Simulation de Bureau de Vote : Conception et Implémentation en C++

AZANDOSSESSI Jaurès
JEAN Karl

April 19, 2024

Abstract

Ce rapport détaille la conception et l'implémentation d'une simulation informatisée d'un bureau de vote en C++. Notre objectif est de proposer un environnement de simulation réaliste et fonctionnel qui reproduise avec précision le processus électoral, de l'enregistrement des électeurs au dépouillement des votes et à la détermination du gagnant. Nous explorons les diverses étapes de développement de notre simulation, en mettant en exergue les choix de conception, les structures de données et les algorithmes utilisés pour le traitement des votes. Les fonctionnalités mises en place, les défis rencontrés et les perspectives d'amélioration sont également abordés. En définitive, notre simulation se présente comme une application concrète des principes de la programmation orientée objet et de la manipulation de données en C++, offrant un outil efficace pour l'étude et l'expérimentation des mécanismes électoraux. Ce projet enrichit non seulement nos compétences en programmation et en algorithmique, mais contribue aussi à une meilleure compréhension des dynamiques électorales.

Introduction

La simulation informatique est un outil puissant pour modéliser et analyser divers phénomènes du monde réel. Dans le domaine de la politique et de la démocratie, la simulation de bureaux de vote revêt une importance particulière. En effet, elle permet d'évaluer l'efficacité des processus électoraux, de tester différentes méthodes de vote et de prédire les résultats des élections.

Dans le cadre de ce projet de programmation orientée C++, nous avons entrepris la conception et l'implémentation d'une simulation de bureau de vote. Notre objectif principal est de créer un système informatique réaliste et fonctionnel qui reproduit fidèlement le déroulement d'une élection. Cette simulation sera capable de gérer l'enregistrement des électeurs, le dépouillement des votes, et la détermination du vainqueur selon les règles du scrutin utilisé.

Au cours de ce rapport, nous détaillerons les différentes étapes de développement de notre simulation, en mettant en lumière les choix de conception, les structures de données utilisées, ainsi que les algorithmes de traitement des votes. Nous discuterons également des fonctionnalités implémentées, des défis rencontrés et des perspectives d'amélioration pour ce projet.

En fin de compte, cette simulation de bureau de vote constitue une application concrète des concepts de programmation orientée objet et de manipulation de données en C++. Elle représente un moyen efficace pour étudier et expérimenter avec les mécanismes électoraux, tout en renforçant nos compétences en programmation et en algorithmique.

Description des Classes

Classe Personne

Description : La classe `Personne` dans l'espace de nommage elections représente un individu participant à un processus électoral, caractérisé par son nom, son prénom, une sensibilité politique sur une échelle de 1 à 10 (de gauche à droite), et un identifiant unique généré automatiquement.

Classe Electeur

Description : La classe `Electeur`, en héritant de `Personne`, incarne un électeur au sein d'une simulation de vote. Elle est équipée d'attributs essentiels au déroulement du vote, tels que la gestion des bulletins et l'enregistrement du choix dans l'isoloir, tout en permettant le suivi du temps passé par l'électeur dans diverses zones du bureau de vote. Les méthodes intégrées facilitent l'exercice du devoir civique de l'électeur : sélection des bulletins à la table de décharge, prise de décision dans l'isoloir basée sur la probabilité et l'écart de sensibilité politique avec les candidats, et enfin, le vote. La classe offre également la possibilité d'éliminer les bulletins inutilisés à la conclusion du processus.

Classe TableDeDecharge :

Description : La classe `TableDeDecharge` représente un élément crucial du processus électoral dans une simulation de vote, où elle gère l'organisation et la distribution des bulletins de vote aux électeurs. Conçue pour simuler l'espace restreint où les électeurs sélectionnent leurs bulletins, triés par candidat, elle intègre une règle stricte d'occupation singulière, assurant qu'un seul électeur à la fois puisse accéder à cet espace. Elle offre également des méthodes pour simuler l'entrée et la sortie des électeurs, le traitement de la file d'attente, et la gestion du temps passé par chaque électeur. En outre, elle permet de visualiser les bulletins et de gérer le flux des électeurs, faisant d'elle un pivot autour duquel s'articule l'expérience de vote dans la simulation.

Classe Bulletins :

Description : La classe `Bulletin` est conçue pour représenter de manière abstraite chaque vote au sein d'une simulation de processus électoral, incarnant le choix d'un électeur envers un candidat spécifique. Au-delà d'une simple association avec un nom de candidat, elle intègre une dimension de sensibilité politique, permettant de nuancer l'acte de vote en fonction des orientations politiques des candidats. Cette conception offre une flexibilité essentielle pour simuler avec précision divers scénarios électoraux, allant du simple dépouillement des voix à des analyses plus complexes basées sur les inclinaisons politiques. En dotant chaque bulletin d'une capacité à refléter non seulement un choix mais aussi une tendance politique, la classe devient un pivot autour duquel s'articulent les mécanismes de simulation, contribuant à une représentation plus riche et nuancée du vote. Elle est ainsi au cœur de la dynamique électorale du projet, facilitant l'exploration de différentes configurations électorales et l'évaluation de leur impact sur les résultats finaux.

