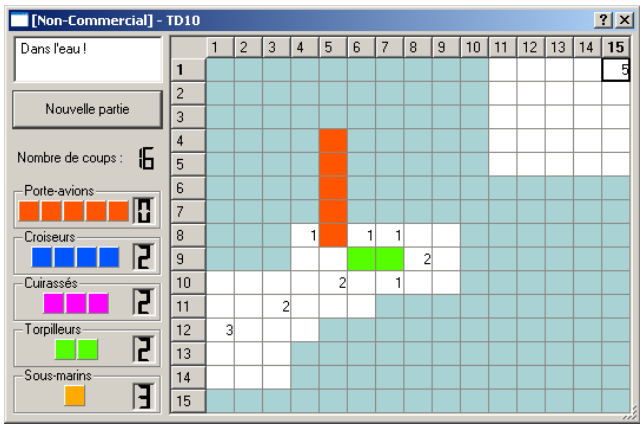
*C*entre *I*nformatique pour les *L*ettres
et les *S*ciences *H*umaines

# TD 10 : Bataille navale

Ce TD propose une version informatisée du jeu de bataille navale. Outre qu'il s'agit ici de jouer contre l'ordinateur, cette version se distingue de celle bien connue de tous les écoliers par le fait que, lorsqu'on tire dans l'eau, on obtient en réponse la distance du bateau le plus proche.

Cette information augmente la part de réflexion intervenant dans le jeu et conduit à tenir des raisonnements proches de ceux suivis dans un autre jeu bien connu, le démineur. La similarité entre ces jeux est renforcée par l'adoption d'une interface analogue : on tire en cliquant sur une case et, lorsqu'on tire dans l'eau, la surface dont on apprend qu'elle ne comporte aucun bateau change de couleur.




*Le programme réalisé au cours du TD 10*

# 1 - Analyse préalable

La mise au point de ce jeu va nous permettre d'illustrer l'intérêt qu'il y a à observer scrupuleusement l'un des grands principes généraux de la programmation :

La gestion de l'interface utilisateur et le traitement du problème doivent être pris en charge par des parties du programme aussi indépendantes l'une de l'autre que possible.

Nous allons donc commencer par écrire les fonctions de la classe de dialogue, en veillant à ne rien y traiter qui ne soit directement lié à l'interface utilisateur. Ceci nous conduira à supposer l'existence d'autres classes, offrant des fonctions qui traitent réellement le problème (positionnement des bateaux, calcul des conséquences d'un tir, etc). Une fois cette classe mise au point, la liste des suppositions faites nous guidera pour la rédaction du reste du programme.

# 2 - Création du projet et dessin de l'interface

En vous inspirant de la procédure décrite dans l'aide mémoire, créez un projet nommé TD10 ☐.

Placez, dans la partie gauche du dialogue, les widgets suivants :

- Un TextEdit (menu Tools, catégorie Input) nommé `message` ☐.
- Un PushButton [Nouvelle partie] nommé `b_nouvellePartie` ☐.
- Un LCDNumber nommé `nb_coups` ☐.

  Le jeu n'a aucun intérêt si le joueur n'essaie pas de minimiser le nombre de coups utilisés pour éliminer complètement la flotte adverse. Il est donc nécessaire de rendre ce nombre visible.

- Cinq LCDNumber (menu Tools, catégorie Display) respectivement nommés `nb_sm`, `nb_torp`, `nb_cuir`, `nb_croi` et `nb_pa` ☐. Ces widgets vont permettre au joueur de connaître le nombre de bateaux restant en jeu.

  A côté de chacun de ces QLCDNumber, placez une série de frames (menu Tools, catégorie Containers) indiquant le nombre de cases occupées par le type de bateau correspondant ☐. Changez la propriété *paletteBackgroundColor* des frames, pour donner à chaque type de bateau une couleur différente ☐.

  Dans chacune de ces séries, donnez à l'une des frames un nom évoquant le type de bateau (`sm`, `torp`, `cuir`, `croi` et `pa`) ☐. Le programme pourra ainsi retrouver facilement la couleur devant être utilisée pour dessiner chaque type de bateau.

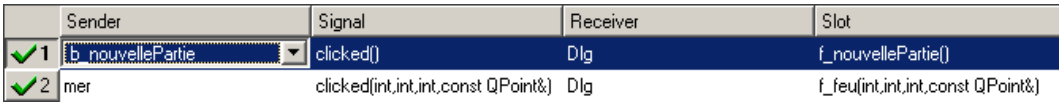
- Cinq GroupBox (menu Tools, catégorie Containers) avec les *captions* convenables, pour rendre l'interface plus intelligible pour l'utilisateur ☐.

