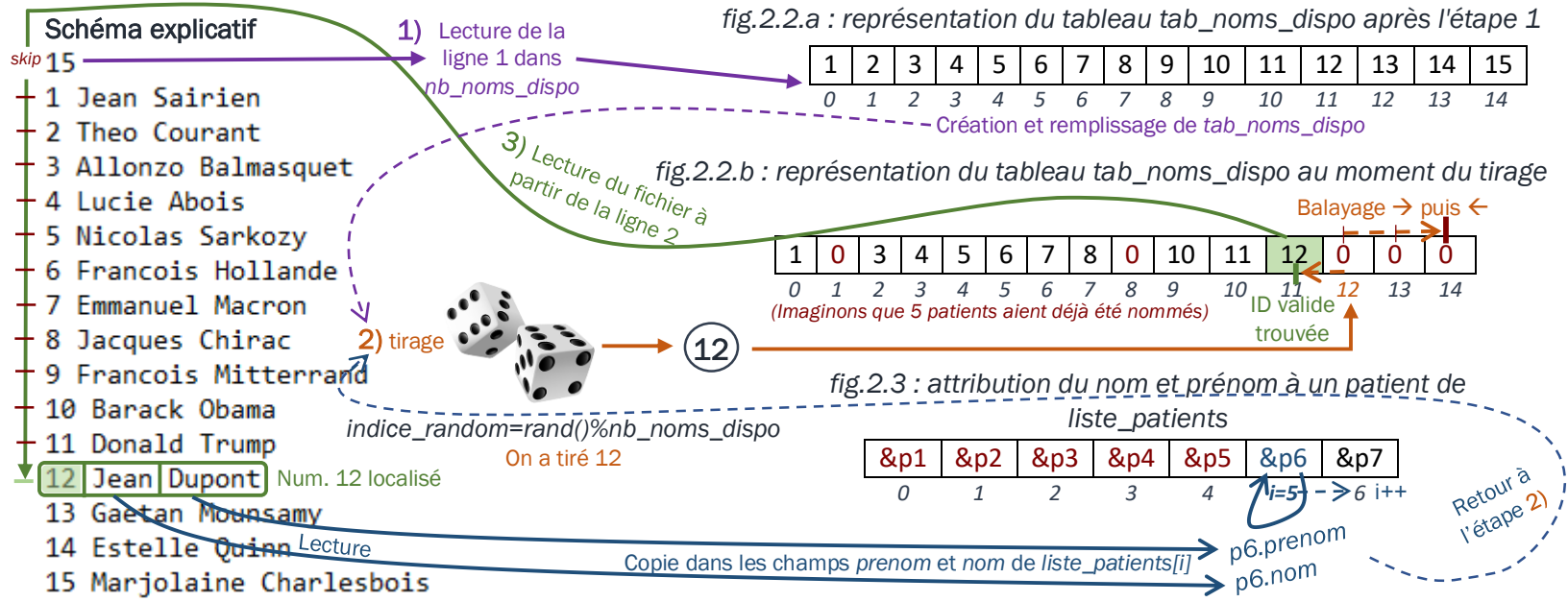


fig.2.1 : contenu d'une liste de noms

fig.2 : illustration du procédé utilisé pour nommer les patients

2.3.b

2.4 – Détermination des circuits empruntés

On considère 3 différents parcours :

- Le **circuit court**, vers lequel sont orientés 40% des patients

Et parmi le circuit long, dans lequel sont envoyés les 60% de patients restants :

- Le parcours « **circuit long *simple*** » : les patients ne subissent pas d'examen complémentaire.
- Le parcours « **circuit long avec examen complémentaire** » : les patients doivent subir un second examen avant de quitter le service d'urgences. (concerne 80% des patients du circuit long).

On représentera ces trois parcours par les différentes valeurs d'une **énumération *circuit*** définie dans "patient.h" : ***COURT***, ***LONG_SIMPLE*** et ***LONG_EXAM_CPT***, correspondant respectivement aux valeurs numériques 0, 1 et 2.

Par défaut, les parcours sont attribués aléatoirement et automatiquement aux patients créés, mais une option permet de les choisir manuellement.

<u>Procédé de détermination automatique :</u>

Les circuits sont déterminés une fois les patients créés.
On utilise pour cela la **génération d'entiers pseudo-aléatoires** basée sur l'heure de lancement du programme.
Des **compteurs associés à chaque circuit** sont initialisés à 0. A l'issue, ils doivent contenir le **nombre de patient passant par chaque circuit**, une valeur nécessaire pour la création des salles et l'établissement de certaines statistiques.

Le processus est répété pour chaque adresse ***ppatient*** du tableau ***liste_patients***. On tire un premier entier ***p*** compris entre 0 et 99 inclus. Les instructions suivantes dépendent de la valeur de ***p*** :

- Si ***p*** < 40, on affecte ***ppatient->circuit = COURT.*** De cette façon, la probabilité qu'un patient soit envoyé en circuit court est de **40%** (40 valeurs possibles dans [[0 ; 39]] et 100 valeurs possibles dans [[0 ; 99]]).
  On incrémente ***nb_patients_cc***, le compteur de patients du circuit court.
  On passe ensuite au patient suivant.

- Sinon (càd. si 40 ≤ ***p*** ≤ 99), le patient est orienté en circuit long. On incrémente ***nb_patients_cl*** *(compteur de patients envoyés en circuit long)* et on tire un second entier ***p*** compris entre 0 et 99 inclus.
  - Si ***p*** > 80 : on affecte ***ppatient->circuit = LONG_SIMPLE.*** De cette façon, la probabilité qu'un patient envoyé en circuit long ne subisse pas d'examen complémentaire vaut 100-80=**20%.**

- Sinon (càd. si $0 \leq p \leq 79$), on affecte ***ppatient->circuit = LONG_EXAM_CPT.***
  Ainsi un patient, orienté en circuit long aura **80%** de chances de subir un examen complémentaire.
  On incrémente ***nb_patients_cl_cpt*** *(compteur de patients du parcours « circuit long avec examen complémentaire »).*

On passe ensuite au patient suivant.

On pourra vérifier la répartition des patients dans les différents circuits à l'aide de la commande '*p*' du menu des statistiques :

```
p

STATISTIQUES RELATIVES AUX CIRCUITS
_____________________________________________________________________

Circuit COURT : 44% des patients.    ≈ 40%
Circuit LONG : 56% des patients.     ≈ 60%
  80% = 80% ont subi des examens complementaires (45% du total).
```

En mode « détermination automatique des circuits », la durée de l'examen complémentaire est demandée pour chaque patient si l'option « durées des tâches données par l'utilisateur » est active.

Choix manuel des circuits :
Si l'option « *choix des circuits par l'utilisateur* » est activée, le programme invite l'utilisateur à saisir le circuit emprunté par chaque patient sous la forme d'un caractère : '*c*' correspond au circuit court et '*l*' au circuit long.
Si l'utilisateur choisit le circuit long, il lui est également demandé si le patient sera amené à subir un examen complémentaire ou non.

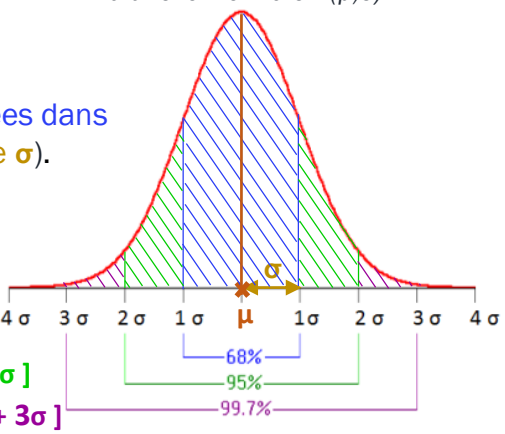
La durée de l'examen complémentaire est demandée uniquement pour les patients admis en examen complémentaire dans ce mode de détermination des circuits.

2.5 – Génération de la durée des tâches

Dans toutes les salles autres que les salles d'attente, on considère la réalisation de **« tâches »** d'une certaine durée. Ces durées sont **des données particulièrement importantes** : elles conditionnent le déroulement de la simulation et ses résultats.
Imaginons par exemple que l'enregistrement soit relativement long et que l'examen IOA soit complété beaucoup plus rapidement. Le débit d'entrée en salle d'attente 2 serait faible, et la salle d'examen IOA serait fréquemment libre sans n'avoir aucun patient à prendre en charge ; En outre, le temps d'attente moyen en salle d'attente 1 serait bien plus élevé que celui de la salle d'attente 2.

<u>Liste exhaustive des tâches rencontrées dans les simulations :</u> *Enregistrement, Examen IOA, Examen principal (circuit court), Examen principal (circuit long), Examen complémentaire*.

fig.2 : Courbe gaussienne caractéristique d'une loi normale N(μ,σ)



<u>Génération automatique des durées :</u>
Par défaut, les durées sont générées aléatoirement, suivant une loi normale.
Leur répartition décrit une courbe de Gauss : la plupart des durées sont centrées dans les environs immédiats d'une valeur (situées entre l'espérance **μ** et l'écart-type **σ**).
Une petite partie d'entre elles sont espacées d'1 à 2 fois **σ** de **μ**.
Presque la totalité des durées restantes (**une infime partie**) se trouvent dans l'intervalle $[\mu - 3\sigma ; \mu - 2\sigma] \cup [\mu + 2\sigma ; \mu + 3\sigma]$.

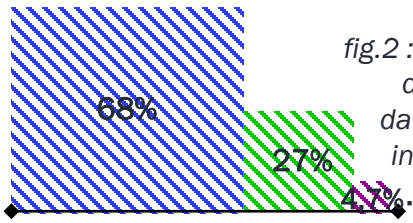


fig.2 : Répartition des durées générées dans les différents intervalles décrits

- $[\mu - \sigma ; \mu + \sigma]$
- $[\mu - 2\sigma ; \mu - \sigma] \cup [\mu + \sigma ; \mu + 2\sigma]$
- $[\mu - 3\sigma ; \mu - 2\sigma] \cup [\mu + 2\sigma ; \mu + 3\sigma]$
- $[\mu - \infty ; \mu - 3\sigma] \cup [\mu + 3\sigma ; \mu + \infty]$

En pratique, une valeur ***esperance*** et une valeur ***ecart*** (égale à 3*σ) sont associées à chaque salle de type (salle_even). Les durées sont générées à la volée, à chaque fois qu'un patient passe par une salle de cette catégorie.
ecart *vaut* 3σ. On fait l'approximation que la totalité des valeurs – et non 99.7% - est comprise entre esperance–ecart et esperance+ecart.
Ainsi, une durée générée avec le couple de paramètres (*esperance* = 40 ; *ecart* = 10), sera comprise dans l'intervalle [30 ; 50].

Trois fonctions sont utilisées pour générer les durées :

- Une fonction ***densite_LN****(esperance,ecart,x)* permettant de calculer une **densité de probabilité** suivant une loi normale N(μ, σ^2 = (ecart/3)²) ; Elle calcule et retourne le réel suivant :

  $$p(x) = \frac{1}{\sigma \times \sqrt{2\pi}} e^{\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}$$, avec x une abscisse choisie entre μ - 3σ et μ + 3σ.

  Elle est appelée à partir d'une fonction qui en calcule l'intégrale.

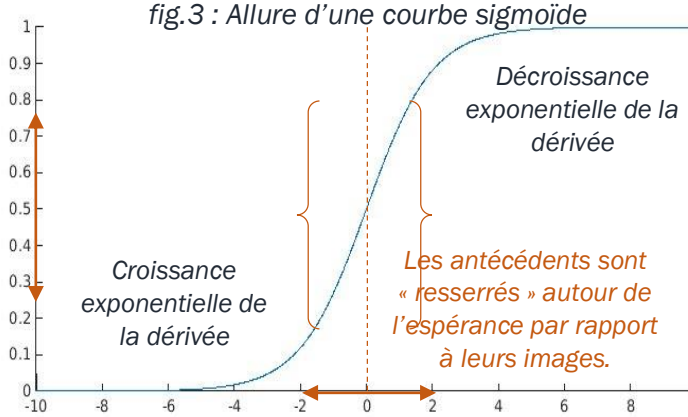
- Une fonction ***integrale_LN****(esperance,ecart,erreur,bornesup)* qui calcule, grâce à la méthode des rectangles, **l'intégrale de la fonction *p*** - que l'on notera ***P*** - sur l'intervalle [esperance – ecart ; ***bornesup***].
  Elle renvoie la moyenne de la sur-estimation et de la sous-estimation de l'intégrale une fois que la distance *abs(surestim – sousestim)* est inférieure à ***erreur***.
  P est la ***fonction de répartition*** de la loi normale N(μ, (ecart/3)²).

- Une fonction *genererDureeTache(esperance,ecart)* qui renvoie la durée générée par le processus suivant :
  - Définition d'un réel ***epsilon*** **majorant l'erreur maximale souhaitée** pour la dichotomie, la comparaison des nombres flottants et l'approximation de l'intégrale.
  - Génération pseudo-aléatoire d'une probabilité ***proba*** comprise entre 0 et *P*(esperance+ecart)
  - **Recherche dichotomique** de l'abscisse ***x*** correspondant à cette probabilité sur l'intervalle [*esperance-ecart* ; *esperance+ecart*], à ***epsilon*** près :
    0 - Initialisation de la borne inf. ***a*** à *esperance-ecart* et de la borne sup. ***b*** à *esperance+ecart* ;
    1 - Calcul de ***x*** = *(**a**+**b**)/2* , la moitié du segment ab.
    2 - Calcul de ***P(x)*** et comparaison avec ***proba +/- epsilon*** :
    - Si ***P(x)*** > ***proba + epsilon*** : b=x (on décale la borne sup. vers la gauche).
    - Si ***P(x)*** < ***proba - epsilon*** : a=x (on décale la borne inf. vers la droite).
    3 - Retour à l'étape 1 tant que ***P(x)*** ∉ [***proba - epsilon*** ; ***proba + epsilon***].
    4 - Fin de la recherche de ***x*** qui vaut maintenant la durée générée en minutes.
  - Conversion de ***x*** en une valeur temporelle de type (*temps*).
  - Retour de ***t***, la durée générée, comprise entre esperance – ecart et esperance + ecart.

*Explication : Comme la représentation graphique de **P** est une courbe **sigmoïde** centrée en **esperance** (croissance exponentielle de la dérivée d'esperance-ecart vers esperance, décroissance exponentielle de la dérivée d'espérance à esperance+ecart), et parce que **proba** est, elle, générée suivant une loi uniforme, **l'antécédent x par P de proba** a une forte probabilité d'être proche de **esperance**.*

*En effet, le taux d'accroissement de **p** au voisinage de **esperance** est extrêmement important par rapport à sa croissance en d'autres abscisses.*

fig.3 : Allure d'une courbe sigmoïde



Couple *(esperance , ecart)* attribué à chaque tâche lors de l'instanciation des salles dans main.c :
Enregistrement : (3,2) ; Examen IOA : (6,3) ; Examens principaux : (45,20) ; Examen complémentaire : (60,30).