Classe Elections :

Description : La classe `Election` sert à modéliser le concept d'une élection dans une simulation ou une application liée à des processus électoraux. Elle encapsule l'essence d'une élection en associant un nom spécifique à l'événement électoral et en gérant une collection de candidats y participant. Cette gestion comprend l'enregistrement des candidats, assurant leur unicité grâce à une structure

de données adaptée, et la capacité d'afficher de manière ordonnée la liste des candidats pour faciliter la consultation par les utilisateurs ou d'autres composants du système. Par cette structuration, `Election` devient un pilier pour la représentation des élections, permettant non seulement de recueillir mais aussi de diffuser les informations relatives aux options disponibles pour les électeurs. En résumant, cette classe fournit un cadre pour la mise en œuvre de la logique électorale, centralisant les données essentielles à la compréhension et à la gestion d'une élection, telles que son intitulé et les acteurs en compétition, facilitant ainsi l'interaction avec le processus électoral dans son ensemble.

Classe Action :

Description : La classe `Action` incarne le concept d'une interface au sein d'un système de simulation électorale, définissant un cadre pour les interactions fondamentales des électeurs avec différents espaces électoraux tels que la table de décharge, l'isoloir, entre autres. En tant que classe abstraite, elle spécifie trois méthodes virtuelles pures, entrer, sortir, et gestionTemps, imposant ainsi aux classes dérivées la mise en œuvre de ces interactions essentielles. Cette abstraction permet d'unifier le comportement des différentes composantes du processus de vote, en garantissant que chaque espace implémente des mécanismes pour accueillir et libérer les électeurs, ainsi que pour gérer le temps passé par ces derniers dans chaque espace.

Classe EspaceIsoloirs :

Description : La classe `EspaceIsoloirs` est conçue pour simuler la gestion et le fonctionnement des isoloirs au sein d'un bureau de vote durant une élection. Elle encapsule toute la logique nécessaire pour maintenir et opérer un ensemble d'isoloirs, permettant d'ajouter des isoloirs, de rechercher un isoloir libre pour un électeur, et de gérer l'entrée d'électeurs dans ces isoloirs. La classe offre des méthodes pour identifier le premier isoloir libre, reflétant ainsi une gestion efficace des ressources disponibles, et pour traiter les électeurs en attente, garantissant que chaque électeur soit dirigé vers un isoloir libre dès que possible. Elle intègre également des mécanismes pour surveiller l'état d'occupation des isoloirs et pour gérer la file d'attente d'électeurs, assurant un flux continu et ordonné au sein du bureau de vote.

Classe Isoloir :

Description : La classe `Isoloir` modélise un espace individuel de vote au sein d'un bureau électoral, où les électeurs exécutent le processus de sélection de leur vote en toute confidentialité. Héritant de la classe abstraite `Action`, elle implémente les interactions essentielles d'entrée et de sortie des électeurs, ainsi que la gestion du temps passé à l'intérieur. La méthode `choisirBulletin` encapsule la logique de sélection du vote, qui peut varier d'un choix éclairé à un vote nul basé sur une probabilité prédéterminée. `Isoloir` fournit un cadre détaillé et fonctionnel pour simuler le passage des électeurs à travers le processus de vote, depuis l'entrée jusqu'à la sélection du bulletin et la sortie, contribuant ainsi à la fidélité et à la complexité de la simulation électorale.

Classe TableDeVote :

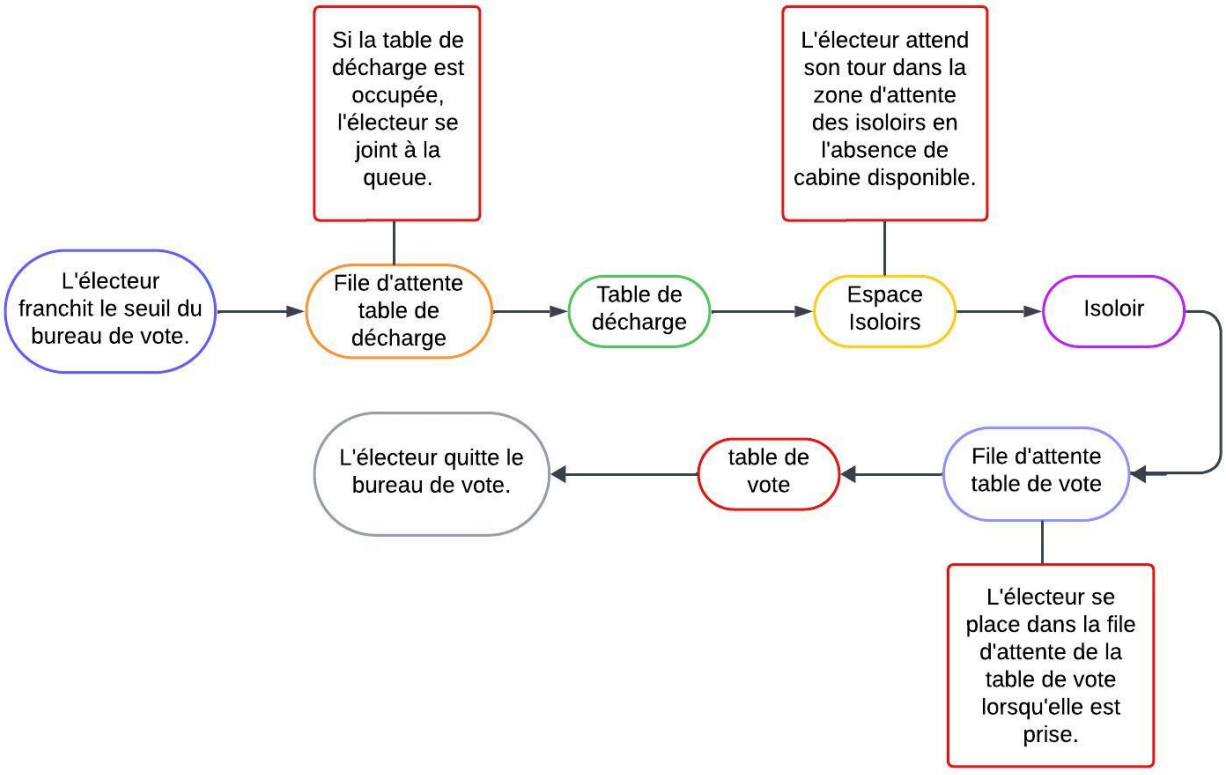
Description : La classe `TableDeVote` incarne le cœur du processus électoral dans une simulation de vote, en simulant l'espace crucial où les électeurs exécutent leur droit de vote sous la supervision du président de l'élection. Elle assure l'unicité de la présence dans cet espace via une gestion rigoureuse de l'occupation et facilite le dépôt des bulletins de vote dans l'urne ainsi que l'émargement, étape finale du processus de vote. En plus de ces opérations, la classe est dotée de fonctions pour la gestion du temps que chaque électeur passe au sein de cet espace, la vérification de la participation via une