Dans la partie droite du dialogue, placez une Table (menu Tools, catégorie Views) nommée `mer` ☐. Modifiez les propriétés *`numRows`* et *`numCols`* pour obtenir un espace de jeu convenable ☐.

Le jeu n'a d'intérêt que si la mer comporte plusieurs centaines de cases (15 x 15, par exemple). La largeur des colonnes sera réglée par le programme, ne vous en occupez pas pour l'instant.

Créez un slot `f_nouvellePartie()` et associez-le au clic sur le bouton [Nouvelle partie].

Créez un slot `f_feu(int, int, int, const QPoint &)` et associez le au clic sur la `mer`.




## 3 - Les types `ENature` et `EDegats`

La bataille navale est une question de dégâts de gravité variable infligés à des bateaux de différents types. Nous avons donc tout intérêt à disposer de types permettant de manipuler explicitement ces informations. Créez un fichier typesenum.h et donnez lui le contenu suivant :

```
1  #pragma once
2  enum ENature {AUCUN, SOUS_MARIN, TORPILLEUR, CUIRASSE, CROISEUR, PORTE_AVIONS};
3  enum EDegats {RATE, TOUCHE, COULE};
```

## 4 - La classe dialogue

La gestion de l'interface implique diverses mises à jour de l'affichage, dont la plupart sont triviales (nombres de coups et de bateaux, messages...). La fonction associée au clic sur la mer est moins simple : une fois le coup transmis à l'objet qui gère la bataille, il faut décolorer les cellules entourant le point d'impact, dans un rayon qui nous sera communiqué en réponse à notre coup.

> Calculer les conséquences du coup proposé ne relève évidemment pas de la classe de dialogue : ces conséquences sont totalement indépendantes du type d'interface utilisateur mise en place. Le changement de couleur de la zone dont on sait qu'elle est vide relève, en revanche, d'un pur choix de type d'interface : le jeu est tout à fait jouable (mais plus difficile...) si l'interface se contente d'afficher au point d'impact un nombre indiquant la distance du bateau le plus proche.

### Constructeur

Lorsque le programme est lancé, plusieurs opérations doivent être effectuées : il faut
- fixer une fois pour toute la couleur utilisée pour représenter les cellules dont on ne sait rien ;
- amorcer le générateur de nombres pseudo-aléatoires *(3)* (cf. Annexe 2) ;
- ajuster la largeur des colonnes de la `QTable` *(4-6)* ;
- spécifier les effectifs initiaux des différentes catégories de bateaux *(7-11)* ;
- préparer une nouvelle partie. Comme cette dernière opération devra également être faite entre deux parties successives, le code correspondant prend logiquement place dans la fonction liée au bouton [Nouvelle partie], fonction qui est donc appelée *(12)* par le constructeur.

Ajoutez deux membres à votre classe de dialogue : une variable de type `QMap<ENature, int>` nommée `m_effectifs` et une constante de type `QColor` nommée `m_couleurMer`.

Donnez au corps du constructeur le contenu suivant :

```
1  TD10DialogImpl::TD10DialogImpl(QWidget* parent, const char* name, bool modal, WFlags f)
                  : TD10Dialog(parent, name, modal, f), m_couleurMer(QColor(169,210,212))
2  {
3  srand(time(NULL)); //amorçage du générateur de nombres pseudo-aléatoires
4  int col;
5  for (col=0 ; col < mer->numCols() ; ++col)
6    mer->setColumnWidth(col, 24); //les colonnes ont une largeur de 24 pixels
7  m_effectifs[PORTE_AVIONS] = 1;
8  m_effectifs[CROISEUR] = 2;
9  m_effectifs[CUIRASSE] = 2;
10 m_effectifs[TORPILLEUR] = 3;
11 m_effectifs[SOUS_MARIN] = 3;
12 f_nouvellePartie();
13 }
```

> Remarquez l'usage de la liste d'initialisation, qui permet d'initialiser la constante `m_couleurMer`. La valeur utilisée pour cette initialisation est elle-même générée en appelant explicitement un constructeur de la classe `QColor`.

