Mini-Projet R6.A.05 : bataille navale client/ serveur

année 2024-2025

Projet R6.A.05 : C++ - bataille navale client/serveur

1. Présentation

Ce projet propose de développer un jeu de bataille navale à travers le réseau dans le cas simple (dans un premier temps) où le client et le serveur se situent sur la même machine (machine locale) et que le serveur ne peut jouer que contre un seul client.

Le projet se déroule sur 5,5 X 1h30 (5,5 séances sur 2 semaines) et fera l'objet d'une note de contrôle continu. Il est à réaliser individuellement et sera déposé sur Moodle le **mercredi 29 janvier à 23h55 au plus tard (attention au malus -2 points si retard)**.

2. Travail demandé

Réaliser un programme en C++ permettant de jouer au combat naval entre un serveur et un client. Partant d'un diagramme de classes de conception de l'application (voir annexe1) et de sources C++ fournis (voir l'archive à disposition sur Moodle), il est demandé de coder trois classes une par une pour arriver à l'application finale.

Pour chaque classe à coder, il est également demandé d'y associer un fichier (*testCMaClasse.cpp*) contenant un lanceur *main()* dont le rôle est de tester la classe développée. Ce test doit être conforme à ce que vous avez toujours appris en technique de développement : test des constructeurs, accesseurs, modificateurs, méthodes publiques, tests dans 3 cas distincts : test des cas normaux, test des cas d'erreurs et test des cas limites. Chaque erreur doit être gérée par un message affiché à l'écran. Chaque méthode (ou constructeur) doit être testé(e) indépendamment des autres dans une **unité de test séparée** (ex. *void testConstructeur1EtAccesseurs(), void testModificateurs(), etc.*). Chaque fichier *CMaClasse.cpp* doit contenir des commentaires de type « **javaDoc** » pour chacune des méthodes codée.

Le lanceur final de l'application (*main.cpp*) est fourni. Ce lanceur final ne compilera et ne s'exécutera correctement **que si vous avez respecté scrupuleusement le cahier des charges des classes à développer**.

Attention :

- le projet doit être développé sous VSCode (comme appris en séances de TPs) avec les répertoires */src*, */bin*, */ws* (qui contiendra les fichiers *Makefile* et *flotille.txt*),
- le rendu sera exclusivement testé sous Linux avec le compilateur g++ (le C++ n'est pas portable et les bibliothèques réseaux ne sont pas identiques d'un OS à l'autre, à vous de bien tester votre jeu AVANT de rendre le projet),
- il faudra obligatoirement rendre avec le projet un fichier *Makefile*. Ce fichier permettra de compiler séparément les différents lanceurs : d'une part les lanceurs qui testent chacune des 3 classes développées et d'autre part le lanceur final (*main.cpp*) qui exécute l'application finale.

3. Description du diagramme de classes de conception (voir annexe1)

Un joueur est de type soit *CJoueurCli* s'il s'agit du joueur client, soit *CJoueurServ* s'il s'agit du joueur serveur. La classe *CJoueur* factorise des actions et des données communes à *CJoueurCli* et *CJoueurServ* MAIS elle ne s'instancie PAS. Les deux méthodes importantes pour le jeu sont : *void attaque (pair<int,int> tir, CCoups& e)* et *void attaqueAdverse (CArmada* f, CCoups* e)*, toutes les 2 décrites ci-après.

Chaque joueur (*CJoueurCli* ou *CJoueurServ*) gère sa propre interface utilisateur (*CGui*) qui affichera DEUX grilles :

- une grille « joueur » avec les bateaux du joueur + les tirs victorieux de l'adversaire + les ploufs de l'adversaire (un exemple de grille « joueur » est donné en annexe2),
- une grille « adversaire » qui fera apparaître uniquement les tirs du joueur de type « touche » ou « plouf ». Pour cela, le joueur mémorise dans sa propre structure *CCoups* tous les tirs qu'il envoie sur la grille adverse avec, pour chaque tir, l'information « touche » ou « dansLEau ».