liste d'émargement et le traitement des files d'attente, garantissant une expérience de vote ordonnée et conforme aux règles établies. Un aspect fondamental de `TableDeVote` réside dans sa capacité à compiler et fournir les résultats du vote, reflétant ainsi l'issue de l'élection avec précision et intégrité. Elle joue un rôle déterminant dans la simulation, assurant non seulement le bon déroulement du vote mais aussi la fiabilité et la transparence des résultats électoraux.

Classe Bureau :

Description : La classe `Bureau` encapsule le fonctionnement intégral d'un bureau de vote dans le cadre d'une simulation électorale, englobant l'enregistrement des électeurs, la gestion des candidats, et l'orchestration du processus de vote. Elle facilite l'ouverture et la fermeture de l'entrée du bureau pour réguler le flux d'électeurs, gère les interactions avec les espaces clés tels que la table de décharge, la table de vote, et les isoloirs, et assure le bon déroulement du vote jusqu'à la compilation des résultats. En intégrant des méthodes pour afficher les listes électorales, simuler l'élection, et présenter les statistiques finales, `Bureau` sert de noyau central pour la simulation, reflétant fidèlement les complexités et les dynamiques d'un véritable environnement électoral.

Parcours des électeurs dans notre application



Analyse de Complexité et Choix des Structures de Données pour les Listes Électorales

Liste d'Émargement

Pour la **liste d'émargement**, l'utilisation d'un `std::unordered_set` s'est avérée être la solution optimale, principalement grâce à sa fondation sur une table de hachage qui facilite des opérations rapides d'insertion, de suppression, et de test d'appartenance. L'ordonnancement des éléments n'étant pas une priorité pour cette structure, le `std::unordered_set` se montre particulièrement adapté aux besoins de la liste d'émargement. En ce qui concerne la complexité des opérations de test d'appartenance, elles se réalisent en moyenne en **O(1)**. Cependant, il est important de noter que dans des cas extrêmes, notamment en présence de nombreuses collisions de hachage, cette complexité peut se dégrader vers **O(n)**. Cette situation reste néanmoins peu fréquente, grâce à l'efficacité des mécanismes de gestion des collisions intégrés à la structure de la table de hachage.

Liste Électorale

Pour la gestion de la **liste électorale**, la structure `std::set` s'est révélée être la plus appropriée. Cette décision repose sur deux principales exigences du projet : le besoin d'une liste électorale triée par ordre alphabétique et l'importance d'éviter les doublons pour maintenir l'intégrité de la liste. Le `std::set`, en tant que structure d'ensemble, réalise le tri automatiquement, facilité par l'ajout d'une fonction auxiliaire dédiée. Cette capacité à trier et à gérer de manière efficace les insertions rend le `std::set` idéal pour notre contexte, où la redondance d'éléments est inacceptable. En effet, dès qu'un élément existe déjà dans l'ensemble, toute tentative d'insertion d'un élément identique est automatiquement rejetée, assurant ainsi l'unicité des électeurs dans la liste.

Quant à la complexité des opérations de test d'appartenance, grâce à la structure interne du `std::set`, qui est basée sur un arbre binaire de recherche équilibré, cette opération est réalisée avec une complexité de **O(log n)**. Cette efficience permet de maintenir des performances élevées même à mesure que la taille de la liste s'accroît, répondant ainsi aux besoins de gestion efficace et ordonnée des électeurs inscrits dans la liste électorale.

Ressources requises pour l'élaboration des Structures de Données Concrètes (SDC)

Dans le cadre de l'adoption des principes d'encapsulation de la programmation orientée objet, chaque classe a été équipée des fonctions essentielles à ses responsabilités. Ainsi, diverses méthodes de la classe **Electeur** sont sollicitées par les classes Isoloir et Espace Isoloirs. Un exemple marquant est la procédure **decision**, qui permet à un électeur de finaliser son choix de candidat en sélectionnant parmi les bulletins pris sur la table de décharge. Pour une meilleure compréhension des Structures de Données Concrètes (SDC) décrites ultérieurement, nous détaillerons dans cette section toutes les fonctions mises en œuvre dans la classe **Electeur**.