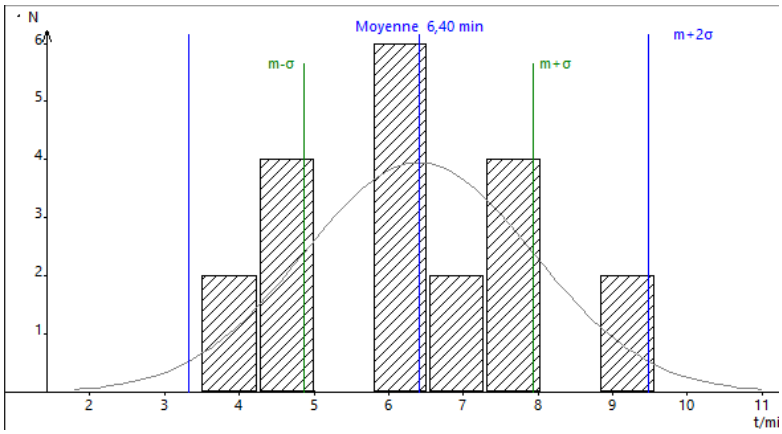
Ces données ont été choisies sur la base de ce rapport : http://www.efurgences.net/publications/attente.pdf

Résultats obtenus : *exemple avec l'examen IOA.*

fig.4.a : Echantillon et statistiques calculées

```
Examen IOA :
        Duree totale : 2 heures et 8 minutes.
        Duree moyenne : 6 minutes et 24 secondes.
        Ecart-type (minutes) : 1.50
        Duree mediane : 6 minutes.
        Duree de la tache pour chaque patient :
                Duree minimale : Patient   4 : (Carine Pichon) : 4 minutes.
                Duree maximale : Patient   8 : (Ella De-Bonzieux) : 9 minutes.
                Patient   4 (Carine Pichon) : 4 minutes.
                Patient  16 (Walter White) : 4 minutes.
                Patient   1 (Tobias Forge) : 5 minutes.
                Patient   3 (Elizabeth Adin) : 5 minutes.
                Patient  17 (Mohammed Afarsa) : 5 minutes.
                Patient  18 (Al Kholiq) : 5 minutes.
                Patient   7 (Clea Molett) : 6 minutes.
                Patient  11 (Jeanne Bot) : 6 minutes.
                Patient   2 (Tobby-Juan Kennoby) : 6 minutes.
                Patient   5 (Larry Silverstein) : 6 minutes.
                Patient   6 (John Bushell) : 6 minutes.
                Patient  20 (Henri Cochet) : 6 minutes.
                Patient   9 (Frank Provost) : 7 minutes.
                Patient  14 (Mona Staire) : 7 minutes.
                Patient  13 (Jiroemon Kimura) : 8 minutes.
                Patient  10 (Alain Provist) : 8 minutes.
                Patient  19 (Jade Herpat) : 8 minutes.
                Patient  12 (Farid Kameche) : 8 minutes.
                Patient  15 (Gaetan Mounsamy) : 9 minutes.
                Patient   8 (Ella De-Bonzieux) : 9 minutes.
```

fig.4.b : Répartition des durées générées



On trouve bien moy. ≈ 6 et σ = 1.5 ≈ 3/3

Pour des nombres de patients supérieurs à 30, la répartition des durées générées se montre satisfaisante. Elle reste cependant très influencée par la composante aléatoire et il se peut que les durées soient générées suivant une tendance inattendue à certains lancements.

<u>Génération manuelle des durées :</u>
En activant une option, l'utilisateur peut aussi choisir manuellement les durées de chaque tâche.
On fera l'acquisition des durées grâce à la fonction *saisir_duree()* décrite précédemment.

2.5

2.6 – Gestion du flux de patients

Le flux de patients est organisé autour des différentes salles instanciées dans la simulation.

On considère une file « ***ENTREE*** » contenant tous les patients de la simulation, organisée par ordre d'arrivée au service des urgences.
La simulation évolue suivant un temps virtuel ***temps_simulation***, initialisé à **l'instant de la première arrivée**, celle du patient en tête de la file *ENTREE.*
Chaque fois que ***temps_simulation*** atteint l'instant d'arrivée du patient en tête de file, celui-ci est récupéré par la salle d'attente 1 et retiré de la file *ENTREE* : il est alors **intégré au flux de patients.**

Le flux de patients est alors entièrement géré au niveau des salles de type (salle_even), que l'on appellera « salles d'événements » : ce sont elles qui acquièrent les patients des salles précédentes et les envoient aux suivantes.

A chaque minute du temps de la simulation, **une boucle balaie toutes les salles d'événement** pour faire évoluer le flux en fonction de **leur état**, de **l'état des salles précédentes**, de ***temps_simulation*** et du **circuit emprunté par chaque patient.**

Si le flux de patients a évolué, on notifie qu'une modification est survenue et **on répète cette boucle jusqu'à ce que les salles d'événement soient dans un état « stable ».** ***temps_simulation*** **est alors incrémenté.**

On considère également une file « ***SORTIE*** », étape ultime de tous les patients de la simulation, qui ne sont alors plus manipulés par l'algorithme de gestion de flux.
Sa capacité maximale est choisie comme étant le nombre de patients de la simulation.
L'ensemble du flux est géré à l'intérieur d'une boucle « **Tant que tous les patients ne sont pas sortis …** », qui se poursuit tant que la file *SORTIE* n'a pas atteint sa capacité maximale.

Une fois cette étape complétée, le flux de patients prend fin.

2.6.a – Représentation des salles

Différentes structures de données sont utilisées pour représenter les salles du service d'urgences.

Files d'entrée / sortie :

On se représentera deux « couloirs » imaginaires, situés **avant l'entrée** et **après la sortie** du service d'urgences.

Les patients arrivant à l'hôpital forment une file indienne dans le couloir d'entrée ; Celle-ci est organisée par ordre d'arrivée au service des urgences. Lorsqu'un patient arrive à l'hôpital, il quitte la file d'entrée, laissant sa place au deuxième patient de la file, qui lui, laisse sa place au troisième …

Les patients quittant le service d'urgences se rangent également dans l'ordre dans le couloir de sortie.

Les files sont représentées par un type structurel appelé « *Queue* », défini dans *queue.h.*
Référence : https://www.thelearningpoint.net/computer-science/data-structures-queues--with-c-program-source-code

Type *Queue* :

capacite (unsigned int) */*Taille du tableau « elements ». Correspond au nombre maximal d'éléments contenus dans la file. Fixé à la création de la file.*/*

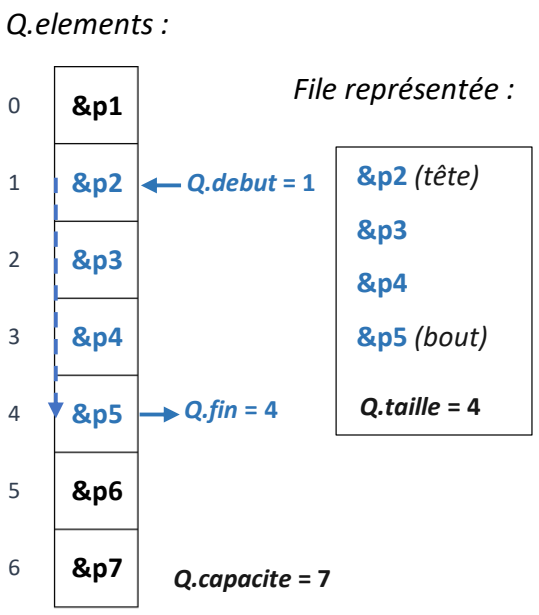
taille (unsigned int) */*Nombre d'éléments considérés dans la file. Evolue lorsqu'on ajoute ou retire un élément.*/*

elements (patient**) */*Tableau d'adresses de patients. La file peut être vue comme la liste des adresses contenues par ce tableau de l'indice "debut" à l'indice "fin"*/*

debut (int) *//elements[debut] correspond à l'élément en tête de file*

fin (int) *//elements[fin] correspond à l'élément en bout de file*

Exemple :



Des fonctions et procédures permettant de manipuler des files sont définies dans la source *queue.c*.
On pourra notamment :

- **Instancier** un élément de type *Queue* de capacité choisie et renvoyer un pointeur *Q* vers celui-ci avec ***creerQueue().***

- **Tester** si une file est vide (en testant la nullité de *Q->taille*) avec ***estVide(Q).***

- Renvoyer l'élément en tête de file (*Q->elements[Q->debut*], ou NULL si *estVide(Q)*) avec ***element_debut_queue().***

- **Ajouter un élément en bout de file** (en vérifiant qu'il reste de la place dans la file, puis en incrémentant *Q->fin* et *Q->taille*, et en insérant l'élément à *Q->elements[Q->fin*]) avec ***enqueue(Q).***

- **Retirer un élément en tête de file** (en vérifiant que *!estVide(Q),* puis en incrémentant *Q->début* et en décrémentant *Q->taille.*) avec ***dequeue(Q).***

- **Trier** un tableau d'adresses de patient **par ordre d'arrivée**, puis créer une Queue à partir de ce tableau et renvoyer un pointeur vers celle-ci, avec ***trierPatients(nb_patients, liste_patients)***.
  (Utilisée pour générer ***ENTREE***).

Les salles d'attente ont également un champ de type *Queue* pour classer les patients par ordre d'entrée et les fournir dans cet ordre aux salles d'événement.

Salles d'attente :
Excepté la salle d'attente 1 qui accueille les patients du couloir d'arrivée une fois que ***temps_simulation*** atteint leur heure d'arrivée, les salles d'attente ont un rôle passif dans la gestion du flux de patients.
Elles sont représentées par le type structurel (*salle_dattente*) défini comme suit dans le header *salles.h* :

Type *salle_dattente* :

```
QUEUE(Queue*)  /*File contenant les patients entrés dans la salle d'attente. Ils en sortiront dans le même ordre (FIFO : first in first out).*/

temps_attente (double)  /*Nombre de minutes attendues dans la salle d'attente. A la fin de la simulation, il vaut la somme du temps attendu par chaque patient. On le remplace ensuite par la durée d'attente moyenne.*/

message (char*)  /*Chaîne de caractère désignant la salle. Exemple « salle d'attente 1 ». Utilisée pour l'affichage des événements. Espace mémoire réservé à la création de la salle et libéré à la fin de la simulation. */
```

Les patients **présents dans la salle d'attente** à un instant donné sont **classés** dans *QUEUE*.

La **somme des durées de séjour** de chaque patient dans la salle d'attente est construite au fur et à mesure de la progression du flux. Elle est conservée dans *temps_attente*.

Une **chaîne identifiant la salle** d'attente est affecté à *message*.

Des fonctions et procédures traitant des éléments de type (*salle_dattente*) sont définies dans la source *salles.c* ;
Parmi elles, on trouve notamment :

- Une fonction ***creerSalleDattente(capacite,message)*** qui permet d'instancier une salle d'attente en créant et en lui attribuant une *Queue* de capacité *capacite*, en initialisant son champ *temps_attente* à 0 et en lui attribuant un message passé en paramètres.
  Elle retourne l'adresse ***S*** de la salle d'attente créée.

- Une procédure ***ajouterPatientSalleDattente(ppatient,S)*** qui permet d'ajouter un patient d'adresse *ppatient* au bout de la file *S->QUEUE* en appelant *enqueue(S->Queue)*.

- Une procédure ***patientsAttendent(S)*** qui **incrémente** *S->temps_attente* **pour chaque patient** présent dans la file *S->Queue* au moment où la fonction est appelée.

Pour simplifier la simulation, on considère que le transfert d'un patient d'une salle à une autre est instantané, et qu'un patient passe par une salle d'attente même si la salle suivante est déjà libre.
Par exemple, le premier patient passera par la salle d'attente 1, puis, à la même minute, en salle d'enregistrement, même si cette dernière était déjà disposée à l'accueillir.

Salles « d'événements » :
Toutes les autres pièces du service d'urgences sont considérées comme des salles « d'événement », c'est-à-dire des salles où l'entrée d'un patient déclenche l'exécution d'une tâche d'une certaine durée, après quoi la salle peut être libérée.

Les salles d'événement jouent un rôle actif dans la simulation : c'est elles qui communiquent directement avec les salles adjacentes, et indirectement avec toutes les autres salles de la simulation.

Les salles d'événement sont représentées par la structure (*salle_even*) de l'en-tête *salles.h*.

Les box sont également considérés comme des salles d'événement, que l'on imbrique dans une autre salle du même type.
Ce sont eux qui accueillent et traitent les patients plutôt que leur salle-mère ; en revanche, les données relatives aux tâches réalisées sont enregistrées dans les champs de la salle-mère et non ceux des box.

La manipulation des salles pourvues de box obéit à des règles différentes que celle des autres salles d'événement.

Les salles d'événement sont représentées comme suit par le type (*salle_even*), défini dans l'en-tête *salles.h* :

Type *salle_even* :

```
//Champs donnant des informations sur la salle d'événement
message(char*)    //Chaîne de caractère désignant la salle

boxs (struct salle_even **)  /*Tableau contenant les adresses des box (considérés comme des
sous-salles d'événement) inclus dans la salle. (Si la salle est un box ou ne contient pas de box, on affectera NULL.)*/
nombre_boxs (unsigned char)  //Nombre de box contenus par la salle. Taille du tableau de box.

salle_prec (salle_dattente*)  //Lien vers la salle D'ATTENTE précédente s'il y en a une, NULL sinon.
salle_suiv (salle_dattente*)  //Lien vers la salle D'ATTENTE suivante s'il y en a une, NULL sinon.
//Ces deux champs sont ignorés pour les salle_even représentant des box.


//Champs liés au rôle de la salle dans la gestion du flux de patients
heure_liberation (temps)  /*Instant de libération de la salle. Calculé comme :
« temps_simulation à l'arrivée du patient + durée de la tâche ».*/

libre(bool)  //Booléen indiquant si la salle est libre ou occupée. Détermine le comportement de la salle dans le flux.

Patient(patient*)  //Adresse du patient contenu dans la salle d'événement.


//Champs nécessaires à l'établissement des statistiques de la simulation. (Ignorés si la salle est un box)
tab_durees(temps*)  /*Tableau contenant pour chaque patient, sa durée de séjour dans la salle. Les durées
correspondent aux patients dont l'id est stockée dans tab_id_patients. Sera trié par ordre de durées croissantes.*/
tab_id_patients(unsigned short int*)  /*Tableau contenant les id des patients dont les durées de séjour sont
contenues aux mêmes indices de tab_durees . Sera trié conjointement à tab_durees.*/

ordre(unsigned short int*)  //Tableau contenant l'id des patients dans l'ordre où ils sont entrés dans la salle.

indice(unsigned short int)  /*Nombre de patients entrés dans la salle à l'issue de la simulation. Utilisé pour balayer
les tableaux de données afin de calculer les statistiques de la même manière pour toutes les salles.*/


/*Champs nécessaires à la génération automatique des durées des tâches associées à chaque salle. (Ignorés pour
les boxs.)*/
d_esperance(unsigned int)  //Espérance choisie pour la génération aléatoire des durées suivant une loi normale.
d_ecart(unsigned int) /*Correspond à 3 fois l'écart-type de la loi normale utilisée pour générer les durées. Les
durées seront toutes comprises entre (d_esperance-d_ecart) et (d_esperance+d_ecart).
```

Les champs du type structurel (*salle_even*) sont utilisés pour l'**affichage des événements**, la **gestion du flux de patients**, le **calcul des statistiques de durée des tâches** et d'**ordre de passage** dans les différents services, et la **génération aléatoire des durées.**

Les champs *salle_prec* et *salle_suiv* **ne peuvent pas contenir d'adresses de salles autres que des salles d'attente.**
Si la salle d'événement considéré n'est pas précédée par une salle d'attente (c'est le cas pour la salle d'examens complémentaires), on affectera *NULL* au champ concerné et on traitera le cas séparément, la façon d'acquérir des patients n'étant pas la même.
Si la salle d'événement considérée n'est pas suivie par une salle d'attente (c'est le cas pour les salles d'examens principaux), on affectera également *NULL*. Les patients seront envoyés à la file *SORTIE*.
Si la salle d'événement considérée est la **salle d'examen IOA**, le lien *salle_suiv* est **affecté en fonction du circuit** emprunté par le patient traité : *SDA4* pour le circuit court et *SDA3* pour le circuit long.