Les classes *CSocket*, *CClientSocket* et *CSeverSocket* servent uniquement au dialogue sur le réseau.

Seuls les classes *CBateau*, *CArmada* et *CGui* sont à développer. *CBateau* et *CArmada* utilisent notamment les classes *vector* et *pair* de l'API C++ (à consulter obligatoirement !).

4. Détail de codage

Première classe à coder : la classe *CBateau*

Un bateau est caractérisé par le nombre de cases (*m_taille*) qu'il occupe sur la grille, par un nom (porte-avion, croiseur, ...) et **par le point de départ de sa position HORIZONTALE** sur la grille (appelé **point d'ancrage** du bateau sur la grille) : *pair<int,int> m_position*, (0, 0) correspondant à la case en haut à gauche sur la grille. Il mémorise également dans un tableau de booléens les dégâts occasionnés sur le bateau (une case est à faux si elle n'est pas touchée par un tir adverse).

Les attributs de *CBateau* sont donc :

int m_taille;	// nombre de cases occupées (horizontalement) sur la grille
string m_nom;	// nom du bateau
pair<int,int> m_position;	// coord. (*int ligneHoriz*, *int colonneVert*) du **point d'ancrage** // *m_position.first* = le numéro de la ligne horizontale // *m_position.second* = le numéro de la colonne verticale
bool m_pDegats;*	// tableau des dégâts de taille *m_taille* (case à *faux* si pas de // dégât)

Les méthodes de la classe *CBateau* sont les suivantes (ET il faudra RAJOUTER les 3 méthodes nécessaires à la bonne gestion de la zone dynamique) :

CBateau ();	// Constructeur par défaut : "neant", (0,0), 0, // NULL
CBateau (string n, pair<int,int> p, int t);	// Constructeur, le bateau n'a encore aucune // case touchée (*m_pDegats* à *faux* partout)
bool getDegats (int i);	// Renvoie *vrai* si la case numéro *i* du bateau est // touchée (*0 <= i < m_taille*)
string getNom();	// Accesseur
int getTaille();	// Accesseur
pair<int,int> getPosition();	// Accesseur : ancrage du bateau sur la grille
void setPosition (int i, int j);	// Modifie la position du bateau sur la grille en // *ième* ligne, *jème* colonne (nouveau point // d'ancrage)
bool estCoule();	// Renvoie *vrai* si le bateau est coulé
bool tirAdverse (pair<int,int> p);	// Tir adverse : renvoie *vrai* si la coordonnée // passée en paramètre est un tir victorieux (une // case du bateau est touchée). Si le tir est // victorieux il doit être marqué dans le tableau // *m_pDegats*. // **ATTENTION** si une case est déjà touchée il ne // faut PAS renvoyer *vrai* (donc renvoyer *faux*) // sinon un tir victorieux est comptabilisé en trop.
friend ostream& operator<< (ostream& os, CBateau& theB);	// Surcharge de l'opérateur << pour afficher à // l'écran les caractéristiques du bateau

Le fichier qui teste la classe *CBateau* s'appelera *testCBateau.cpp*.

Deuxième classe à coder : la classe *CArmada*

Une armada se compose d'une collection de bateaux (*vector<CBateau> m_listeBateaux*). La méthode *bool placerAleatoirement()* place aléatoirement TOUS les bateaux de la flottille à des positions compatibles avec les dimensions de la grille. Pour le placement aléatoire d'un bateau sur la grille, voici la marche à suivre :

- utiliser *rand()* pour tirer un entier aléatoire *k* compris entre *0* et (y compris) *borneSup* :
  1. *srand (time(NULL)) ; // initialisation du tirage aléatoire 1 seule fois*
  2. *int k = rand() % (borneSup + 1) ; // utilisation de rand() et du modulo*
- placer le bateau de telle sorte qu'il ne touche pas un autre bateau de la grille qui se trouve sur la même ligne (ils peuvent par contre se chevaucher si ils se situent

sur des lignes différentes),
- si le nombre de bateaux est trop important et rend le placement impossible sur la grille (définir un nombre de tentatives maximum avec une constante MAX_ESSAIS) alors abandonner le placement et renvoyer *faux*.