type Bulletin = **Enregistrement**
    chaîne de caractères nomCandidat;
    entier sensibilitePolitique;
fin enregistrement

type tabCapa = **Enregistrement**
    entier capacite;
    entier occupation;
    pointeur vers tableau de Bulletin tab;
fin enregistrement

type Electeur = **Enregistrement**
    entier duree;
    tabCapa bulletins;
    chaîne de caractères choix;
    booleen estEntree;
    booleen estEntreeDecharge;
    booleen estDansEspaceIsoloir;
    booleen estEntreeIsoloir;
    booleen estEntreeTableDeVote;
    booleen aPrisSesBulletins;
    booleen aFaitSonChoix;
fin enregistrement

Fonctions de la classe Electeur

Algorithm 1 assignerDuree(inout Electeur e, in entier temps)

1: **début**
2: e.duree ← temps;
3: Fin

Algorithm 2 validerEntreeIsoloir(inout Electeur e)

1: **début**
2: e.estEntreeIsoloir ← true;
3: Fin

Algorithm 3 obtenirDuree(inout Electeur e) → entier

1: **début**
2: retourner e.duree;
3: Fin

Algorithm 4 modificationDuree(inout Electeur e)

1: **début**
2: e.duree ← e.duree - 1;
3: Fin

Algorithm 5 validerChoixDeCandidat(inout Electeur e)

1: **début**
2: e.aFaitSonChoix ← true;
3: Fin

Algorithm 6 decision(inout Electeur e, in reel pn, in entier dmax)

```
1:  variables
2:  reel probability;
3:  entier i;
4:  chaîne de caractères meilleurChoix;
5:  entier meilleureDistance;
6:  chaîne de caractères blanc;
7:  entier distance;
8:  début
9:  probability ← generer_nombre_alatoire();          ▷ génère un nombre aléatoire entre 0 et 1
10: meilleurChoix ← "";
11: blanc ← "";
12: meilleureDistance ← dmax + 1;
13: si probability < pn alors
14:     e.choix ← "nul";
15: sinon
16:     pour i allant de 1 à e.bulletins.occupation faire
17:         si obtenirNomCandidat(mem(e.bulletins.tab)[i]) = "blanc" et est_vide(blanc) alors
18:             blanc ← obtenirNomCandidat(mem(e.bulletins.tab)[i]);
19:         finsi
20:         distance = valeur_absolue(spol(e) - obtenirSensibilitePolitique(mem(e.bulletins.tab)[i]));
21:         si distance <= dmax et (est_vide(meilleurChoix) ou distance < meilleurDistance)
22:             meileurChoix ← obtenirNomCandidat(mem(e.bulletins.tab)[i]);
23:             meilleureDistance ← distance;
24:         finsi
25:     fin pour
26:     si est_vide(meilleurChoix) alors
27:         si !est_vide(blanc) alors
28:             meilleurChoix ← blanc;
29:         finsi
30:     finsi
31:     si est_vide(meilleurChoix) alors
32:         e.choix ← "nul";
33:     sinon
34:         e.choix ← meilleurChoix;
35:     fin sinon
36: ecrire(" ISOLOIR");
37: ecrire(" " + (id(e)+1) + " choisit " + e.choix)
38: Fin
```

SDA et SDC d'un isoloir

SDA d'un isoloir

Isoloir est le type d'un isoloir dans un bureau de vote muni des opérations suivantes:

- **creer_isoloir**(in reel pn, in entier duree, in dp, in compteurId, out Isoloir iso)

  sortie : *iso* est un isoloir vide.

- **entrer_dans_isoloir**(inout Isoloir iso, in Electeur* e)

  pré : *iso* doit être vide.

  rôle : Permet de faire rentrer un électeur dans un isoloir vide.

- **gestion_temps**(inout Isoloir iso)

  pré : *iso* doit contenir un électeur.

  rôle : Permet de contrôler la durée d'un dans un isoloir.

- **choisirBulletin**(inout Isoloir iso)

  pré : *iso* n'est pas vide.

  rôle : Permet à un électeur de faire son choix de bulletin parmi le lots de bulletins pris sur la table de décharge.

- **sortir_isoloir**(inout Isoloir iso)

  pré : *iso* n'est pas vide.

  Séquence d'exécution : L'électeur présent dans l'isoloir sort.

- **est_occupe**(inout Isoloir iso)→ **booléen**

  sortie : retourne vrai si l'isoloir est occupé, faux sinon.

SDC d'un isoloir

type Isoloir = **Enregistrement**
  booléen isoloirOccupe;
  entier idIsoloir;
  entier distance;
  entier dureeIsoloir
  réel probabiliteNulle ;
  pointeur vers Electeur electeurPresent;
fin enregistrement

Algorithm 7 creer_isoloir(in reel pn, in entier duree, in dp, in compteurId, out Isoloir iso)

1: **début**
2: iso.isoloirOccupe ← false;
3: iso.idIsoloir ← compteurId;
4: iso.distance ← dp; ▷ **Coût** : Constant $O(1)$
5: iso.dureeIsoloir ← duree;
6: iso.probabiliteNulle ← pn;
7: iso.electeuurPresent ← `nullptr`;
8: Fin

Algorithm 8 entrer_dans_isoloir(inout Isoloir iso, in Electeur* e)

1: **début**
2: iso.electeurPresent ← `adresse(e);`
3: iso.isoloirOccupe ← true;
4: assignerDuree(mem(e), iso.dureeIsoloir); ▷ **Coût** : Constant $O(1)$
5: validerEntreeIsoloir(e);
6: ecrire(" ISOLOIR");
7: ecrire(" "+id(mem(e))+1+" "+"entre");
8: Fin

Algorithm 9 choisirBulletin(inout Isoloir iso)

1: **début**
2: decision(mem(iso.electeurPrsent), iso.probabiliteNulle, iso.distance); ▷ **Coût** : Linéaire $O(N)$
3: Fin

La procédure *decision* examine séquentiellement la liste des bulletins que l'électeur a collectés à la table de décharge. Elle sélectionne le bulletin correspondant au candidat de préférence, basé sur l'alignement de la sensibilité politique de l'électeur avec celle indiquée sur le bulletin. Le nombre total de bulletins examinés au cours de cette procédure est désigné par N.