La plupart des traitements réalisés sur les salles d'événement s'effectuent directement dans la source *main.c*. D'autres sont réalisés par des fonctions définies dans la source *salles.c* :

- Les salles de type (*salle_even*) sont instanciées au moyen de la fonction ***creerSalleEven****(message, Salle_Prec, Salle_Suiv,* ***nb_patients****,D_Esperance,D_Ecart,Boxs,Nombre_Boxs)*.
  Elle initialise les champs de la salle avec les données passées en entrée et réserve l'espace mémoire nécessaire à la salle et à ses tableaux de données *tab_durees, tab_id_patients* et *ordre* (tous les trois de taille *nb_patients*). Elle retourne l'adresse ***S*** de la salle nouvellement créée.
  Elle est **appelée depuis la fonction main avant le début du flux de patients.**
  Nb_patients correspond au **nombre de patients qui passera par la salle *S :**
  Pour la salle d'enregistrement et la salle d'examen IOA, on indiquera le nombre total de patients.
  Pour les autres salles, on indiquera le compteur de patients du parcours associé à la salle.

- Les box sont instanciés à l'aide de la fonction ***creerBox****(message)*. Les champs utilisés sont initialisés : par exemple, *libre* est initialisé à *true* pour permettre à la salle d'accueillir son premier patient.
  Les champs inutilisés sont initialisés en conséquence : les tableaux sont initialisés à *NULL*, le nombre de box à 0…

- Pour grouper les box afin de les intégrer à une salle-mère, on utilise la fonction ***grouperBoxs****(nb_boxs,…)* afin de créer un tableau de taille *nb_boxs* contenant les adresses des *nb_boxs* box passés en argument.
  Il s'agit d'une **fonction à nombre d'arguments variable**, utilisant les macros de la bibliothèque *<stdarg.h>*.
  Le tableau (*salle_even ***) renvoyé pourra être affecté au champ *boxs* d'une salle-mère.

- Pour trier conjointement les tableaux ***S->tab_durees*** et ***S->tab_id_patients*** à l'issue de la simulation, on appellera la procédure ***trierStats****(S)*.

- Pour calculer la durée totale de la tâche associée à la salle **S* (en cumulant les durées de séjour de chaque patient), on appellera ***dureeTotale****(S)*.

Liste des salles instanciées dans la simulation :
Déclarées au début de main.c ; Instanciées juste avant le début du flux de patients.

ENTREE *(Queue*)*	*Couloir d'entrée imaginaire*
SDA1 *(salle_dattente*)*	Salle d'attente 1
S_ENREG *(salle_even*)*	Salle d'enregistrement
SDA2 *(salle_dattente*)*	Salle d'attente 2
S_EXAMEN_IOA *(salle_even*)*	Salle d'examen IOA
SDA3 *(salle_dattente*)*	Salle d'attente 3
BOX1, BOX2, BOX3 *(salle_even*)*	Box de la salle d'examen principale du circuit long
S_EXAMEN_LONG *(salle_even*)*	Salle d'examen principale du circuit long
BOX_CPT1, BOX_CPT2 *(salle_even*)*	Box de la salle d'examen complémentaire du circuit long
S_EXAM_CPT *(salle_even*)*	Salle d'examen complémentaire du circuit long
SDA4 *(salle_dattente*)*	Salle d'attente 4
BOX4, BOX5 *(salle_even*)*	Box de la salle d'examen principale du circuit court
S_EXAMEN_COURT *(salle_even*)*	Salle d'examen principale du circuit long
SORTIE *(Queue*)*	*Couloir de sortie imaginaire*

Un tableau ***tableau_salles_even[5]*** contenant *S_ENREG*, *S_EXAMEN_IOA*, *S_EXAMEN_LONG*, *S_EXAMEN_COURT* et *S_EXAM_CPT* est utilisé pour réaliser des traitements groupés sur les salles d'événement.
Un tableau ***tableau_salles_attente[4]*** contenant *SDA1*, *SDA2*, *SDA3* et *SDA4* est utilisé pour réaliser des traitements groupés sur les salles d'attente.

2.6.b – Principe général

Le procédé mis en œuvre pour faire évoluer le flux de patients **n'est pas le même pour toutes les salles d'événements** ;
Cependant, si on le simplifie **en faisant abstraction de leurs différences**, on peut le généraliser comme suit :

Instructions préalables au début du flux

- Instanciation de toutes les salles et remplissage de la file *ENTREE*.
- Initialisation de variables booléennes influant sur l'affichage à leur valeur par défaut.
- Ouverture du fichier « ***historique.txt*** » en écriture.
- Déclaration de **variables intermédiaires** pour manipuler les patients, les box, etc … d'une salle.
- Initialisation de ***temps_simulation*** à l'**heure d'arrivée du premier patient** de *ENTREE*.

Boucle a : « *tant que tous les patients ne sont pas sortis …* »

- Affichage de la date si elle a changé.

Boucle b : « *Une $1^{ère}$ fois, puis tant que modification…* »

- *modification = false ;*

Si *!estVide(ENTREE)*

Si « *temps_simulation a atteint l'heure d'arrivée du patient en tête d'ENTREE :*

- *ppatient* = ***element_debut_queue****(ENTREE) ;* *//on prend le patient en tête d'ENTREE*
- ***ajouterPatientSalleDattente****(SDA1,ppatient) ;* *//on l'envoie en bout de la file de SDA1*
- ***dequeue****(ENTREE) ; //on le supprime d'ENTREE*
- « Afficher l'événement et le graver dans l'historique »

Boucle « *Pour chaque salle ou box psalle …* *//on balaie les salles de salle_even, et chacun de leurs box*

Si « **psalle est occupée » :*

Si « *temps_simulation a atteint l'heure de libération de psalle : »*

- *ppatient* = ***psalle->Patient*** *; //on prend le patient contenu dans la salle ou le box*
- « **Envoyer le patient dans la sda. suivante**/un box de la salle suivante/*SORTIE* »
- « **Afficher l'événement** et le graver dans l'historique »
- ***psalle->Patient = NULL ; psalle->libre = false ;*** *//psalle redevient disponible*
- *modification = true ; //on enregistre une modification pour répéter la boucle a*

Maintenant, si « **psalle est libre » :*

Si « *un patient attend dans la salle précédente» :*

- *ppatient* = « ***patient attendant*** » */*Patient en tête de file de la salle d'attente précédente, ou dans un box prêt à être libéré de la salle précédente pour la salle d'examen complémentaire*/*
- ***psalle->Patient*** = *ppatient ; //on prend le patient de la salle précédente*
  psalle->libre = *false ; //la salle a maintenant le statut « occupée »*
- « **Afficher l'événement** et le graver dans l'historique »
- « Obtenir ***duree*** » *//on génère automatiquement ou on demande la saisie de la durée de la tâche*
- ***psalle->heure_liberation*** = *temps_simulation ;) //on màj. l'heure de libération de la salle*
  ajouterTemps*(&psalle->heure_liberation,duree) //vaut désormais temps_simulation+duree*
- « Incrémenter psalle->indice et mettre à jour les tableaux de données de *psalle »
- « Supprimer le patient proposé par la salle précédente » *//&la passer à « libre »*
- *modification = true ; //on enregistre une modification pour répéter la boucle a*

- ***incrementerTemps(&temps_simulation) ;*** *//ajout d'une minute à temps_simulation*

Boucle c : « *Pour chaque salle sda de tableau_salles_attente… »*
Mise à jour du temps d'attente total : ***patientsAttendent(sda) ;***

Instructions exécutées une fois tous les patients sortis

- « Tri des tableaux de données et calcul de certaines statistiques liées aux salles ».
- « Fermeture du fichier historique »
- « Affichage du menu principal de fin de simulation. »

Cas particuliers

En pratique, on balaye les 5 « salles-mères » de *tableau_salles_even* et **on distingue les salles possédant des box et les salles qui en sont dépourvues.**

Si la salle *psalle* ne possède pas de box

Elle se libère en envoyant son patient en bout de file de *psalle->salle_suiv->QUEUE et* se remplit en prenant le patient *en tête de file de* patient *psalle->salle_prec->QUEUE.*
Si la salle *psalle* est *S_EXAMEN_IOA*, *psalle_salle->suiv* est attribuée en fonction de *psalle->Patient->circuit* avant d'être libérée.

Sinon : la salle *psalle* possède des box

S'il ne s'agit pas de *S_EXAM_CPT* :

S'il *psalle* pas non plus *S_EXAMEN_LONG* :

Chaque sous-salle *pbox* de *psalle->boxs* se libère en envoyant son patient dans la file *SORTIE*

Sinon, il s'agit de *S_EXAMEN_LONG :*

Chaque sous-salle *pbox* de *psalle->boxs* se libère en envoyant son patient dans la file *SORTIE* **si et seulement si *pbox->Patient->circuit* est différent de *LONG_EXAM_CPT* : sinon, *pbox* doit attendre d'être libéré** par *S_EXAM_CPT*.

Dans les deux cas, chaque sous-salle *pbox* de *psalle->boxs se remplit en prenant les patients en tête de file de psalle->salle_prec->QUEUE*.

S'il s'agit de *S_EXAM_CPT* :

Chaque sous-salle *pbox* de *psalle->boxs* se libère en envoyant son patient dans la file *SORTIE.*
Chaque sous-salle *pbox* de *psalle->boxs* se remplit en prenant un patient d'une sous-salle *pbox2* de *S_EXAMEN*_LONG-*>boxs* dont l'instant de libération est atteint. Il faut également indiquer pbox2->libre=true une fois *pbox2* libéré.