La lecture de TOUTE l'armada à partir d'un fichier texte *flotille.txt* est réalisée par la méthode *void getArmadaFromFile()* qui lit les bateaux un par un et les ajoute dans la collection de bateaux *m_listeBateaux*. Le placement aléatoire des bateaux se fera séparément dans une deuxième étape lors de la mise en place du jeu (voir fichier *main.cpp*). Le fichier *flotille.txt* doit obligatoirement respecter la structure suivante :

 # ceci est une ligne de commentaire qui commence par « # »
 # format de 1 ligne :
 # nomDuBateau<ESPACE>nombreSurGrille<ESPACE>nombreCasesHorizontales
 # exemples :
torpilleur 2 2
porte-avion 1 4

L'attribut de *CArmada* est donc :

vector<CBateau> m_listeBateaux; // la collection de bateaux qui est VIDE au départ

Les méthodes de la classe *CArmada* sont les suivantes :

PAS de constructeur : le constructeur par défaut crée le tableau *m_listeBateaux* (de taille zéro) et cette construction est AUTOMATIQUE (et sera donc automatiquement détruit).

void ajouterBateau (CBateau& unBat);	// Ajoute un bateau dans la structure // *m_listeBateaux* (à la suite avec la méthode // *push_back* de la classe *vector*)

CBateau getBateau (int i);*	// Accesseur : renvoie (par pointeur) le bateau qui
	// se trouve à l'index *i* dans le tableau (*0 <= i <*
	// *taille*)

int getEffectifTotal();	// Renvoie le nombre total de bateaux de l'armada
int getNbreTotCases();	// Accesseur : renvoie le nombre total de cases // occupées par l'armada
int getEffectif();	// Renvoie le nombre de bateaux qui ne sont pas // encore coulés (flottille encore en vie sur l'eau)
void getArmadaFromFile();	// Lecture du fichier *flotille.txt* qui contient la liste // complète de tous les bateaux

bool placerAleatoirement();	// Placement aléatoire ET automatique
	// horizontalement de TOUS les bateaux sur la
	// grille. Renvoie *faux* si le positionnement
	// automatique a échoué.

Le fichier qui teste la classe *CArmada* s'appelera *testCArmada.cpp*.

Troisième classe à coder : la classe *CGui*

Chaque joueur a besoin de visualiser sa propre grille (ses propres bateaux et les ploufs adverses) mais également la grille de l'adversaire pour connaître l'emplacement des tirs victorieux et des ploufs du joueur. Dès lors, DEUX grilles doivent être affichées : elles sont mémorisées par les attributs *m_grilleJou* et *m_grilleAdv* de la classe *CGui*.

La classe *CGui* hérite de la classe abstraite *CBaseJeu* dans laquelle est déclarée, parmi d'autres méthodes abstraites, la méthode abstraite *virtual void remplirDeuxGrilles (ostream& os) = 0* qui doit se charger de l'affichage des 2 grilles. TOUTES les méthodes abstraites (*virtual...=0*) de *CBaseJeu* doivent être redéfinies dans *CGui*.

Afin de pouvoir afficher dans les grilles, les bateaux du joueur, de même que les tirs de l'adversaire et les tirs du joueur sur la grille adverse, la classe *CGui* possède 2 attributs supplémentaires : l'un de type pointeur sur l'armada du joueur (*m_pArmada*) et l'autre de type pointeur sur la structure de mémorisation des tirs (*m_pCoups*).