Algorithm 10 sortir_isoloir(inout Isoloir iso)

1: **début**
2: iso.isoloirOccupe ← false;
3: ecrire(" "+id(mem(iso.electeurPresent))+1+" "+"sort"); ▷ **Coût** : Constant $O(1)$
4: validerChoixDeCandidat(mem(iso.electeurPresent));
5: iso.electeurPresent ← nullptr;
6: Fin

Algorithm 11 gestion_temps(inout Isoloir iso)

1: **début**
2: si iso.electeurPresent $\neq$ `nullptr` alors
3: si obtenirDuree(mem(iso.electeurPresent)) > 0 alors
4: modificationDuree(mem(iso.electeurPresent));
5: fin si ▷ **Coût** : $\Omega(1) \cap O(N)$
6: si obtenirDuree(mem(iso.electeurPresent)) $= 0$ alors
7: choisirBulletin(iso.electeurPresent);
8: sortir_isoloir(iso);
9: fin si
10: fin si
11: Fin

La méthode '*gestion_temps*' exécute des opérations de coût constant (représenté par $\Omega(1)$) quand la durée restante de l'électeur n'est pas épuisée, ce qui constitue le meilleur cas. Néanmoins, dès que le compteur de temps de l'électeur atteint zéro, à chaque itération, cette méthode invoque la procédure '*decision*', dont le coût est linéaire, $O(N)$. Cette situation représente donc le pire des cas pour la performance de '*gestion_temps*'.

Algorithm 12 est_occupe(inout Isoloir iso)→ booléen

1: **début**
2: Retourner iso.isoloirOccupe; ▷ **Coût** : Constant $O(1)$
3: Fin

SDA et SDC d'un espace d'isoloirs

SDA d'un espace d'isoloirs

EspaceIsoloirs est le type représentant un espace d'isoloirs dans un bureau de vote muni des opérations suivantes:

- **creer_espace**(in reel pn, in entier nombresIsoloirs, in entier duree, in entier dp, out EspaceIsoloirs esp);

  sortie : *esp* est un espace d'isoloir vide.

- **rechercheIsoloirLibre**(inout EspaceIsoloirs esp) → **entier**

  rôle : renvoie l'indice du premier isoloir libre.

- **etatIsoloir**(inout EspaceIsoloirs esp) → booleen

  rôle : retourne vrai si il y a un isoloir occupé et faux sinon.

- **traiterFileAttente**(inout EspaceIsoloirs esp)

  rôle : Fais passer le premier électeur de la file d'attente dans le premier isoloir libre.

- **obtenirTailleFile**(inout EspaceIsoloirs esp) → **entier**

  rôle : renvoie la taille de l'espace d'isoloirs c'est-à-dire le nombre d'isoloirs dans l'espace.

- **gestionDuree**(inout EspaceIsoloirs esp)

  rôle : contrôle le temps d'un électeur dans un isoloir occupé de l'espace.

- **entrerIsoloirLibre**(in Electeur* e)

  rôle : Fais rentrer l'électeur dans un isoloir libre si la file d'attente est vide sinon le fait rentrer dans la file.

SDC d'un espace d'isoloirs

type Maillon = **Enregistrement**
  Electeur* e;
  pointeur vers Maillon suiv;
fin enregistrement

type fileElecteur = **Enregistrement**
  entier nombresElecteurs;
  pointeur vers Maillon premier;
  pointeur vers Maillon dernier;
fin enregistrement

type tabCapaEspace = **Enregistrement**
  entier capacite;
  entier occupation;
  pointeur vers tableau de Isoloir tableau;
fin enregistrement

type EspaceIsoloirs = **Enregistrement**
  tabCapaEspace espaceIsoloirs;
  fileElecteur fileAttenteIsoloir;
  file d'entier isoloirsVides;
  entier isoloirsOccupe;
fin enregistrement

Algorithm 13 creer_espace(in reel pn, in entier nombresIsoloirs, in entier duree, in entier dp, out EspaceIsoloirs esp)

```
1:  variables
2:  entier i,
3:  Isoloir iso;
4:  début
5:  esp.isoloirsOccupe ← 0;
6:  esp.espaceIsoloirs.capacite ← nombresIsoloirs;
7:  esp.espaceIsoloirs.occupation ← 0;
8:  esp.espaceIsoloirs.tableau ← allocation de tableau de esp.espaceIsoloirs.capacite d'Isoloir;
9:  creer_file(esp.fileAttenteIsoloir);
10: creer_file(esp.isoloirVides);                    ▷ Coût : Linéaire O(N)
11: pour i allant de 1 à nombresIsoloirs faire
12:     creer_isoloir(pn, duree, dp, 200, iso);
13:     mem(esp.espaceIsoloirs.tableau)[i] ← iso;
14:     esp.espaceIsoloirs.occupation ← ++;
15:     enfiler(esp.isoloirVides, i);
16: fin pour;
17: Fin
```

La procédure '*creer_espace*' est conçue pour initialiser un nouvel espace d'isoloirs. Elle utilise une boucle qui itère de 1 jusqu'au nombre total d'isoloirs prévu, ce qui induit une complexité linéaire, $O(N)$, où N représente le nombre d'isoloirs. Grâce à l'utilisation d'un tableau dynamique, aucun agrandissement ultérieur n'est nécessaire, car l'espace est configuré dès le début avec le nombre exact d'isoloirs requis. Cette approche prévient les coûts additionnels liés à un redimensionnement potentiel durant l'utilisation de l'espace isoloir.