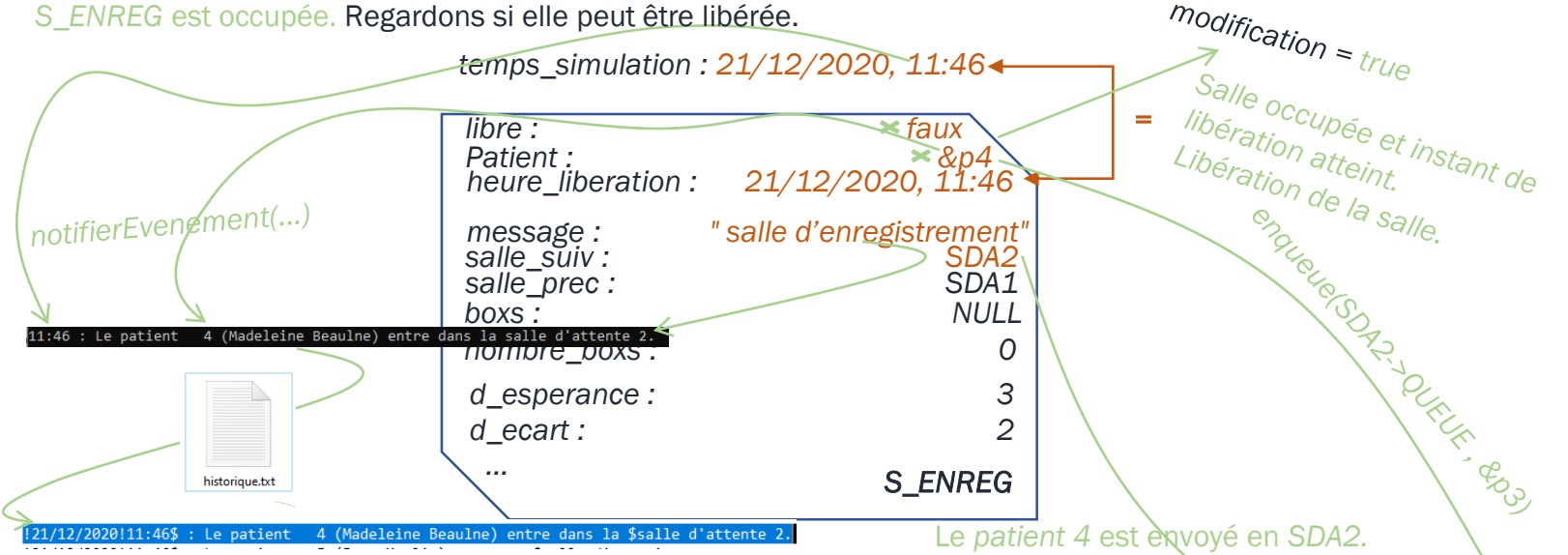
2.6.b

2.6.c – Exemples de situations

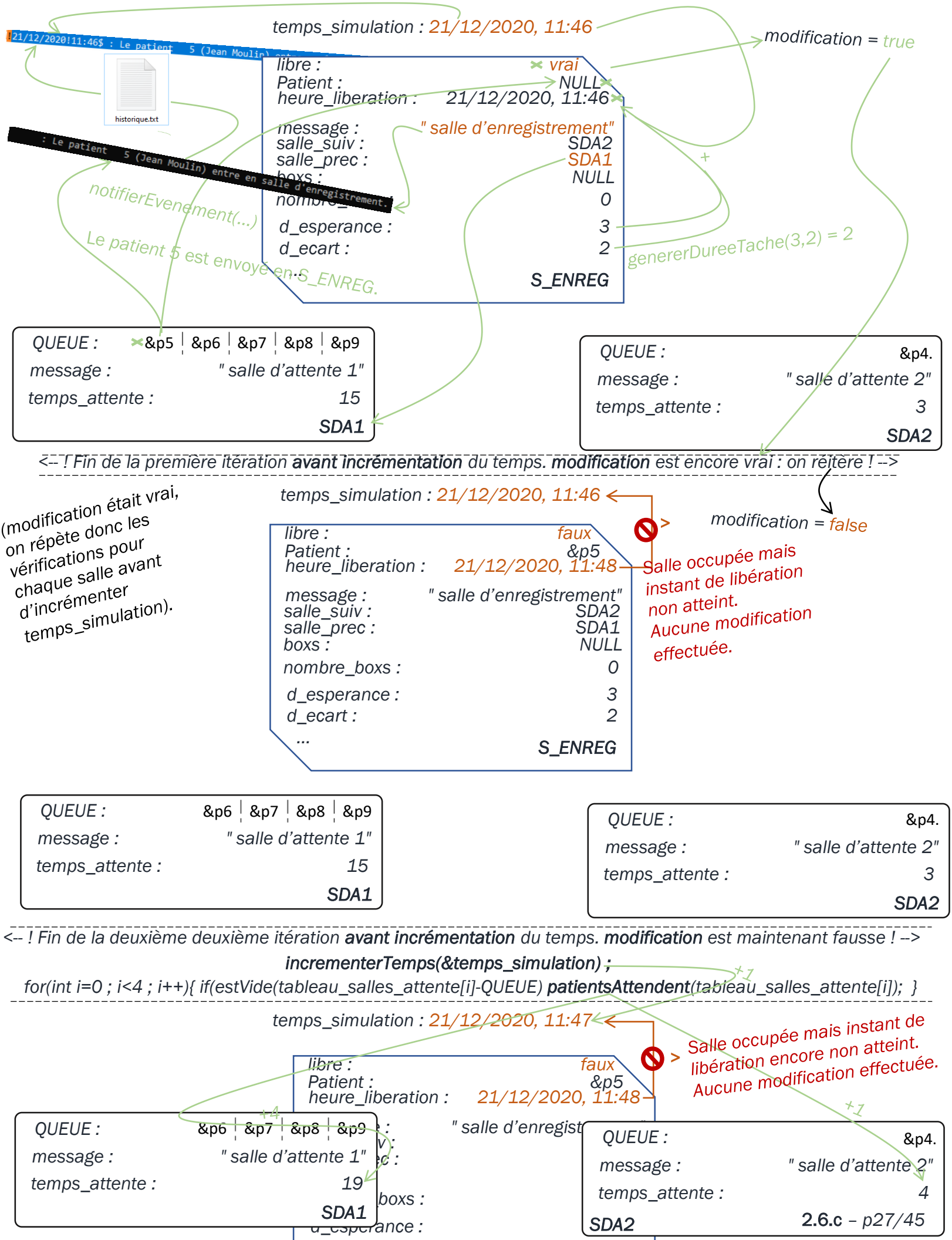
Voici des exemples illustrant plusieurs situations rencontrées lors de la gestion du flux de patients :

Exemple 1 : Fonctionnement de la salle d'enregistrement

S_ENREG est occupée. Regardons si elle peut être libérée.



S_ENREG est libérée. Regardons si elle peut accueillir un nouveau patient dès maintenant.


temps_simulation : 21/12/2020, 11:46
modification = true
historique.txt
: Le patient 5 (Jean Moulin) entre en salle d'enregistrement.
libre : vrai
Patient : NULL
heure_liberation : 21/12/2020, 11:46
message : " salle d'enregistrement"
salle_suiv : SDA2
salle_prec : SDA1
boxs : NULL
nombre_boxs : 0
d_esperance : 3
d_ecart : 2
... S_ENREG
notifierEvenement(...)
Le patient 5 est envoyé en S_ENREG.
genererDureeTache(3,2) = 2
QUEUE : &p5 | &p6 | &p7 | &p8 | &p9
message : " salle d'attente 1"
temps_attente : 15
SDA1
QUEUE : &p4.
message : " salle d'attente 2"
temps_attente : 3
SDA2
<-- ! Fin de la première itération avant incrémentation du temps. modification est encore vrai : on réitère ! -->
(modification était vrai, on répète donc les vérifications pour chaque salle avant d'incrémenter temps_simulation).
temps_simulation : 21/12/2020, 11:46
modification = false
libre : faux
Patient : &p5
heure_liberation : 21/12/2020, 11:48
message : " salle d'enregistrement"
salle_suiv : SDA2
salle_prec : SDA1
boxs : NULL
nombre_boxs : 0
d_esperance : 3
d_ecart : 2
... S_ENREG
Salle occupée mais instant de libération non atteint. Aucune modification effectuée.
QUEUE : &p6 | &p7 | &p8 | &p9
message : " salle d'attente 1"
temps_attente : 15
SDA1
QUEUE : &p4.
message : " salle d'attente 2"
temps_attente : 3
SDA2
<-- ! Fin de la deuxième deuxième itération avant incrémentation du temps. modification est maintenant fausse ! -->
incrementerTemps(&temps_simulation) ;
for(int i=0 ; i<4 ; i++){ if(estVide(tableau_salles_attente[i]-QUEUE) patientsAttendent(tableau_salles_attente[i]); }
+1
temps_simulation : 21/12/2020, 11:47
libre : faux
Patient : &p5
heure_liberation : 21/12/2020, 11:48
Salle occupée mais instant de libération encore non atteint. Aucune modification effectuée.
+4
+1
QUEUE : &p6 | &p7 | &p8 | &p9
message : " salle d'attente 1"
temps_attente : 19
SDA1
QUEUE : &p4.
message : " salle d'attente 2"
temps_attente : 4
SDA2

Exemple 2 : Déroulement d'un flux de patients simple

Générons manuellement deux patients auxquels on attribue les caractéristiques suivantes :

	Patient 1	Patient 2
« Prénom Nom »	Arrive En Premier	Arrive En Deuxieme
Heure d'arrivée (le 21/12/2020)	10:00	10:02
Parcours choisi	Circuit long avec examen complémentaire	Circuit long avec examen complémentaire
Durée de l'examen complémentaire	45 minutes	40 minutes

Exécutons la simulation :

```
DEBUT DE LA SIMULATION :
________________________________________________________________________________

       Le : 21/12/2020
10:00 : Le patient   1 (Arrive En Premier) arrive aux urgences et entre en salle d'attente 1.
      : Le patient   1 (Arrive En Premier) entre en salle d'enregistrement.
        Entrez la duree de la tache au format hh:mm ou mm :  4 ⟶ 10:04

                                                                    ⟶ 10:04
10:02 : Le patient   2 (Arrive En Deuxieme) arrive aux urgences et entre en salle d'attente 1.
```

Comme prévu, le patient 1 arrive aux urgences le 21/12/2020 à 10:00.
La salle d'enregistrement est disponible, il y entre directement. Il va y passer 4 minutes.
Le patient 2 arrive à 10:02. La salle d'enregistrement étant occupée par le patient 1 jusqu'à 10:04, il doit patienter en salle d'attente 1.

```
10:04 : Le patient   1 (Arrive En Premier) entre dans la salle d'attente 2.
      : Le patient   2 (Arrive En Deuxieme) entre en salle d'enregistrement.
        Entrez la duree de la tache au format hh:mm ou mm :  3 ⟶ 10:07
      : Le patient   1 (Arrive En Premier) entre en salle d'examen IOA.
        Entrez la duree de la tache au format hh:mm ou mm :  6 ⟶ 10:10
```

A 10:04, le patient 1 termine son enregistrement et est envoyé en salle d'attente 2.
Il peut directement entrer dans la salle d'examen IOA. L'examen se terminera à 10:10.
La salle d'enregistrement ayant été libérée, le patient 2, qui attendait en salle d'attente 1, peut maintenant y entrer et y rester jusqu'à 10:07.

```
10:07 : Le patient   2 (Arrive En Deuxieme) entre dans la salle d'attente 2. ⟶ 10:10
```

A 10:07, le patient 2 termine son enregistrement et est envoyé en salle d'attente 2.
Il attendra jusqu'à 10:10 que la salle d'examen IOA soit libérée.

```
10:10 : Le patient   1 (Arrive En Premier) entre dans la salle d'attente 3.
      : Le patient   2 (Arrive En Deuxieme) entre en salle d'examen IOA.
        Entrez la duree de la tache au format hh:mm ou mm :  4 ⟶ 10:14
      : Le patient   1 (Arrive En Premier) entre en salle d'examens (circuit long), box 1.
        Entrez la duree de la tache au format hh:mm ou mm :  35 ⟶ 10:45
```

A 10:10, l'examen IOA du patient 1 se termine.
Etant orienté vers le circuit long, la prochaine étape de son parcours est la salle d'attente 3. Il y entre.
Les 3 box de la salle d'examen principale sont libres. Il entre dans le premier et l'occupe jusqu'à 10:45.
La salle d'examen IOA ayant été libérée, le patient 2, qui attendait en salle d'attente 2, peut maintenant y entrer et y rester jusqu'à 10:14.

```
10:14 : Le patient   2 (Arrive En Deuxieme) entre dans la salle d'attente 3.
      : Le patient   2 (Arrive En Deuxieme) entre en salle d'examens (circuit long), box 2.
        Entrez la duree de la tache au format hh:mm ou mm :  26 ⟶ 10:40
```

A 10:14, l'examen IOA du patient 2 se termine.
Etant orienté vers le circuit long, la prochaine étape de son parcours est la salle d'attente 3. Il y entre.
Les box 2 et 3 de la salle d'examen principale sont libres. Il entre dans le box 2 et l'occupe jusqu'à 10:40.

```
10:40 : Le patient   2 (Arrive En Deuxieme) est pris en charge au box d'examen complementaire 1.
                                                                    ⟶ 11:25
```

A 10:40, l'examen principal du patient 2 se termine.
Il doit maintenant passer 45 minutes au box d'examen complémentaire *(durée choisie à la création du patient)*.
Son examen se terminera donc à 11:25. Les deux box de la salle sont libres ; il entre dans le premier.

Générons manuellement deux patients auxquels on attribue les caractéristiques suivantes :

```
10:45 : Le patient   1 (Arrive En Premier) est pris en charge au box d'examen complementaire 2.
```

A 10:45, l'examen principal du patient 1 se termine. → 11:25
Il doit maintenant passer 40 minutes au box d'examen complémentaire *(durée choisie à la création du patient)*.
Son examen se terminera donc à 11:25. Seul le box 2 est libre ; il y est dirigé.

```
11:25 : Le patient   2 (Arrive En Deuxieme) quitte le service des urgences.

11:25 : Le patient   1 (Arrive En Premier) quitte le service des urgences.


FIN DU FLUX DE PATIENTS
```

A 11:25, les deux patients terminent leur examen complémentaire. Ils peuvent quitter le service des urgences.
Tous les patients simulés étant sortis de l'hôpital, le flux de patients se termine.

2.6.d – Affichage des énénements

Voici un exemple d'affichage des événements du flux de patients par le programme :

```
       Le : 21/12/2020
10:00 : Le patient   1 (Toto Toulmond) arrive aux urgences et entre en salle d'attente 1.
      : Le patient   1 (Toto Toulmond) entre en salle d'enregistrement.
      : Le patient   2 (Bob Palindrome) arrive aux urgences et entre en salle d'attente 1.

10:04 : Le patient   1 (Toto Toulmond) entre dans la salle d'attente 2.
```

Les règles suivantes s'appliquent :

- La date, **si elle n'a pas encore été affichée à l'écran**, est écrite sur une ligne, séparée des événements.
- Une ligne décrivant un événement se compose comme suit :
  - Ecriture de **l'heure si elle n'a pas encore été affichée**. **Saut de ligne avant un changement d'heure.**
  - Indication de l'ID, du prénom et du nom du **patient concerné.**
  - Ecriture d'une proposition adaptée à l'événement : « *arrive aux urgences et entre en …* » « *est pris en charge au …* »
  - Conclusion de la phrase par le **nom donné à la salle** concernée (et potentiellement celui du box).

Les événements sont tous affichés au moyen de la fonction
bool ***notifier_evenement****(bool sauter, temps t, char *descriptif, char *message_salle, FILE *f)*, qui permet d'afficher dans la console un message correspondant à un événement et respectant les règles énoncées précédemment.
Elle grave par le même temps une version modifiée du message dans un fichier qui servira d'historique.

Pour éviter de réafficher l'heure lorsqu'elle a déjà été donnée, on utilise un booléen : ***changementminute*** initialisé ,et réinitialisé à *vrai* chaque fois que ***temps_simulation*** est incrémenté.
On appellera systématiquement la fonction *notifier_evenement* comme suit :
changementminute = *notifier_evenement(****changementminute****,……)*
En effet, celle-ci sautera une ligne et affichera l'heure avant d'écrire le descriptif de l'événement si la valeur de son premier paramètre est *vrai*. Elle retourne systématiquement *faux*.
Ainsi, **après chaque changement de *temps_simulation*, *changementminute*** vaudra *vrai* et l'affichage du premier événement sera précédé d'un saut de ligne et de l'écriture de l'heure. **Une fois cet affichage réalisé,** *changementminute* vaudra ***faux*** **jusqu'à sa prochaine réinitialisation** : l'heure ne sera plus réaffichée avant chaque événement.

Pour éviter de réafficher la date lorsqu'elle a déjà été donnée, on enregistre *temps_simulation* avant chaque incrémentation dans ***temps_simulation_prec*****.**
On retient également, à chaque fois qu'un événement est organisé (exemple : sortie planifiée de la salle d'enregistrement), ***jour_prochainevenement***, la prédiction du jour **le plus proche** durant lequel se **produira le prochain événement.** (Cette prédiction permet d'éviter d'afficher la date d'un jour où aucun événement ne se produit. Si l'utilisateur crée, par exemple, un patient arrivant au début d'un mois, et un autre arrivant à la fin de ce

mois, il n'est pas souhaitable d'afficher la date chaque jour où aucun patient n'est présent au service des urgences.
On affiche donc la date en début de boucle « *tant que tous les patients ne sont pas sortis …* » sous la condition ***changementdate*** suivante :
changementdate = ***(temps_simulation_prec.jour*** ≠ ***temps_simulation.jour*** ET ***temps_simulation.jour==jour_prochainevenement***).

Enfin, pour construire le message notifiant un événement, on combine dans une chaîne *descriptif* les informations données par les champs *id*, *nom* et *prenom* du patient transféré et la proposition faisant le lien avec le nom de la salle.
La fonction *notifier_evenement* écrira éventuellement l'heure passée en paramètre d'entrée (on utilisera *temps_simulation*), suivie de *descriptif*, suivie du champ *message* de la salle concernée.
Cas particuliers :
On combinera les chaînes *message* des box et de leur salle-mère.
Le message associé à la salle *SORTIE* est simplement « *quitte le service des urgences* ». **2.6.d**

2.7 – Ecritures et lectures des historiques

Un fichier « ***historique.txt*** » est créé ou réécrit à chaque simulation pour enregistrer son déroulement. Il peut ensuite être lu pour afficher l'historique complet d'une simulation ou celui d'un patient unique.

fig.1 : organisation du fichier historique produit

```
        Le : 29/02/2020
!29/02/2020!10:06$ : Le patient   1 (Tom Connaughton) arrive aux urgences et entre en $salle d'attente 1.
!29/02/2020!10:06$ : Le patient   1 (Tom Connaughton) entre en $salle d'enregistrement.

!29/02/2020!10:08$ : Le patient   1 (Tom Connaughton) entre dans la $salle d'attente 2.
!29/02/2020!10:08$ : Le patient   1 (Tom Connaughton) entre en $salle d'examen IOA.
```

L'écriture de l'historique est réalisée en même temps que l'affichage en direct par la fonction ***notifierEvenement.***

Elle ne suit pas les mêmes règles que l'affichage dans la console :
Afin de pouvoir lire l'historique d'un patient de manière optimale, on doit rajouter, **au début de chaque ligne**, la **date et l'heure**, ainsi que des **caractères spéciaux** ' *!* ' et ' *$* ' qui servent à **séparer les différentes parties** des messages.