Un exemple de grille du joueur est donné en annexe2. Par pure convention pour l'affichage, définir une constante TAILLE_GRILLE = *nbre de lignes + 1* (ou *nbre de colonnes + 1* car la grille est carrée) pour tenir compte de l'affichage supplémentaire des légendes *0...(nbre colonnes – 1)* et *0...(nbre lignes – 1)*.

Les attributs de CGui sont donc (exemple : pour une grille 10X10, TAILLE_GRILLE=11)

char m_grilleJou[TAILLE_GRILLE-1][TAILLE_GRILLE-1];	// Grille du joueur
char m_grilleAdv[TAILLE_GRILLE-1][TAILLE_GRILLE-1];	// Grille de l'adversaire
CArmada m_pArmada;*	// Pointeur sur l'unique armada // du joueur
CCoups m_pCoups;*	// Pointeur sur l'unique structure // d'enregistrement des tirs

Les méthodes de la classe *CGui* sont les suivantes :

CGui();	// Constructeur par défaut : *m_pArmada* et // *m_pCoups* à NULL

CGui(CArmada pArmada,
CCoups* pCoups);*
	// Constructeur qui reçoit un pointeur sur l'armada
	// et un pointeur sur la structure d'enregistrement
	// des tirs

virtual ~CGui();	// Destructeur : destruction des zones pointées par // *m_pArmada* et *m_pCoups*
void setArmadaCoups (CArmada pArmada, CCoups* pCoups);*	// Modificateur : mise à jour des attributs

bool positionnerBateaux();	// Méthode appelée au début du jeu pour
	// positionner tous les bateaux sur la grille en
	// exécutant *placerAleatoirement* de *CArmada*.
	// Renvoie *faux* si le positionnement a échoué.

pair<int,int> choisirAttaque();	// Saisie de la coordonnée (ligne, colonne) de
	// l'attaque (vérification obligatoire bon/mauvais
	// après saisie de
	// la coordonnée par l'utilisateur)

void afficheGagne();	// Affiche la partie est gagnée à l'écran
void affichePerdu();	// Affiche la partie est perdue à l'écran

friend ostream& operator<< (ostream& os, CGui& theG);	// Surcharge de l'opérateur << pour l'affichage des
	// grilles. Cette fonction fait un simple appel à
	// *remplirDeuxGrilles(...)* de l'objet *theG*.

void remplirDeuxGrilles (ostream& os);	// voir explication (1) ci-dessous

void afficherLaGrille (ostream& os, string jouOuAdv);	// Affichage d'une grille (joueur ou adversaire) à
	// l'écran, c'est-à-dire affichage du contenu de
	// *m_grilleJou* ou *m_grilleAdv* + les légendes
	// (*0...TAILLE_GRILLE-2*).

(1) la méthode *void remplirDeuxGrilles (ostream& os)* doit :

- remplir la grille du joueur *m_grilleJou* :
  - en mettant *'B'* dans les cases occupées par les cases des bateaux (du joueur) non touchées
  - en mettant *'X'* dans les cases occupées par les cases des bateaux (du joueur) touchées
  - en mettant *'O'* dans les cases ou les tirs adverses ont échoués (*ploufs*)

  Pour cela, se baser d'une part sur l'armada du joueur pour les bateaux (un bateau connait l'emplacement des tirs adverses) et, d'autre part, sur la structure *m_pCoups* du joueur pour les *ploufs* de l'adversaire (clé de recherche *ploufAdverse*).
  Afficher cette grille en appelant *afficherLaGrille (os, "joueur")*.

- remplir la grille de l'adversaire *m_grilleAdv* :
  - en mettant *'X'* dans les cases où les tirs du joueur sont victorieux
  - en mettant *'O'* dans les cases où les tirs du joueur ont échoués (*ploufs*)

  Pour cela, consulter la structure *m_pCoups* du joueur pour rechercher la position de ses tirs à l'aide des clés de recherche *touche* et *dansLEau*.
  Afficher cette grille en appelant *afficherLaGrille (os, "adversaire")*.