Algorithm 14 rechercheIsoloirLibre(inout EspaceIsoloirs esp) → **entier**

```
1: variables
2: entier indice;
3: début
4: si !est_vide(esp.isoloirsVides) alors
5:     indice ← premier(esp.isoloirsVides);                ▷ Coût : Constant O(1)
6:     defiler(esp.isoloirsVides);
7:     retourner indice;
8: finsi ;
9: retourner -1;
10: Fin
```

Algorithm 15 etatIsoloir(inout EspaceIsoloirs esp) → booleen

```
1: début
2: retourner esp.isoloirsOccupe > 0;                       ▷ Coût : Constant O(1)
3: Fin
```

Algorithm 16 traiterFileAttente(inout EspaceIsoloirs esp)

```
1: variables
2: entier indice;
3: Electeur* prochain;
4: début
5: tant que !est_vide(esp.fileAttenteIsoloir) && esp.isoloirsOccupe < esp.espaceIsoloirs.occupation
   faire
6:     indice ← rechercheIsoloirLibre(esp);
7:     si indice ≠ -1 alors                                ▷ Coût : Linéaire O(N)
8:         prochain ← premier(esp.fileAttenteIsoloir);
9:         defiler(esp.fileAttenteIsoloir);
10:        entrer_dans_isoloir(mem(esp.espaceIsoloirs.tableau)[indice], mem(prochain));
11:        esp.isoloirsOccupe ++;
12:        finsi;
13:    fin tant que;
14: Fin
```

la procédure '*traiterFileAttente*' parcourt la file d'attente et essaie de placer les électeurs dans les isoloirs libres jusqu'à ce que la file soit vide ou qu'il n'y ait plus d'isoloirs libres. Cela pourrait potentiellement impliquer de parcourir toute la file d'attente et tous les isoloirs, en fonction de leur état d'occupation.

Algorithm 17 obtenirTailleFile(inout EspaceIsoloirs esp) → **entier**

```
1: début
2: retourner esp.fileAttenteIsoloir.nombresElecteurs;          ▷ Coût : Constant O(1)
3: Fin
```

Algorithm 18 gestionDuree(inout EspaceIsoloirs esp)

```
1: variables
2: entier i;
3: début
4: pour i allant de 1 à esp.espaceIsoloirs.occupation faire
5:     si est_occupe(mem(esp.espaceIsoloirs.tab)[i]) alors
6:         gestion_temps(mem(esp.espaceIsoloirs.tab)[i]);          ▷ Coût : Linéaire O(N)
7:             si !est_occupe(mem(esp.espaceIsoloirs.tab)[i] alors
8:                 enfiler(esp.isoloirsVides, i);
9:                 esp.isoloirsOccupe - -;
10:            finsi
11:    finsi
12: fin pour
13: Fin
```

Cette méthode parcourt tous les isoloirs pour mettre à jour leur état basé sur le temps restant. Chaque isoloir gère son temps, et en fonction de leur état (occupé ou non), ils peuvent être libérés, ce qui requiert une mise à jour de la liste des isoloirs libres. La complexité est linéaire en fonction du nombre d'isoloirs.

Algorithm 19 entrerIsoloirLibre(in Electeur* e)

```
1: variables
2: entier indice;
3: début
4: validerPresenceEspaceIsoloir(mem(e));          ▷ Coût : Ω(1) ∩ O(N)
5: indice ← rechercheIsoloirLibre(esp);
6: si indice ≠ -1 alors
7:     entrer_dans_isoloir(mem(esp.espaceIsoloirs.tableau)[indice], mem(e));
8:     esp.isoloirsOccupe ++;
9: sinon
10:    enfiler(esp.fileAttenteIsoloir, mem(e));
11: fin sinon
12: traiterFileAttente(esp);
13: Fin
```

La procédure '*entrerIsoloirLibre*' tente d'abord de placer un électeur dans un isoloir libre via la fonction '*rechercheIsoloirLibre*', qui fonctionne à coût constant. Si aucun isoloir n'est libre, l'électeur est ajouté à la file d'attente, également à coût constant. Ensuite, '*traiterFileAttente*' est appelée pour faire avancer les électeurs dans la file vers les isoloirs libres. Si tous les isoloirs sont occupés ou la file est vide, cette opération est rapide et à coût constant. Toutefois, si des isoloirs se libèrent et que la file contient des électeurs, le coût peut devenir linéaire.