2.7.a - Historique de la simulation

La commande *'h'* du menu principal de la simulation permet de l'afficher.
La lecture de l'historique complet est effectuée caractère par caractère.

A chaque ligne, si un caractère ' *!* ' est lu, tous les caractères lus jusqu'au prochain ' *!* ' inclus sont **ignorés**. De cette façon, **une même date n'est écrite qu'une fois, comme pour l'affichage en direct.**

Une fois le deuxième ' *!* ' atteint, on insère les caractères suivants dans une chaîne ***heure*** jusqu'à rencontrer un caractère ' *$* '. Si l'heure n'a pas encore été affichée à l'écran, on l'écrit au début d'un événement, comme pour l'affichage en direct, et on la sauvegarde dans une chaîne ***heure_prec***.
Tant que *heure_prec* et *heure* sont identiques, l'heure n'est pas affichée.
De cette manière, **une même heure n'est écrite qu'une fois, comme pour l'affichage en direct.**

A partir du premier *'$'* rencontré, tous les caractères sont affichés à l'exception du second caractère '$'.

fig.2.1 : méthode de lecture de l'historique complet : caractères affichés et ignorés

```
        Le : 29/02/2020
!29/02/2020!10:06$ : Le patient   1 (Tom Connaughton) arrive aux urgences et entre en $salle d'attente 1.
!29/02/2020!10:06$ : Le patient   1 (Tom Connaughton) entre en $salle d'enregistrement.
```

fig.2.2 : lecture de l'historique par le programme

```
        Le : 29/02/2020
10:06 : Le patient   1 (Tom Connaughton) arrive aux urgences et entre en salle d'attente 1.
      : Le patient   1 (Tom Connaughton) entre en salle d'enregistrement.
```

Mais alors, pourquoi ne pas simplement écrire exactement les mêmes caractères dans l'historique et dans la console ?

2.7.b - Historique d'un patient choisi

Si l'on adoptait les mêmes règles que pour l'écriture dans la console, il serait difficile d'établir l'historique d'un patient unique ; Prenons un exemple :

fig.3 : exemple de situation problématique

```
23:58 : Le patient  19 (Garance Marillier) arrive aux urgences et entre en salle d'attente 1.
      : Le patient   7 (Leonid Brejnev) entre dans la salle d'attente 2.
      : Le patient   8 (Barbara Liskov) entre en salle d'enregistrement.
      : Le patient  20 (Linus Torvalds) arrive aux urgences et entre en salle d'attente 1.

        Le : 01/03/2020
00:03 : Le patient   8 (Barbara Liskov) entre dans la salle d'attente 2.
      : Le patient   9 (Otto Graaf) entre en salle d'enregistrement.

00:04 : Le patient   4 (Jacques Ouzy) entre dans la salle d'attente 4.
      : Le patient   5 (Medard Gauthier) entre en salle d'examen IOA.
```

Imaginons que l'on veuille lire l'historique du patient 5 uniquement :
Trois problèmes se présentent alors :

- La **date a changé** entre deux événements, **il faut absolument l'afficher.** Le patient 5 n'est pas directement affiché après la nouvelle date, il est donc **impossible de récupérer cette information.**
- **L'heure n'est pas écrite avant l'événement concernant le patient 5.** Or, elle a changé depuis sa dernière tâche. Il faut donc trouver un moyen de la récupérer pour l'afficher dans l'historique du patient 5.
- La proposition contenant les informations du patient et la proposition de liaison sont **superflues** pour la lecture de l'historique d'un patient choisi. **Seule la salle concernée constitue une information utile.**

En écrivant, dans le fichier-historique, chaque événement de la manière suivante :

```
!01/03/2020!00:04$ : Le patient   5 (Medard Gauthier) entre en $salle d'examen IOA.
```

On s'assure de pouvoir récupérer la date, l'heure et le nom de la salle.

- La date est comprise entre deux caractères *' ! '*. Elle est lue et stockée dans une chaîne ***date***. Elle est affichée si et seulement si elle est différente d'une chaîne ***date_prec*** correspondant à la dernière date affichée. Chaque fois que ***date*** est affichée, on la copie dans ***date_prec*** pour ne pas la réécrire à nouveau.
- L'heure est délimitée à gauche par le caractère *' ! '* et à droite par le caractère *' $ '*. Grâce à une méthode similaire, elle est affichée au début d'une ligne **uniquement si elle a changé.**
- Les caractères suivants, jusqu'au prochain *' $ '*, **sont ignorés** : il est **superflu d'écrire plusieurs fois d'affilée le numéro et le nom du patient choisi**.
  Seuls les caractères lus à partir du dernier caractère *' $ '* sont affichés en fin de ligne. Ils indiquent **la salle concernée.**

En pratique, la lecture s'effectue en **stockant chaque ligne dans une chaîne « *evenement* » tour à tour.**

fig.4 : affichage de l'historique d'un patient

```
HORAIRES DE PASSAGE DANS LES DIFFERENTS SERVICES
_______________________________________________
Parcours du patient 5 (Medard Gauthier) :

        Le 29/02/2020 :
23:38 : Salle d'attente 1.

23:45 : Salle d'enregistrement.

23:49 : Salle d'attente 2.

        Le 01/03/2020 :
00:04 : Salle d'examen IOA.

00:12 : Salle d'attente 4.

00:56 : Salle d'examens (circuit court), box 5.

01:56 : Quitte le service des urgences.
```

La fonction ***strstr*** de <*string.h*> est utilisée pour localiser une portion de message *"Le patient ID ("* dans la chaîne evenement – *ID* étant l'*id* du patient choisi par l'utilisateur.

Si elle n'a pas été trouvée, **on passe directement à la ligne suivante.**
Si elle a été trouvée, on parcourt les différentes parties *d'evenement* grâce à un curseur que l'on positionne à l'aide des fonctions *strchr* (pour **localiser les premier caractère *' ! ' ou ' $ '***) et *strrchr*(**pour localiser les derniers**).

La date, l'heure et la salle sont lues dans des chaînes correspondantes. La date et l'heure sont éventuellement affichées. La salle l'est systématiquement.

2.8 – Statistiques de la simulation

Ordre d'arrivée et de sortie des patients, temps passé aux urgences - commande *'a'* - *(main.c, 1408-1470)*
Le tableau ***liste_patients***, contenant les adresses de tous les patients de la simulation, est trié avant le début du flux de patients pour construire la file *ENTREE*. Une fois la simulation terminée, on pourra le parcourir pour récupérer les adresses de tous les patients dans l'ordre où ils sont arrivés.

```
Ordre d'arrivee :

              Le 10/10/2020 :
         1 - 11:01 : Patient   1 (Adrienne Kepoura)
         2 - 12:02 : Patient   2 (Ossama Bienaidet)
         3 - 13:04 : Patient   3 (Josephine Japy)
         4 - 14:03 : Patient   4 (Farah Paupierre)
         5 - 14:59 : Patient   5 (Bruce Lee)
         6 - 16:03 : Patient   6 (Marcus Aurelius)
         7 - 16:58 : Patient   7 (Daniel Keown)
         8 - 17:58 : Patient   8 (Julie Dante-Pansey)
         9 - 19:15 : Patient   9 (Otto Graaf)
        10 - 19:52 : Patient  10 (Omer Dilpleux)
```

On récupère, pour chaque patient adressé dans *liste_patient*, son instant d'arrivée, son id, son prénom et son nom (ces informations sont contenues dans les champs du type *patient*).

Un message est ensuite construit sur la base de ces informations, puis il est affiché à l'écran.

Pour l'ordre de sortie, on procèdera de manière similaire. On prendra cette fois les patients du tableau ***SORTIE->elements***, de l'indice *SORTIE->debut* à l'indice *SORTIE->fin*, où ils sont classés par ordre de sortie.
On affichera en plus des autres informations la durée de séjour des patients à l'hôpital : elle est calculée comme la différence entre l'instant de sortie et l'instant d'entrée des patients. Pour chaque adresse *ppatient*, on obtiendra la durée de séjour en appelant ***diffTemps****(ppatient->arrivee,ppatient->sortie)*.

```
Ordre de sortie :

              Le 10/10/2020 :
         1 - 11:55 : Patient   1 (Adrienne Kepoura). Temps passe aux urgences : 54 minutes.
         2 - 13:54 : Patient   3 (Josephine Japy). Temps passe aux urgences : 50 minutes.
         3 - 14:17 : Patient   2 (Ossama Bienaidet). Temps passe aux urgences : 2h15min.
         4 - 15:00 : Patient   4 (Farah Paupierre). Temps passe aux urgences : 57 minutes.
         5 - 17:06 : Patient   5 (Bruce Lee). Temps passe aux urgences : 2h07min.
         6 - 18:08 : Patient   7 (Daniel Keown). Temps passe aux urgences : 1h10min.
         7 - 18:21 : Patient   6 (Marcus Aurelius). Temps passe aux urgences : 2h18min.
         8 - 19:08 : Patient   8 (Julie Dante-Pansey). Temps passe aux urgences : 1h10min.
         9 - 20:54 : Patient   9 (Otto Graaf). Temps passe aux urgences : 1h39min.
        10 - 21:40 : Patient  10 (Omer Dilpleux). Temps passe aux urgences : 1h48min.

Les patients ne sont pas arrives et repartis des urgences dans le meme ordre.
```

On compare chaque adresse de *SORTIE->elements[i]* à *liste_patients[i]*. Si, pour un même indice *i*, les deux adresses tirées sont différentes, un booléen *meme_ordre* initialisé à *faux* prend la valeur *vrai*.
A la fin, un message dépendant de la valeur de *meme_ordre* est affiché pour indiquer si les patients sont entrés et sortis dans le même ordre ou non.

Ordre de passage des patients dans chaque service - commande *'o'* - *(main.c, 1372-1406)*
Chaque salle *S* de type (*salle_even**) possède un champ **S->*ordre***, un tableau d'entiers.
A l'issue du flux de patients, il est de taille *S->indice* et contient les ***id*** de tous les patients entrés en *S*, dans l'ordre.

Pour afficher l'ordre de passage des patients dans chaque service, on procède comme suit :

- Balayage du tableau *tableau_salles_even* contenant les 5 salles d'événement. Pour chaque salle *S*...
  - Affichage de *S->message*, en transformant la première lettre en majuscules.
    - Si *S->indice != 0* : Pour chaque ***id*** de *S->ordre[0]* à *S->ordre[S->indice]*...
      - Appel de la fonction ***id_getPatient****(id,liste_patients,nombre_patients)* pour récupérer l'adresse ***ppatient*** du patient d'ID *id*.
      - Affichage d'un message : «Patient »+ ***id*** +«(»+ *ppatient->**prenom*** + *ppatient->**nom*** +«)»
    - Sinon : affichage d'un **message indiquant qu'aucun patient n'est entré** dans la salle.

Exemple : affichage de l'ordre d'entrée des patients dans deux salles.

```
Examen complementaire :
       Patient   2 (Ossama Bienaidet)
       Patient   5 (Bruce Lee)
       Patient   6 (Marcus Aurelius)
       Patient   9 (Otto Graaf)
       Patient  10 (Omer Dilpleux)

Examen (circuit court) :
       Patient   1 (Adrienne Kepoura)
       Patient   4 (Farah Paupierre)
       Patient   7 (Daniel Keown)
       Patient   8 (Julie Dante-Pansey)
```

Temps d'attente moyen de chaque salle d'attente - commande *'s'* - *(main.c, 1141-1200)*
Chaque salle S de type (*salle_dattente**) possède un champ **S->*temps_attente***. Il s'agit d'un réel initialisé à 0 à la création de la salle, auquel on ajoute **1** pour chaque patient présent dans la salle à chaque incrémentation de ***temps_simulation***.
Ainsi, à l'issue du flux de patients, il est égal au temps d'attente total exprimé en minutes.
On parcourt alors *tableau_salles_dattente :* pour chaque salle d'attente *S…*

Si *S->temps_dattente* est non nul …

On divise *S->temps_attente* (valant alors la durée totale d'attente) par *S->QUEUE->capacite*, le nombre total de patients entrés dans la salle pour obtenir la durée moyenne d'attente.
On remplace *S->temps_attente* par le résultat obtenu.

Ensuite, lorsque l'utilisateur exécute la commande *'s'* :

On parcourt *tableau_salles_dattente :* pour chaque salle d'attente *S…*

On convertit *S->temps_attente* en jours, heures, minutes et secondes, et on affiche le résultat obtenu de manière optimisée (en évitant les fautes d'accord et l'affichage de zéros non significatifs). Un message personnalisé est affiché si le temps d'attente est nul.

Le nombre de secondes devient significatif dans cette situation, c'est pour cette raison qu'on utilise un réel pour stocker le temps d'attente moyen de la salle d'attente : les chiffres après la virgule peuvent être convertis en secondes.
Exemple de résultat :

```
TEMPS D'ATTENTE MOYEN DANS CHAQUE SALLE D'ATTENTE
_____________________________________________________________________________
       Salle d'attente 1 : 1 minute et 6 secondes.
       Salle d'attente 2 : 3 minutes et 6 secondes.
       Salle d'attente 3 : 10 minutes et 9 secondes.
       Salle d'attente 4 : 16 minutes et 30 secondes.
```

Remarque : le temps d'attente moyen des salles d'attente est sensible à de nombreux paramètres.

- *L'écart de la plage horaire d'arrivée des patients : plus il est petit, plus les durées seront élevées.*
- *Le nombre de patients générés : plus il est grand, plus les temps durées seront élevées.*
- *La durée des tâches. Plus elle est longue pour une salle, plus la durée de séjour dans la salle d'attente la précédant est importante. En revanche, les salles d'attente suivantes verront la leur diminuer.*
- *La distribution des circuits et le nombre de box des salles spécifiques à chaque circuit. Ces paramètres influent sur les durées de séjour dans les salles d'attente 3 et 4.*

En faisant varier ces paramètres, on pourra obtenir des temps d'attente très élevés ou très réduits, voire nuls.
Le temps d'attente moyen est une donnée particulièrement importante dans le contexte d'un service d'urgences, où des vies sont en jeu.

Temps passé aux urgences : - commande *'t'* - *(main.c, 1202-1267)*
Le temps total passé aux urgences par un patient d'adresse *ppatient* est calculé comme l'écart **entre les deux instants** *ppatient->sortie* et *ppatient->arrivee*.
Pour établir différentes statistiques relatives au temps passé aux urgences, on créée un tableau de valeurs temporelles ***durees*** de taille le nombre total de patients *nombre_patients*.