Le fichier qui teste la classe *CGui* s'appelera *testCGui.cpp*.

Le lanceur final : main.cpp

Le fichier *main.cpp* est fourni.

Pour compiler l'application finale, il faudra procéder, dans l'ordre, à la compilation des classes suivantes : *CBateau*, *CArmada*, *CCoups*, *CGui*, *CSocket*, *CClientSocket*, *CServerSocket*, *CJoueur*, *CJoueurCli*, *CJoueurServ* (la compilation de *CBaseJeu* n'est pas nécessaire car c'est une classe abstraite).

Le déroulement de l'application finale (*main.cpp*) est décrit dans le fichier *deroulement.txt* de l'archive qui vous est fournie.

5. Séquencement et notation

Pour chaque classe codée, un test doit y être associé. La compilation de chaque test doit pouvoir se faire grâce à un fichier *Makefile* dont un exemple est fourni. Dans la notation, le test de chaque classe sera évalué.

L'ordre dans lequel doit être développé l'application est le suivant (5,5 X 1h30 = 8h15) :

- classe *CBateau* + *testCBateau* : 2h
- classe *CArmada* + *testCArmada* : 2h30
- classe *CGui* + *testCGui* : 2 X 1h30 = 3h
- test final avec le lanceur *main.cpp* fourni (exécution sans bugs) : 45 min.

La notation tiendra compte :

- de l'état d'avancement du projet
- de la conformité du code par rapport au cahier des charges
- de la qualité et la complétude des tests
- de la qualité des commentaires, de la clarté de codage et du respect des règles de style C++

6. Rendu

Une archive *(nom_prenom.zip, pas de .rar !)* est à déposer sur Moodle (attention, retard = malus !).

L'archive doit contenir :

- un répertoire */src* qui contient tous les fichiers **.cpp* et **.h* de l'application, y compris les fichiers fournis (*main.cpp*, *CSocket.cpp* etc.) et les fichiers de test (*testCBateau.cpp* etc.),
- un répertoire */ws* qui contient les fichiers *Makefile* et *flotille.txt*,
- un fichier *readme.txt* qui doit :
  - décrire l'état d'avancement du projet
  - reprendre clairement la liste des classes qui peuvent être compilées et testées grâce au *Makefile* fourni

Important :

- NE PAS rendre une archive du projet VsCode
- NE PAS créer de sous-répertoires dans */src*
- NE PAS créer de nouvelles classes

Informations utiles

De l'aide sur C++ est disponible sur les sites suivants :