Trace d'exécution

```
ELECTION 'Euro 2024'
    Candidat n°0 : X nel 5
    Candidat n°1 : Y rik 2
    Candidat n°2 : Z pat 8

BUREAU n°310
LISTE ELECTORALE
    A bic (0)
    B gad (1)
    C ann (2)
    D pol (3)
    E lam (4)
    F bul (5)
    G yap (6)
    X nel (7)
    Y rik (8)
    Z pat (9)

PREPARATION DECHARGE
    X nel : 10 bulletins
    Y rik : 10 bulletins
    Z pat : 10 bulletins

OUVERTURE BUREAU n°310
TMAX = 20

T = 1
T = 2
T = 3
    ENTREE
        4 entre
    DECHARGE
        4 entre
T = 4
    ENTREE
        8 entre
T = 5
    DECHARGE
        4 prend Y rik
        4 prend Z pat
        4 prend X nel
    ENTREE
        3 entre
T = 6
    DECHARGE
        4 sort
    ISOLOIR
        4 entre
T = 7
    DECHARGE
        8 entre
    ENTREE
        2 entre
T = 8
T = 9
    DECHARGE
        8 prend Z pat
        8 prend Y rik
T = 10
    DECHARGE
        8 sort
    ISOLOIR
        8 entre
T = 11
    DECHARGE
        3 entre
    ENTREE
        1 entre
T = 12
    ISOLOIR
        4 choisit Y rik
        4 sort
    VOTE
        4 entre
T = 13
    DECHARGE
        3 prend Y rik
T = 14
    DECHARGE
        3 sort
    VOTE
        4 vote
    ISOLOIR
        3 entre
T = 15
    DECHARGE
        2 entre
    ENTREE
        6 entre
T = 16
    ISOLOIR
        8 choisit Z pat
        8 sort
T = 17
    DECHARGE
        2 prend X nel
        2 prend Z pat
    VOTE
        4 sort
    SORTIE
        4 sort
T = 18
    DECHARGE
        2 sort
    VOTE
        8 entre
    ISOLOIR
        2 entre
T = 19
    DECHARGE
        1 entre
    ENTREE
        5 entre

FERMETURE ENTREE

T = 20
    ISOLOIR
        3 choisit Y rik
        3 sort
    VOTE
        8 vote
T = 21
    DECHARGE
        1 prend X nel
        1 prend Y rik
T = 22
    DECHARGE
        1 sort
    ISOLOIR
        1 entre
T = 23
    DECHARGE
        6 entre
    VOTE
        8 sort
    SORTIE
        8 sort
T = 24
    ISOLOIR
        2 choisit Z pat
        2 sort
    VOTE
        3 entre
T = 25
    DECHARGE
        6 prend Z pat
        6 prend X nel
T = 26
    DECHARGE
        6 sort
    VOTE
        3 vote
    ISOLOIR
        6 entre
T = 27
```

DECHARGE
5 entre
T = 28
ISOLOIR
1 choisit X nel
1 sort
T = 29
DECHARGE
5 prend X nel
5 prend Y rik
VOTE
3 sort
SORTIE
3 sort
T = 30
DECHARGE
5 sort
VOTE
2 entre
ISOLOIR
5 entre
T = 31
T = 32
ISOLOIR
6 choisit Z pat
6 sort
VOTE
2 vote
T = 33
T = 34
T = 35
VOTE
2 sort
SORTIE
2 sort
T = 36
ISOLOIR
5 choisit Y rik
5 sort
DECHARGE
ISOLOIR
VOTE
1 entre
T = 37
T = 38
VOTE
1 vote
T = 39
T = 40
T = 41
VOTE
1 sort
SORTIE
1 sort
T = 42
VOTE
6 entre
T = 43
T = 44
VOTE
6 vote
T = 45
T = 46
T = 47
VOTE
6 sort
SORTIE
6 sort
T = 48
VOTE
5 entre
T = 49
T = 50
VOTE
5 vote
T = 51
T = 52
T = 53
VOTE
5 sort
SORTIE
5 sort

FERMETURE BUREAU n°310

Bureau n°310 : RESULTATS Euro 2024

PRESIDENT : C ann
nb electeurs : 10
nb votes : 7
participation : 70%
abstention : 30%
nul : 0 (0%)
blanc : 0 (0%)
Z pat : 3 (43%)
Y rik : 3 (43%)
X nel : 1 (14%)

Bilan du Projet

Complétude du projet

Dans ce projet, nous avons rigoureusement implémenté toutes les étapes essentielles du processus de vote, couvrant l'entrée dans le bureau de vote, la sélection des bulletins à la table de décharge, la prise de décision dans l'isoloir, et la formalisation finale du vote à la table de vote avec émargement. Chaque méthode a été optimisée pour minimiser les coûts d'exécution tout en maximisant la performance.

Sur le plan de la programmation orientée objet, nous avons strictement respecté ses principes fondamentaux, tels que l'encapsulation, qui nous a permis de protéger l'état interne des objets et d'assurer une interface claire pour leur manipulation. En outre, l'héritage a été utilisé pour hiérarchiser et réutiliser le code efficacement, tandis que le polymorphisme a facilité l'extension et la modification des fonctionnalités sans compromettre l'architecture existante.

Nous avons également intégré des tests unitaires pour chaque composant, assurant que chaque fonctionnalité répond aux exigences prévues et fonctionne correctement dans divers scénarios. Cela a été crucial pour maintenir la robustesse du système et pour simplifier les phases de débogage et de maintenance. En faisant le bilan, nous avons constaté que tous les objectifs fixés pour ce projet ont été atteints.

Correction du projet

Au cours de la réalisation de ce projet, nous avons rencontré plusieurs défis, notamment des erreurs liées à la copie d'objets. Initialement, dans notre application, nous avons eu des problèmes avec la copie des objets Electeur. Cela entraînait des situations où les modifications apportées ne se répercutaient pas sur les objets concernés, ce qui rendait les mises à jour incohérentes.