Quatre variables de type *temps* sont également utilisées :
temps_total, qui stockera la somme de toutes les durées de séjour. On l'initialise à 0.
tmin et ***tmax***, qui enregistreront respectivement les durées de séjour minimum et maximum, associées aux patients d'adresses ***ppatientmin*** et ***ppatientmax***. On les initialise avec les durées de séjour du patient *liste_patients[0]*.
duree, une variable intermédiaire qui servira à stocker une durée de séjour.

On balaie ensuite *liste_patients* : pour chaque adresse *ppatient* …

- On affecte à *duree* la valeur *diffTemps(ppartient->sortie,ppatient->arrivee)*, càd. la durée de séjour de **ppatient.*
- Si *comparerTemps(duree,tmin)==-1* , càd. « si *duree* est inférieure à la durée minimale de séjour trouvée jusqu'alors … »
    - On affecte *tmin = duree,* le nouveau minimum.
    - On affecte *ppatientmin = ppatient*, le patient correspondant.
- Si *comparerTemps(duree,tmax)==1* , càd. « si *duree* est supérieure à la durée maximale de séjour trouvée jusqu'alors … »
    - On affecte *tmax= duree,* le nouveau maximum.
    - On affecte *ppatientmax = ppatient*, le patient correspondant.
- On ajoute *duree* à *temps_total* en appelant ***ajouterTemps****(&temps_total,duree).*
- Enfin, on insère *duree* dans le tableau ***durees***.

A l'issue de ce bloc d'instructions, on trie le tableau *durees* dans l'ordre croissant en appelant ***trierDurees****(durees,nombre_patients)*. Ce tri est nécessaire pour la détermination de la médiane.

Les informations suivantes sont ensuite affichées dans la console :
- La **durée de séjour totale** : ***temps_total***

- La durée de séjour **moyenne** : ***dureeMoyenne****(**t_total,nombre_patients**);*
(Cette procédure, définie dans la source *"temps.c"*, convertit la valeur temporelle donnée par le premier paramètre en un nombre réel de minutes, le divise par le second paramètre, recompose une durée exprimée en *jours, heures, minutes et secondes* et l'affiche de manière optimisée.)

- L'**écart-type** des durées de séjour : ***ecart_type****(durees,nombre_patients)*
(Cette fonction, définie dans la source *"temps.c"*, calcule à partir du tableau de durées passé en premier paramètre et par son effectif passé en second paramètre, la somme des carrés des écarts de chaque durée du tableau à la durée moyenne, le tout divisé par l'effectif, puis retourne la racine carrée du résultat : un nombre réel de minutes.)

- La durée **médiane** : ***durees****[* $\frac{nombre\_patients + 1}{2} - 1$ *]*
(Le tableau *durees* a préalablement été trié dans l'ordre croissant.
Si son effectif est impair, la médiane de la série correspond donc à la case du milieu, d'indice (*effectif+1)/2 – 1.*
Exemple : case d'indice 4 (5e case) pour un effectif de 9 valeurs.
Si son effectif est pair, la médiane de la série correspond également à la case d'indice (*effectif+1)/2 – 1,* puisqu'on réalise une division entière : le résultat est donc arrondi à l'entier inférieur.
Exemple : case d'indice 4 (5e case) pour un effectif de 10 valeurs, car
E(*(10+1)/2)-1 = E(5,5)-1 = 4.*

- Les **durées extrêmes** *tmin* et *tmax,* ainsi que l'id, le prénom et le nom des **patients** ***ppatientmin* et ***ppatientmax* **correspondants** (*ignoré si un seul patient a été créé simulé*).

- La **durée de séjour** et **les informations** désignant chaque patient. *(A partir de 7 patients simulés, cette fonctionnalité s'active uniquement si l'utilisateur le souhaite, pour éviter de polluer la console.)*

Exemple de résultat de la commande 't' :

```
TEMPS PASSE AUX URGENCES :
_____________________________________________________________________________

        Total : 22 heures et 30 minutes.
        Moyen : 2 heures et 15 minutes.
        Ecart-type (minutes) : 53.55
        Median : 2 heures et 3 minutes.
        Minimal : Patient 2 (Amar Dy) : 1 heure et 1 minute.
        Maximal : Patient 7 (Till Lindemann) : 4 heures et 4 minutes.

Voulez-vous afficher le temps passe aux urgences par chaque patient ? (o/n) :
```

Statistiques relatives à la durée de chaque tâche : - commande *'d'* - *(main.c, 1307-1370)*
Chaque S de type (*salle_even**) construit, au cours de la gestion du flux, une série statistique à partir de ses champs ***S->tab_durees***, un tableau contenant les durées des tâches réalisées en *S*, et ***S->indice***, un entier donnant les positions d'insertion des valeurs dans les tableaux de données, et valant, à terme, leur effectif.
On utilisera également le champ ***S->tab_id_patients*** , un tableau d'entiers contenant les ***id*** de chaque patient entré dans la salle.

A l'issue du flux de patients, pour chaque salle *S* de *tableau_salles_even*, on appelle ***trierStats****(S)*, fonction définie dans la source "*salles.c*", qui trie conjointement *S->tab_durees* et *S->tab_id_patients* par ordre de durées croissantes.
Le tri est réalisé de manière à ce que les id des patients du second tableau correspondent toujours à leurs durées de séjour consignées dans le premier tableau, pour les mêmes indices.

Lorsque l'utilisateur exécute la commande *'d'*, sont affichées, pour chaque salle *S* de *tableau_salles_even* ayant accueilli plus d'un patient, les informations suivantes :

- Un message désignant la salle concernée.

- La **durée totale** de séjour dans la salle : *temps* ***duree_totale = dureeTotale****(S)* ;
  (Cette fonction de "*salles.c*" fait la somme des durées de *S->tab_durees* et retourne un résultat de type *temps*).

- La **moyenne** de la série statistique, donnée par *dureeMoyenne(duree_totale,S->indice)* ;
  (Procédure détaillée à la page précédente.)

- Son **écart-type** : *ecart_type(S->tab_durees,S->indice)* ;
  (Fonction détaillée à la page précédente.)

- Sa valeur **médiane** : *S->tab_durees[* $\frac{S->indice+1}{2} - 1$*]*.

- Ses **valeurs extrêmes** : *S->tab_durees[0]* (**durée min.**) et *S->tab_durees[nombre_patients]* (**durée max.**), si elles sont différentes.
  On affichera également les **informations des patients associés** grâce à la correspondance des indices de *S->tab_durees* et *S->tab_id_patients.*

- **Toutes les durées de séjour, dans l'ordre croissant**, et les informations des patients qui leur sont associés.

```
STATISTIQUES DES TACHES REALISEES
_____________________________________________________________________________

Enregistrement :
        Duree totale : 15 minutes.
        Duree moyenne : 3 minutes.
        Ecart-type (minutes) : 0.89
        Duree mediane : 3 minutes.
        Duree de la tache pour chaque patient :
                Duree minimale : Patient   1 : (Pat Akrep) : 2 minutes.
                Duree maximale : Patient   3 : (Mario Nett) : 4 minutes.
                Patient   1 (Pat Akrep) : 2 minutes.
                Patient   5 (Julien Allemand) : 2 minutes.
                Patient   2 (Jean-Neymar Descoursadistance) : 3 minutes.
                Patient   4 (Marc Assaint) : 4 minutes.
                Patient   3 (Mario Nett) : 4 minutes.
Examen IOA :
        Duree totale : 31 minutes.
        Duree moyenne : 6 minutes et 12 secondes.
        Ecart-type (minutes) : 0.98
        Duree mediane : 6 minutes.
        Duree de la tache pour chaque patient :
                Duree minimale : Patient   4 : (Marc Assaint) : 5 minutes.
                Duree maximale : Patient   1 : (Pat Akrep) : 8 minutes.
                Patient   4 (Marc Assaint) : 5 minutes.
                Patient   3 (Mario Nett) : 6 minutes.
                Patient   2 (Jean-Neymar Descoursadistance) : 6 minutes.
                Patient   5 (Julien Allemand) : 6 minutes.
                Patient   1 (Pat Akrep) : 8 minutes.
Examen (circuit long) :
        Duree totale : 3 heures et 31 minutes.
        Duree moyenne : 52 minutes et 45 secondes.
        Ecart-type (minutes) : 9.18
        Duree mediane : 56 minutes.
        Duree de la tache pour chaque patient :
                Duree minimale : Patient   2 : (Jean-Neymar Descoursadistance) :
                Duree maximale : Patient   5 : (Julien Allemand) : 59 minutes.
                Patient   2 (Jean-Neymar Descoursadistance) : 37 minutes.
                Patient   3 (Mario Nett) : 56 minutes.
                Patient   4 (Marc Assaint) : 59 minutes.
                Patient   5 (Julien Allemand) : 59 minutes.
Examen complementaire :
        Duree totale : 3 heures et 25 minutes.
        Duree moyenne : 1 heure, 8 minutes et 19 secondes.
        Ecart-type (minutes) : 16.21
        Duree mediane : 1 heure et 15 minutes.
        Duree de la tache pour chaque patient :
                Duree minimale : Patient   4 : (Marc Assaint) : 46 minutes.
                Duree maximale : Patient   3 : (Mario Nett) : 1 heure et 24 minu
                Patient   4 (Marc Assaint) : 46 minutes.
                Patient   2 (Jean-Neymar Descoursadistance) : 1 heure et 15 minu
                Patient   3 (Mario Nett) : 1 heure et 24 minutes.
Examen (circuit court) :
        Seul le patient   1 (Pat Akrep) est passe par la salle d'examens (circui
```

Pourcentage et liste des patients ayant emprunté chaque circuit - commande *'p'* - *(main.c, 1269-1305)*
Les pourcentages de patients ayant emprunté chaque circuit sont calculés à partir des compteurs suivants :
nombre_patients (total de patients simulés ; ne peut pas être nul, sinon, la simulation ne se lance pas),
nb_patients_cc (nombre de patients orientés en circuit court),
nb_patients_cl (nombre de patients orientés en circuit long avec ou sans examen complémentaire),
nb_patients_cl_cpt (nombre de patients admis en salle d'examen complémentaire).

Un entier ***pourcentage*** est déclaré et initialisé à 0.
On exécute ensuite les instructions suivantes :

//CIRCUIT COURT
pourcentage = 100**nb_patients_cc/nombre_patients* ; (Division entière, arrondie à l'entier inférieur.)
Si *pourcentage* est non nul :
- Affichage d'un message « Circuit COURT : » + ***pourcentage*** + « % des patients.»
- Pour chaque adresse *ppatient* de *liste_patients*...
  - Si *ppatient->circuit == COURT* :
    - Affichage des informations (*id*, *prenom*, *nom*) du patient **ppatient.*

//CIRCUIT LONG
pourcentage = 100 – *pourcentage* ;
Si *pourcentage* est non nul :
- Affichage d'un message « Circuit LONG : » + ***pourcentage*** + « % des patients.»
- Pour chaque adresse *ppatient* de *liste_patients*...
  - Si *ppatient->circuit == LONG* :
    - Affichage des informations (*id*, *prenom*, *nom*) du patient **ppatient.*

//EXAMEN COMPLEMENTAIRE
- ***pourcentage*** = 100**nb_patients_cl_cpt/nb_patients_cl* ;
- Affichage d'un message ***pourcentage*** + « % ont subi des examens complémentaires »
- ***pourcentage*** = 100**nb_patients_cl_cpt/nombre_patients* ;
- Affichage d'un message « (» + ***pourcentage*** + « % du total)»
- Pour chaque adresse *ppatient* de *liste_patients*...
  - Si *ppatient->circuit == LONG_EXAM_CPT*:
    - Affichage des informations (*id*, *prenom*, *nom*) du patient **ppatient.*

Exemple de résultat de la commande 'p' :

```
STATISTIQUES RELATIVES AUX CIRCUITS
________________________________________________________________________________

Circuit COURT : 40% des patients.
        Patient   3 : (Fabrice Chapuis)
        Patient   9 : (Marinette Lucroy)
        Patient  11 : (Pat Akrep)
        Patient  12 : (Elaine Billington)
        Patient  13 : (Chantal Akoral)
        Patient  14 : (Clement Tine)
        Patient  18 : (Theo Courant)
        Patient  21 : (Francis Kart)
        Patient  22 : (Camille Loiseau)

Circuit LONG : 60% des patients.
        Patient   4 : (William Shakespeare)
        Patient   7 : (Cyprien Madec)
        84% ont subi des examens complementaires (50% du total).
                Patient   1 : (Patt Tachon)
                Patient   2 : (Nikita Kroushchtiov)
                Patient   5 : (Achille Rossignol)
                Patient   6 : (Ali Mentation)
                Patient   8 : (Charles Otto-Fraaz)
                Patient  10 : (Ura Koyama)
                Patient  15 : (Noemie Merlant)
                Patient  16 : (Farid Hannibal)
                Patient  17 : (Gerard Menvussat)
                Patient  19 : (Antonia Gerena)
                Patient  20 : (Paul Ochon)
```

III – Problèmes rencontrés

Et solutions mises en place

3.1 – Problèmes liés à la gestion du temps

Au cours de la conception de notre programme, nous avons été confrontés à de nombreux problèmes liés à la **gestion du temps.**
Nous n'avons pas pensé à toutes les subtilités dès le début, elles se sont révélées au fur et à mesure.
Voici une liste non exhaustive des différents problèmes relatifs à la gestion du temps que nous avons eus à résoudre :

- **Modes « unité fixée » :** au début de la conception de la simulation, la saisie des patients n'était pas automatisée. Très vite, il nous est apparu nécessaire de permettre à l'utilisateur de **fixer un jour ou un mois d'arrivée des patients**, pour ne pas avoir à saisir la date à chaque nouveau patient. Il a alors fallu ajouter de nombreuses fonctions de saisie spécifiques à chaque mode :
Par exemple, en mode mois fixé, l'utilisateur saisit d'abord un mois et une année, puis, pour chaque patient, un jour et une heure d'arrivée. Il fallait donc créer une fonction permettant la saisie d'un mois et d'une année uniquement, et une autre permettant la saisie d'un jour et d'une heure uniquement.
Il a également fallu permettre à l'utilisateur de désactiver cette option pendant la création des patients.
Plus tard, lorsque nous avons automatisé la création des patients, le procédé mis en place nous a posé de nombreux problèmes. Il a fallu adapter à chaque combinaison d'options (*ex : mois fixé et jour non fixé …*) les procédés d'acquisition et de vérification des plages horaires. Par exemple, si l'utilisateur a fixé un jour ou un mois, et qu'il choisit le 31/12/2020 à 23h59, il se retrouve bloqué lorsqu'il faut saisir l'instant de fin de la plage horaire, puisque celui-ci doit appartenir au même jour ou au même mois, mais être postérieur à l'instant de début de la plage horaire.

- **Saisie de valeurs temporelles :** Saisir des valeurs temporelles de manière sécurisée s'est révélé plus complexe que nous l'avions imaginé. La gestion des différents formats autorisés pour chaque type de saisie nous a demandé beaucoup de travail. La vérification de la validité d'une date nous a également donné du fil à retordre, étant donné la complexité du calendrier grégorien (dernier jour dépendant du mois, pas de 29 février les années non-bissextiles …).