http://www.cplusplus.com/ref

http://www.cppreference.com/

Pour la classe **string**

http://www.cppreference.com/cppstring

Flux I/O et leur formatage

http://www.cplusplus.com/reference/iostream

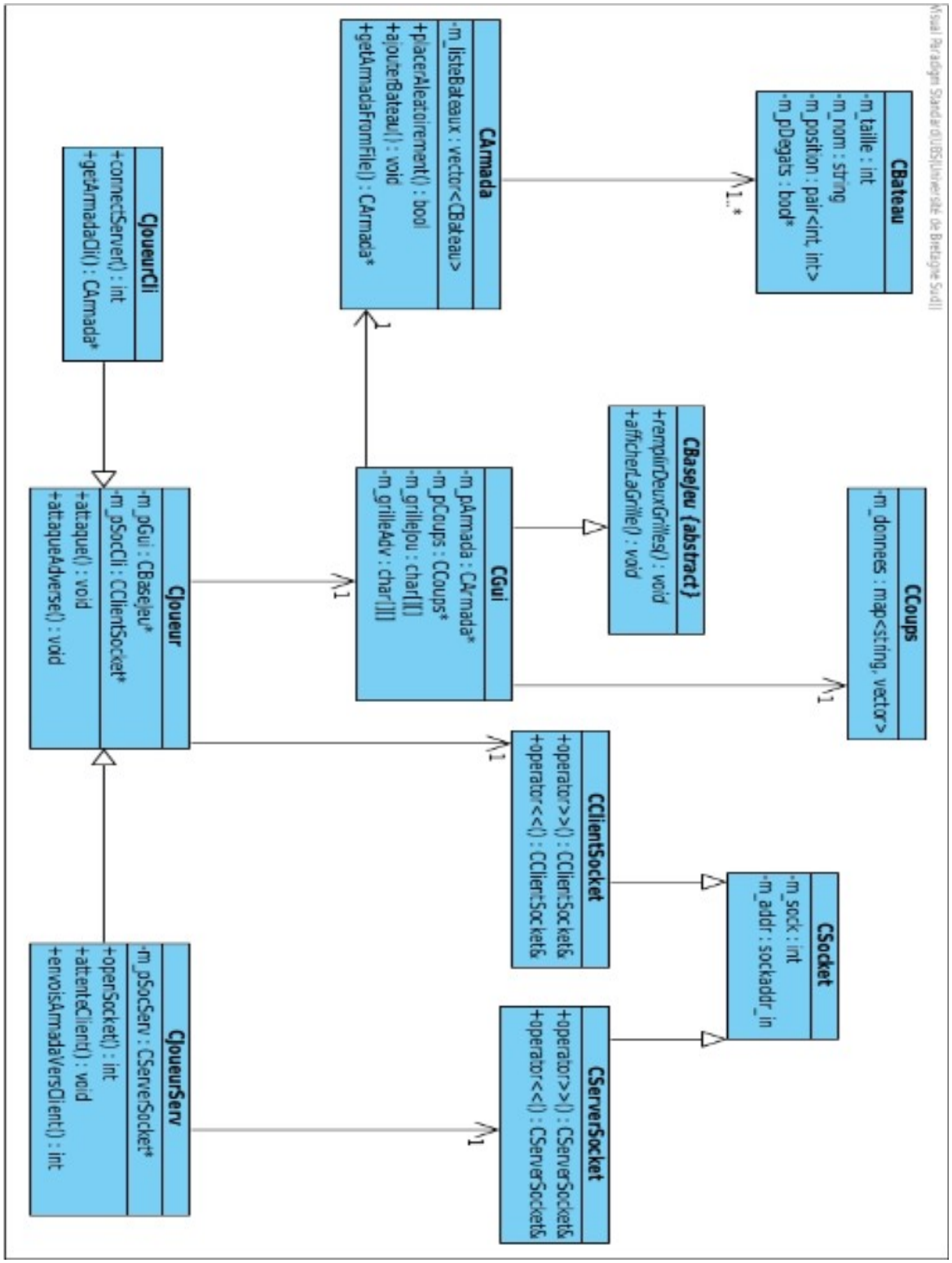
Pour la classe **vector**

http://www.cplusplus.com/reference/stl/vector

Pour la classe **pair**

http://www.cplusplus.com/reference/utility/pair/

Annexe1 : le diagramme de classes de conception



Annexe2 : un exemple de grille du joueur
(*TAILLE_GRILLE* = *11* en tenant compte des légendes *0...9*)

ligneHoriz \ colonneVert	0	1	2	3	4	5	6	7	8	9
0	-	-	-	-	-	-	-	-	-	-
1	-	B	X	B	-	-	-	-	-	-
2	-	-	-	-	-	-	B	X	-	-
3	-	-	O	-	-	-	-	-	-	-
4	-	-	-	-	-	-	-	-	-	-
5	-	-	-	-	-	O	-	-	O	-
6	-	-	-	-	-	-	-	-	-	-
7	-	-	-	B	B	B	-	B	-	-
8	-	-	X	X	-	-	-	-	-	-
9	-	-	-	-	-	-	-	-	-	-

légende :

- B : emplacement d'un bateau du joueur
- O : tir dans l'eau **de l'adversaire**
- X : bateau touché ou coulé du joueur