Ce problème s'est avéré difficile à identifier. Nous avons donc fait appel au débogueur C++ GDB, qui s'est révélé indispensable pour comprendre l'erreur. Suite à cette analyse, nous avons modifié notre approche en passant directement des pointeurs vers l'objet Electeur dans la liste électorale. Cette solution non seulement simplifiait le code, mais garantissait également la cohérence des données à travers l'application.

Cette expérience a souligné l'importance de choisir judicieusement entre la copie d'objets et l'utilisation de références ou de pointeurs, surtout dans un contexte où la cohérence et l'intégrité des données sont cruciales.

Améliorations Futures

Pour enrichir et structurer notre section sur les améliorations futures, nous envisageons plusieurs extensions significatives pour notre simulation de bureau de vote, afin d'accroître sa fonctionnalité et sa pertinence.

Multi-bureau de vote

Actuellement, notre simulation se concentre sur un seul bureau de vote. Nous prévoyons d'élargir ce cadre pour inclure plusieurs bureaux de vote. Cela permettrait l'enregistrement des électeurs dans des bureaux spécifiques, basé sur des critères géographiques ou démographiques déterminés, offrant ainsi une simulation plus représentative des élections nationales ou régionales.

Processus électoral complet

Nous souhaitons développer un processus électoral plus complet en intégrant des phases telles que les campagnes électorales, la préparation des listes électorales, et la publication des résultats. Cela enrichirait l'expérience utilisateur en fournissant un aperçu complet du cycle électoral.

Amélioration de la gestion des données

Pour gérer efficacement l'augmentation de la complexité due à l'ajout de multiples bureaux de vote, une refonte de notre système de gestion des données sera nécessaire. Cela comprendra l'optimisation des structures de données pour un accès plus rapide et la sécurisation des informations sensibles des électeurs.

Interface utilisateur interactive

Pour rendre notre simulation plus accessible et engageante, le développement d'une interface utilisateur graphique interactive est prévu. Cela permettrait aux utilisateurs de visualiser en temps réel les différentes étapes du processus électoral et d'interagir directement avec le système pour modifier les paramètres ou suivre des scénarios spécifiques.

Analyse statistique des résultats

En plus de simuler le processus de vote, nous aimerions fournir des outils d'analyse statistique pour évaluer les résultats des élections, permettant aux utilisateurs de générer des rapports détaillés sur les tendances de vote, la participation, et d'autres métriques pertinentes.

Ces améliorations visent non seulement à étendre les fonctionnalités de notre projet mais aussi à en faire un outil pédagogique et de recherche plus robuste pour les études en sciences politiques et en administration publique.

Evaluation Critique

Notre application répond aux exigences initiales et satisfait les attentes des utilisateurs grâce aux implémentations réalisées. Un soin particulier a été apporté à la lisibilité du code, avec des noms de variables et de fonctions explicites et un commentaire exhaustif de chaque segment de code.

En matière de conformité aux normes de la programmation orientée objet, nous avons intégré des éléments clés tels que le polymorphisme, l'héritage et l'encapsulation. Ces pratiques ont renforcé la structure et la robustesse de notre projet.

Au cours de ce projet, nous avons acquis de nombreuses connaissances, notamment dans l'utilisation des bibliothèques C++ standard telles que `std::vector`, `std::stack`, et `std::map`. Les débuts étaient difficiles, principalement à cause de la nécessité de comprendre comment accéder et utiliser ces bibliothèques. Toutefois, ce travail assidu nous a permis de bâtir une solide compréhension des fondements de la programmation orientée objet en C++.

Nous ne maîtrisons peut-être pas encore parfaitement toutes les subtilités, mais nous avons établi une base solide qui nous permettra de mener à bien des projets futurs de manière plus efficace et plus rapide.

Conclusion

Ce projet de simulation d'un bureau de vote en C++ a été une entreprise enrichissante et formatrice qui a mis en évidence la puissance et la flexibilité de la programmation orientée objet. En adoptant une approche méthodique et structurée, nous avons réussi à concevoir et à implémenter une simulation qui non seulement répond aux exigences techniques spécifiques, mais offre également une plateforme dynamique pour comprendre les processus électoraux.

Nous avons traversé divers défis, de la conception initiale à l'implémentation des fonctionnalités complexes et de la gestion des données. Chaque obstacle a été une opportunité d'apprentissage, nous poussant à approfondir notre compréhension des principes de la programmation C++ et à affiner notre capacité à résoudre des problèmes.

Les tests exhaustifs et le débogage ont joué un rôle crucial dans l'assurance de la fiabilité et de la robustesse de notre application. Ces étapes ont renforcé notre confiance dans la capacité du programme à simuler fidèlement et efficacement les différents aspects d'un bureau de vote.

En regardant vers l'avenir, les améliorations envisagées, telles que l'extension à plusieurs bureaux de vote et l'enrichissement des phases du processus électoral, ouvriront de nouvelles voies pour rendre la simulation plus complète et plus utile dans un contexte éducatif ou de recherche.

En conclusion, ce projet n'a pas seulement été une démonstration de compétences techniques, mais aussi une illustration du rôle crucial que joue la technologie dans la facilitation et la compréhension des pratiques démocratiques. Alors que nous continuons à explorer et à étendre cette application, nous restons engagés à contribuer à un avenir où la technologie soutient de manière transparente et efficace la gouvernance et la participation civique.