- **Opérations sur les valeurs temporelles :** Là encore, nous avons sous-estimé la difficulté de la définition d'opérations permettant de calculer la somme ou la différence de deux valeurs temporelles. En effet, il a fallu créer nos propres fonctions de comparaison et de test (nullité du temps, valeur maximale d'une unité …).
Par exemple, ajouter une minute à une valeur de type *temps* tout en conservant un format correct impose de tester l'état de chaque champ : par exemple, si l'on veut ajouter une minute à l'instant 31/12/2020, 23:59, il faut incrémenter la valeur de l'année et remettre les autres champs à leur valeur minimale pour obtenir 01/01/2021, 00:00.

3.2 – Problèmes liés à l'utilisation des salles

Lorsque nous avons conçu notre algorithme de gestion du flux de patients, nous n'avons pas pensé à certaines subtilités complexifiant sa mise en œuvre en langage C. Comme ils nous sont apparus au fur et à mesure du développement du programme, nous n'avons pas pu repenser l'intégralité de sa structure et il nous a fallu mettre en place des solutions qui compliquaient et alourdissaient le programme.

- **Approche générale de la gestion du flux :** Nous avons, dès le début, souhaité mettre au point une technique flexible, que l'on pourrait adapter dynamiquement si les consignes du sujet était modifiées, en ajoutant des salles par exemple. En effet, en programmation, il est coutume de concevoir des programmes modelables et adaptables.
Or, avec du recul, cette approche ne s'est pas révélée idéale pour le développement de notre programme : étant donné le petit nombre de salles et leurs différents comportements, il aurait peut-être été préférable de gérer le flux **en considérant chaque salle individuellement** pour simplifier la structure et la compréhension du programme **plutôt que d'essayer d'appliquer une technique généralisée à toutes les salles.**

Au final, nous avons de toutes façons été obligés de considérer différents cas (*Salle avec ou sans box ? Salle suivante est une salle d'attente ou non ?...*) et de les traiter séparément – l'architecture du programme et sa clarté en pâtissent.
Nous retiendrons donc, à l'avenir, qu'il est important de choisir l'approche la plus appropriée au fonctionnalités devant être remplies par le programme plutôt que de **chercher à tout prix à généraliser les traitements.**

- Liens avec les salles adjacentes : Dans notre programme, les salles d'attente et les salles d'événement sont représentées par des types différents. Par conséquent, il n'a pas été possible, comme nous voulions le faire au début, d'enregistrer dans chaque salle d'événement des liens vers les salles adjacentes.
Par exemple, la salle d'examen complémentaire n'est pas précédée ni suivie d'une salle d'attente.
Au début, nous avons essayé de désigner la salle suivante et la salle précédente par des adresses de type (*void**). Cependant, **le langage C ne prend pas en charge la généricité** : pour accéder aux champs de la salle adressée, il faut que son type soit connu du compilateur. De toutes façons, les deux structures n'ayant pas le même nombre de champs, **la taille** des pointeurs de type (*salle_even**) et (*salle_dattente**) **n'était pas la même.**
Une possibilité aurait été de créer deux champs pour chaque type de salles adjacentes, mais cette solution était poussive et complexifiait les traitements.
Une autre solution aurait été de **représenter toutes les salles par un même type**, et de n'utiliser que les champs utiles à chaque catégorie de salle : c'est ce que nous avons fait avec les box, qui sont représentés comme des sous-salles d'événement. Mais cette méthode est aussi particulièrement lourde et impose de **réserver de la mémoire inutilement.**

- Rebouclage d'une étape du flux : Comme les transferts réalisés par une salle influent sur le comportement des autres salles, il s'est avéré nécessaire de **retenter de faire progresser le flux après chaque transfert**, avant d'incrémenter le temps virtuel de la simulation.
Par exemple, la salle d'enregistrement prend la main sur le contrôle du flux après la salle d'examen IOA.
Imaginons que la salle d'examen IOA soit libre, mais qu'aucun patient ne soit encore présent dans la salle d'attente 2. Aucun transfert n'a lieu au niveau de cette salle.
Ensuite, c'est au tour de la salle d'enregistrement de modifier le flux de patients : imaginons qu'elle soit occupée et que son instant de libération soit atteint. Elle va alors envoyer son patient en salle d'attente 2.
Or, la salle d'examen IOA a déjà terminé son "tour" et ne prendra pas le patient envoyé en salle d'attente 2 avant que le temps soit incrémenté.
Pour remédier à ce problème, il a fallu utiliser la variable booléenne *modification* afin de détecter la survenue d'un transfert. Tant qu'un **transfert est détecté**, *modification* reprend la valeur *faux* et **toutes les étapes du flux sont répétées**. Le temps virtuel n'est incrémenté qu'une fois que les salles ont atteint un **état "stable".**
Cette complication aurait pu être évitée en **incluant les secondes** dans la structure *temps* : l'incrémentation du temps serait alors d'une seconde et l'on pourrait **négliger le temps mis par une salle à réagir** à la modification du flux.

- Passage à la salle d'examen complémentaire : La salle d'examen long est directement connectée à la salle d'examen complémentaire : il n'y a pas de salle d'attente les séparant. Par conséquent, si un patient devant subir un examen complémentaire a terminé son examen principal, il doit rester dans le box qu'il occupe jusqu'à ce qu'une place se libère dans la salle d'examen complémentaire.
Une solution, pour ne pas bloquer inutilement une place, aurait pu être de créer une file d'attente servant de « tampon » entre les deux salles.
Comme le sujet n'en mentionnait pas l'existence, nous avons décidé considérer que les patients bloquaient justement l'accès au box occupé, jusqu'à ce qu'une place se libère en salle d'examen complémentaire. La durée de l'examen principal ne s'en retrouve pas faussée, puisque les durées des tâches sont enregistrées à leur génération et non à la libération de la salle.
Il faut alors faire libérer les box de la salle d'examen principale par la salle d'examen complémentaire, s'ils contiennent des patients qui y sont orientés.
En pratique, les deux salles d'événement du circuit long sont traitées comme des cas particuliers : les patients de la salle d'examen principale ne sont pas envoyés à la file de sortie s'ils sont admis en salle d'examen complémentaire ; Les box de cette dernière salle sont remplis en balayant ceux de la salle précédente à la recherche de patients admis.

3.3 – Problèmes liés à l'affichage

Les différentes conventions adoptées pour l'affichage des différents résultats nous ont également amenés à complexifier le programme.

- Affichage unique de la date et de l'heure : Pour aérer l'affichage et faciliter la lecture des événements, nous avons décidé que la date et l'heure ne seraient affichées qu'à chaque changement. Cette convention complexifie bien plus le code qu'il n'y paraît : elle impose d'utiliser des variables pour stocker les valeurs temporelles affichées en dernier, ou des booléens qui sont désactivés une fois « consommés ».
Par exemple, pour afficher l'heure et sauter une ligne en cas de changement de minute, on utilise un booléen remis à *vrai* à chaque incrémentation de *temps_simulation* et qui devient *faux* dès le premier affichage.
Pour ne pas afficher la date les jours où aucun événement ne se produit, il faut en plus « prédire » le jour où aura lieu le prochain événement à chaque fois qu'un événement est planifié. La date ne sera alors affichée que si le jour prédit est atteint et qu'elle est différente de la dernière date affichée. *(voir partie 2.6.d – p29).*
Cette convention complique également l'affichage de certaines statistiques (comme l'ordre d'arrivée des patients) et l'organisation de l'historique : en effet, il faut écrire le contenu de l'historique de manière à ce qu'à chaque événement soit associée la date et l'heure de sa survenue ; Il faut ensuite le lire de manière à n'afficher la date et l'heure que lorsqu'elles changent d'une ligne à l'autre. *(voir partie 2.7– p30).*

- Equilibrage des lignes : Une fois la création des patients automatisée, nous avons essayé de simuler des grands nombres de patients. Or, nous avons remarqué que l'affichage devenait très inesthétique lorsque le nombre de chiffres de l'id des patients changeait : le décalage entre les différentes parties du message pouvait changer d'une ligne à l'autre, ce qui rendait la lecture désagréable.
Pour remédier à ce problème, nous avons décidé que les *id* de patients seraient toujours affichés au format *"%3d"*, c'est-à-dire, qu'ils prennent la place de 3 caractères quelque soit leur nombre de chiffres.
Ainsi, pour des nombres de patients allant jusqu'à 999, le décalage est toujours le même. Au-dessus, de ce nombre, le décalage n'est plus choquant.
Cependant, ce décalage augmente l'espace devant être réservé en mémoire pour les différentes chaînes de caractères et paraît trop important pour des petits nombres de patients. Illustration : `Le patient   6`
Une solution serait de choisir le décalage des *id* en fonction du nombre de chiffres de *nombre_patients*, mais cela complexifierait le code en de nombreux endroits.

- Défilement des résultats des commandes : Afin que l'utilisateur puisse saisir autant de commandes qu'il le souhaite, les menus sont réaffichés instantanément après chaque résultat.
Or, certaines commandes, comme celle qui affiche l'historique, peuvent écrire un résultat très long dans la console. L'utilisateur doit alors faire défiler vers le haut le texte de la console jusqu'à revenir au début du résultat affiché pour le consulter.
Malheureusement, nous n'avons pas trouvé de solution simple à ce problème, dont le confort d'utilisation du programme pâtit. Ajouter un délai avant l'affichage de chaque ligne n'est pas envisageable et nous ne connaissons pas de moyen d'empêcher le défilement vertical de la console lorsque plusieurs lignes sont écrites.

3.4 – Inclusion des headers

Le code de notre programme est divisé en **plusieurs fichiers-sources et en-têtes.**
L'instruction *#include "header.h"* est utilisée pour intégrer les définitions d'une en-tête à un autre fichier.
Nous avons rencontré un problème lié à l'assemblage des fichiers headers : lorsqu'on inclut dans chaque fichier toutes les en-tête nécessaires à son fonctionnement, le compilateur peut détecter de **multiples définitions d'un même élément.** L'erreur est générée par l'**inclusion du même header dans deux fichiers différents qui sont ensuite assemblés.**
Or, plusieurs fichiers nécessitent la connaissance des types et fonctions définis dans de mêmes en-têtes : par exemple, *aleatoire.h* et *salles.h* doivent tous deux inclure *temps.h*.
Pour éviter l'erreur de définition multiple, nous n'avons pas trouvé d'autre moyen que d'inclure les headers les uns dans les autres, du plus petit niveau d'inclusion au plus haut, à la manière de poupées russes.
L'imbrication suit le schéma suivant : *main.c* ← *salles.h* ← *queue.h* ← *patient.h* ← *aleatoire.h* ← *temps.h.*

3.5 – Problèmes relatifs à la génération de données suivant des lois aléatoires

- Génération aléatoire des durées des tâches : c'est la fonctionnalité que nous avons eu le plus de mal à mettre en place. Nous n'avons, au départ, pas trouvé d'aide concernant la génération d'échantillons de données suivant une loi normale sur internet : nous avons donc conçu une méthode « maison ». Au début, nous avions développé une méthode qui permettait d'obtenir une répartition satisfaisante des résultats, mais qui ne fonctionnait plus si l'on changeait l'espérance et l'écart utilisés.

Après de nombreux changements, nous avons fini par mettre en place une méthode consistant à **générer une probabilité et à approcher son antécédent par la fonction de répartition** d'une loi normale de paramètres connus. Les résultats obtenus nous paraissent satisfaisants, mais ils ne suivent réellement une loi normale que pour un très **grand nombre de patients** et leur répartition reste très **influencée par la composante aléatoire** – de fait, il est possible, pour certaines simulations, d'obtenir un résultat insatisfaisant.

De plus, cette méthode fait, pour chaque durée, l'**approximation de plusieurs intégrales** et trouve l'antécédent de la probabilité générée au moyen d'une **recherche dichotomique** : elle est donc **particulièrement gourmande en ressources.** *(voir 2.5 – p19).*

Au moment de la rédaction du rapport, nous avons communiqué avec d'autres étudiants ayant choisi le même sujet que nous, et ils nous ont expliqué avoir utilisé la **méthode de Box-Muller**, beaucoup plus simple à mettre en œuvre et bien moins gourmande en ressources. **Nous ne l'avons cependant pas implémentée** dans notre projet, celui-ci ayant déjà atteint un stade quasi-terminal de son développement, et le recours à une aide extérieure étant interdit.

Au final, si la méthode utilisée dans notre programme est loin d'être optimale, elle remplit tout de même la tâche de générer des **durées réalistes** dans une **plage de durées délimitée par le concepteur.** De plus, à partir d'une vingtaine de patients générés, **l'espérance et l'écart-type calculés** pour la durée de chaque tâche **sont proches des valeurs attendues :** $\mu(calculée) \approx esperance$ et $\sigma(calculé) \approx ecart/3$.

- Génération aléatoire des heures d'arrivée : *(voir 2.3.b – p14) ;*

Nous n'avons, au départ, pas eu de problème particulier pour implémenter cette fonctionnalité.

Rapidement, nous avons regretté la régularité des intervalles entre les instants d'arrivée des patients. **Souhaitant rendre la répartition des durées plus irrégulière, et différente à chaque lancement,** nous avons décidé d'ajouter une composante aléatoire capable de **modifier l'instant choisi pour l'arrivée d'un patient.** Il nous a alors fallu beaucoup de temps pour adapter son influence d'une manière satisfaisante quels que soient la plage horaire et le nombre de patients créés.

De plus, nous avons remarqué que la génération des patients prenait une tournure inattendue pour des effectifs supérieurs à 22 : en effet, le calcul des probabilités des événements suivant la loi de Poisson fait intervenir un calcul factoriel, qui **devenait imprécis à partir de *23!*.**

Il a alors fallu adapter le processus en considérant des paquets de 22 patients au maximum, et des portions de plages horaires correspondantes. Ce procédé a toutefois **un impact sur les instants choisis** par rapport à une version capable de répartir les arrivées de n'importe quel nombre de patients ; Pour minimiser l'importance de cet impact, et pour **éviter d'avoir un patron de répartition similaire répété** pour chaque paquet de 22 patients, nous avons modifié notre utilisation de la composante aléatoire.

Ainsi, cette dernière remplit désormais deux fonctions : **rendre plus irrégulière la répartition des patients et éviter la répétition d'un même patron de répartition.**

Enfin, nous avons eu à résoudre deux dilemmes :

Fallait-il permettre à l'utilisateur de **choisir et le nombre de patients et leur plage horaire d'arrivée** pour lui offrir plus de contrôle sur la simulation, ou créer automatiquement une plage horaire d'arrivée à partir du nombre de patients pour lui offrir plus de simplicité d'utilisation et des résultats plus réalistes ? Nous avons opté pour la première option.

Il nous a également fallu choisir entre permettre l'arrivée d'un premier patient dès l'instant de début de la plage horaire, pour rendre plus intuitif le fonctionnement de la simulation au risque de fausser la génération des patients suivant un processus de poisson, **ou déterminer aléatoirement les heures d'arrivée de tous les patients.** (*exemple : l'utilisateur choisit la plage 12h-14h. Le premier patient arrive nécessairement à 12h00).* Nous avons, par souci de rigueur, opté pour la deuxième solution.

- **Attribution de noms aux patients :** Si cette fonctionnalité n'était pas demandée dans le sujet, nous avons toutefois pensé qu'elle serait bienvenue et qu'elle nous permettrait de personnaliser notre projet.
Sa complexité réside dans la manière d'utiliser la liste de noms : nous voulions **éviter** au maximum l'attribution de **deux noms identiques à deux patients différents**. Il a alors fallu mettre en place un système d'**indices identifiant** les lignes et permettant de dresser la liste des nombres encore non attribués.
De plus, nous voulions, dans notre programme, retrouver des combinaisons prénom + nom qui formeraient des jeux de mots ou qui désigneraient des personnes réelles (célébrités, professeurs de NF05, nos propres noms ...). Il a alors fallu **établir la liste de toutes les combinaisons disponibles dans un fichier.** Nous aurions également pu associer chaque prénom à un nom choisi au hasard, mais cela nous aurait empêché d'afficher ces jeux de mots désopilants.
A chaque ajout de combinaisons prénom+nom dans le fichier *liste_noms.txt*, il fallait s'assurer qu'il ne manquait aucun numéro de ligne dans l'ensemble d'entiers compris entre 1 et le nombre de noms indiqué au début du fichier – dans le cas contraire, le programme pouvait rechercher indéfiniment un numéro de ligne inexistant.
Enfin, il a fallu **gérer le cas où le fichier ne pouvait pas être localisé ou lu** par le programme. Nous avons décidé que les patients porteraient alors tous le même nom : « John Doe ».
Pour éviter ce cas, il faut que l'utilisateur s'assure que le **répertoire de travail** (*c'est-à-dire, le dossier où se trouve l'éxécutable si le programme est lancé directement ou depuis un IDE,*
ou le répertoire parcouru si le programme est exécuté depuis un terminal – il faut alors penser à exécuter cd chemin/du/repertoire_de_la_liste_de_noms avant d'ouvrir le programme) **contient bien la liste de noms.**
Une solution pour remédier à ce problème serait de créer un *Makefile* qui copierait la ressource *"liste_noms.txt"* dans le répertoire de compilation, mais il faurait alors imposer l'utilisation de ce fichier lors de la compilation du code.

3.6 – Problèmes de compatibilité

Une fois le programme ayant atteint un stade avancé de son développement, nous avons jugé utile de le **tester sur d'autres machines que les nôtres**, et en particulier sur des machines utilisant **d'autres systèmes d'exploitation.**
Cette vérification a permis de mettre en évidence plusieurs **erreurs de programmation qui n'avaient pas empêché le bon fonctionnement du programme** sur les ordinateurs avec lesquels nous avons conçu le code : en effet, selon le système d'exploitation, le processeur et le compilateur utilisé, le programme peut se comporter différemment.

- La premier problème de compatibilité que nous avons rencontrés s'est révélé lorsque nous avons commencé le développement du programme. Nos deux ordinateurs fonctionnaient sous Windows 10, cependant, nos processeurs n'étaient pas conçus par le même fabricant. A un moment, après avoir programmé une partie différente du projet chacun de notre côté, nous avons mis en commun notre code pour disposer tous les deux de la même version. Le programme compilait sur nos deux ordinateurs, mais il renvoyait une erreur lors de l'accès à la case d'un tableau chez l'un de nous. Cette erreur nous a permis de mettre en évidence une négligence dans le code, qui était tolérée sur l'une de nos machines, mais pas sur l'autre.

- En compilant le programme sous une distribution de Linux, nous avons eu un problème lors de la lecture des historiques, ce qui nous a permis de remarquer que le caractère *EOF* était inscrit à plusieurs reprises à la fin du fichier et qu'**une chaîne de caractères de notre programme n'était pas terminée par le caractère *'\0'*.**
Etrangement, cette négligence n'avait pas empêché le programme de fonctionner comme attendu sur Windows.

- Le programme compilait normalement sur MacOS, cependant, au lancement d'une simulation, nous obtenions l'erreur « *segmentation fault 11* » qui n'avait pas été renvoyée lors de nos tests sur Windows et Linux. Elle nous a permis de repérer une erreur lors de l'allocation dynamique d'un espace mémoire : **nous avions réservé un espace mémoire de taille *(sizeof(patient*)*) au lieu d'un espace mémoire de taille *(sizeof(patient))* pour instancier un patient.**
Sur nos versions de Windows et Linux, cette erreur n'avait pourtant pas empêché le programme de fonctionner comme attendu.

IV - Conclusion

4.1 – Avis rétrospectif sur le travail réalisé

4.1.a - Temps passé :

Temps passé par le binôme pour réaliser les différentes tâches du projet

Total : Environ 104 heures.

Programme : Environ 49h au total.

- ***main***

Environ 25h au total.

- ***gestion des options*** : 2h
- ***création des patients*** : 5h
- ***flux de patients*** **:** 6h
- ***statistiques*****:** 5h
- ***historique*** **:** 2h
- ***optimisation de l'affichage*** : 2h
- ***debuggage et peaufinage*** **:** *3h*

- ***gestion du temps***

Environ 5h au total.

- ***salles et files***

Environ 4h au total.

- ***génération de données suivant des lois aléatoires***

Environ 11h au total.

- ***gestion des durées des tâches*** : 5h *(nous avons dû recommencer plusieurs fois)*

- ***génération des heures d'arrivée des patient s***: 3h
- ***nommage des patients :*** 3h

- ***salles et files***

Environ 4h au total.

Autres travaux liées au projet : 55h

- ***commentaire complet du code :*** Environ 6h au total.
- ***production de la documentation :*** Environ 4h au total.
- ***production du rapport*** `Total : 196 jours, 20 heures et 39 minutes.`(Environ 35h en réalité).
- ***préparation de la présentation :*** Environ 10h au total.

4.1.b – Défauts du programme réalisés :

En portant un regard rétrospectif sur le projet, nous avons remarqué plusieurs maladresses que nous éviterons la prochaine fois que nous devrons concevoir un programme.

- ***main*** **trop chargé :** gérer le flux, l'affichage des menus et la génération des patients à partir de fonctions extérieures au *main* aurait été préférable et aurait rendu la compréhension du code plus aisée.

- **Traitements non optimisés :** l'algorithme de génération des durées est **particulièrement gourmand en ressources,** alors qu'il aurait pu être réalisé plus simplement à l'aide de la méthode de **Box-Muller**.
  Si la différence est à peine perceptible sur une machine moderne et grâce à la rapidité d'exécution des logiciels programmés en C, elle ne serait **pas négligeable** sur des **machines plus anciennes** ou si le projet avait été conçu à l'aide d'un **langage de plus haut niveau** comme le java ou le python.

- **Complexité de la gestion du flux de patients :** il aurait sûrement été préférable de **traiter chaque salle individuellement** plutôt que de chercher à généraliser les traitements. Le code aurait gagné en clarté et en concision.

- **Génération aléatoire des durées :** si nous avons réussi à proposer des solutions aux problèmes rencontrés dans ce domaine, nous regrettons un certain manque de rigueur en terme de **respect des lois de probabilités** dus aux compromis réalisés.

- **Organisation générale du programme :** l'architecture du programme, est, dans son ensemble, confuse, notamment en raison des différents problèmes que nous avons rencontrés et qui nous ont obligés à la repenser au fur et à mesure du développement.
  Certains traitements sont exécutés directement dans la fonction *main*, certains sont exécutés depuis d'autres fonctions, et par endroits, les **différences entre les manières de programmer des deux membres de notre binôme** se font beaucoup ressentir.
  De plus, les solutions mises en place pour représenter les salles et les patients ne sont **pas toujours optimales en terme de puissance de calcul et d'utilisation de la mémoire** : par exemple, **les durées des tâches auraient pu être stockées sous la forme d'un nombre de minutes** plutôt que de valeurs de type *temps*.
  L'expérience que nous avons acquise en réalisant ce projet devrait nous permettre, à l'avenir, de **mieux concevoir l'architecture de nos programmes** et d'**anticiper les problèmes** auxquels nous avons été confrontés.

Ce sont, bien évidemment, les seuls défauts de notre programme !

4.1.c – Limites et avantages du langage C :

Bien que nous ayons tous deux déjà programmé, l'un de nous en langage python et l'autre en java, notre binôme n'avait aucune expérience du langage C préalable à NF05. Travailler sur ce projet nous a permis d'approfondir notre compréhension de la programmation et de découvrir le langage C.
Cet apprentissage nous a permis de mettre en lumière plusieurs avantages et inconvénients du C par rapport aux langages de plus haut niveau que nous savions déjà utiliser :

Inconvénients :

- Certaines structures de données utiles, telles que les files, doivent être entièrement définies par les programmeur, de même que les fonctions permettant de les manipuler, là où, dans des langages de plus haut niveau, les bibliothèques standard peuvent déjà proposer une définition de ces objets.
- Travailler avec des éléments représentant les patients et les salles aurait sûrement été plus simple dans un langage de programmation orienté objet.
- Le traitement des chaînes de caractères est plus complexe en C que dans les langages que nous connaissions précédemment.
- Le C ne prend pas en charge la généricité et ne permet pas de travailler avec des valeurs de type non fixé. Ces fonctionnalités auraient été pratiques pour généraliser le traitement du flux de patients en appelant une même fonction sur toutes les salles.

Avantages :

- La rapidité d'exécution du C présente un avantage certain pour la réalisation de ce type de programmes. Des traitements tels que l'approximation d'une intégrale ou la recherche dichotomique d'un antécédent peuvent être laborieux avec des langages de plus haut niveau.

- Paradoxalement, la possibilité d'utiliser des pointeurs nous a semblé simplifier la programmation.
  Les pointeurs nous sont apparus comme des outils très pratiques et puissants, alors que leur absence dans les langages de plus haut niveau est censée représenter un progrès.

- Le langage C dispose de suffisamment de sucre syntaxique pour simplifier la réalisation de certaines tâches (*exemple : tableau[i] est équivalent et plus simple à écrire que *tableau+i*).
  Il a également la bonne dose de sel syntaxique pour des projets de cette taille : par exemple, l'absence de ';' et de '{}' en python, remplacée par la nécessité de marquer les blocs d'instructions par des indentations, aurait été gênante pour le développement d'un tel programme.

4.1.c

4.1.d – Travail en groupe :

Nous n'avions, ni l'un ni l'autre, jamais travaillé en groupe sur un projet informatique par le passé. Ce projet a été l'occasion pour nous de développer cette compétence.
Au début, nous avancions chacun sur une partie différente du programme avant de rassembler notre code pour former une version commune que nous téléchargions tous les deux.
Cette façon de faire s'est rapidement montrée impratique : comme chacun d'entre nous avait besoin de modifier plusieurs parties du code déjà créé, il était parfois impossible de combiner le travail réalisé par chacun d'entre nous.
Au fur et à mesure que le projet avançait, nous avons perfectionné notre méthode de travail pour collaborer plus efficacement, en utilisant notamment *github* pour déposer nos fichiers, et en consignant chaque modification apportée dans un fichier texte.

4.1.d

4.2 – Perspectives d'amélioration

Génération aléatoire de durées avec paramètres saisis par l'utilisateur : Il pourrait être intéressant d'ajouter une option permettant à l'utilisateur de **saisir l'espérance et l'écart** souhaités **pour générer la durée de chaque tâche**. De la même manière, on pourrait proposer à l'utilisateur que les circuits soient distribués automatiquement mais avec d'autres pourcentages de distributions que ceux du sujet.

Décès d'un patient : Il serait intéressant de proposer à l'utilisateur un option permettant le décès ou le départ prématuré de certains patients en cours de simulation. On choisirait alors les patients concernés et leur heure de décès avant de lancer la simulation, ou on les déterminerait aléatoirement.

Etat des patients : On pourrait assigner aux patients une **valeur dépendante de la gravité de leur état.** En fonction de cette valeur, on pourrait leur donner une certaine **priorité d'accès aux salles d'examen** *(les patients dont le pronostic vital est engagé pourraient par exemple être placés en début de file des salles d'attente)*. On pourrait également **affecter les circuits en fonction de la gravité de l'état des patients.**

File d'attente avant l'examen complémentaire : On pourrait envoyer les patients attendant qu'un box de la salle d'examens complémentaires se libère dans une file d'attente pour éviter qu'ils occupent une place dans la salle d'examen principale.

Conservation de plusieurs historiques : Il pourrait être utile de conserver les historiques de chaque simulation lancée au cours d'une même session d'utilisation du programme. On pourrait alors afficher l'historique d'une simulation précédente.
En ajoutant à l'historique les statistiques des simulations associées, on pourrait même comparer plusieurs simulations entre elles, pour déterminer, par exemple, au cours de laquelle le temps d'attente moyen dans les salles d'attente est le plus court.

Ignorer l'affichage d'une entrée en salle d'attente : On pourrait faire en sorte que l'événement « ...entre dans la salle d'attente ... » **ne soit pas affiché** si un patient ne fait que passer par une salle d'attente, **sans attendre.**
Il faudrait alors que chaque message notifiant l'événement lié à une entrée en salle d'attente soit retenu et affiché en fin de boucle « tant que les salles sont dans un état instable... », sous la condition que la salle d'attente concernée n'est pas redevenue vide.

Heure de pointe : Il serait intéressant de donner à l'utilisateur la possibilité de générer aléatoirement les arrivées des patients en choisissant une heure de pointe aux alentours de laquelle on enregistrerait un pic d'arrivée.
Cette fonctionnalité pourrait être réalisée en modifiant le processus de génération des arrivées des patients : la probabilité d'occurrence de l'événement « *un patient arrive au service d'urgences à l'instant t* », intervenant dans le processus poissonnien, serait alors variable au lieu d'être constante. La probabilité d'occurrence serait calculée comme un polynôme du second degré de variable le temps, de manière à ce que son comportement décrive une courbe parabolique de sommet l'instant correspondant à l'heure de pointe.
On obtiendrait alors un pic d'arrivées à l'instant choisi.

4.3 - Bilan

Ce projet a été l'occasion pour nous d'approfondir plusieurs notions du langage C et d'étoffer notre expérience de la programmation en général.
Il a également été formateur en ce qui concerne le travail collaboratif et l'utilisation d'outils d'aide au développement : github, doxygen…
Dans l'ensemble, nous sommes satisfaits du travail que nous avons fourni, mais reconnaissons avoir été plutôt inefficaces et improductifs : le programme aurait pu être développé plus rapidement.
Les compétences que nous avons développées grâce à cet exercice et les enseignements que nous en avons tirés sauront nous être utiles, à l'avenir, lors de la conception de projets informatiques.
Ils pourraient même nous être salutaires à l'occasion du final de NF05…

Les fichiers constituant le code de notre programme sont intégralement commentés.

Vous trouverez également dans /CODE/doc la documentation du programme, générée à l'aide de doxygen.