## La fonction f_nouvellePartie()

Cette fonction doit d'une part mettre la QTable dans un état représentant le début de partie et, d'autre part, demander la réinitialisation du jeu. La première partie de la fonction est très simple :

```
1  void TD10DialogImpl::f_nouvellePartie()
2  {
3  int col;
4  int ligne;
5  for(ligne = 0 ; ligne < mer->numRows() ; ++ligne)
6    for (col = 0 ; col < mer->numCols() ; ++col)
7      {
8      mer->clearCell(ligne,col);
9      QPixmap maPixmap (mer->columnWidth(col), mer->rowHeight(ligne));
10     maPixmap.fill(m_couleurMer);
11     mer->setPixmap(ligne, col, maPixmap);
12     }
```

Ce fragment de code utilise cinq fonctions membre de la classe QTable. Si le rôle et l'utilisation des quatre premières semblent évidents, la dernière implique l'utilisation d'une instance de la classe QPixmap. Cette classe est l'une de celles proposées par Qt pour manipuler les images. Dans le cas présent, il s'agit simplement d'une image ayant exactement la taille de la cellule concernée *(9)* et uniformément revêtue de la couleur choisie.

La seconde partie de la fonction utilise une autre variable membre, m_leJeu, qui est une instance de la classe CBataille que nous allons créer pour gérer la partie. Cette classe devra disposer d'une fonction installe() ayant pour effet de choisir les positions, dans l'instance au titre de laquelle elle est exécutée, des bateaux d'une flotte dont les effectifs lui sont communiqués par un premier argument, en respectant les limites de l'espace de jeu indiquées par le second argument.

Ce second argument est une valeur de type QPoint, obtenue par appel explicite d'un constructeur de cette classe (la classe QPoint est une classe très simple, spécialement prévue pour permettre le stockage de couples du type abscisse/ordonnée).

```
13 message->setText("Pour tirer, cliquez sur une case");
14 m_partieEnCours = m_leJeu.installe(m_effectifs,
                                      QPoint(mer->numRows(), mer->numCols()));
15 if(!m_partieEnCours)
16   message->setText("Il n'y a pas assez place pour les bateaux !");
17 inventaire();
18 }
```

1

Il n'est pas certain que la fonction installe() parvienne à effectuer sa tâche : si les dimensions qui lui sont passées sont trop faibles, elle risque de ne pas parvenir à positionner tous les bateaux requis. Elle signalera alors cet état de fait en renvoyant false, et il appartient à la fonction appelante de prévoir les mesures nécessaires dans ce cas *(15-16)*.

Ajoutez au dialogue une fonction f_tirage() définie comme suggéré ci-dessus ☐, une variable booléenne nommée m_partieEnCours ☐ et une variable de type CBataille nommée m_leJeu ☐.

## La fonction inventaire()

Le seul intérêt du code de cette fonction est qu'il révèle de nouvelles exigences quant à l'interface de la classe CBataille : elle devra comporter deux fonctions membre capables de renvoyer l'une le nombre de coups tirés et l'autre le nombre d'exemplaires restants d'un type de bateaux.

```
1  void TD10DialogImpl::inventaire()
2  {
3  nb_coups->display(m_leJeu.nombreDeCoups());
4  nb_sm->display(m_leJeu.nombreDe(SOUS_MARIN));
5  nb_torp->display(m_leJeu.nombreDe(TORPILLEUR));
6  nb_cuir->display(m_leJeu.nombreDe(CUIRASSE));
7  nb_croi->display(m_leJeu.nombreDe(CROISEUR));
8  nb_pa->display(m_leJeu.nombreDe(PORTE_AVIONS));
9  }
```

2

3

Ajoutez la fonction inventaire() à votre classe de dialogue ☐.

## La fonction f_feu()

La première partie de cette fonction est assez simple : on tire, et on modifie la case d'impact pour rendre compte de l'effet du coup :

```
1  void TD10DialogImpl::f_feu(int ligneImpact, int colImpact, int, const QPoint &)
2  {//on tire
3  if(! m_partieEnCours)
4    return;
5  CVerdict effet = m_leJeu. evalueConsequences(QPoint(colImpact, ligneImpact));
   //traitement de la case d'impact
6  mer->clearCell(ligneImpact, colImpact); //elle passe donc en blanc
7  const int rayon = effet.distance();
8  if (effet.victime() == AUCUN)
9    mer->setText(ligneImpact, colImpact, QString::number(rayon));
10 else
11   {
12   QPixmap pixImpact(mer->columnWidth(colImpact), mer->rowHeight(ligneImpact));
13   switch(effet.victime())
14     {//choix d'une couleur
15     case SOUS_MARIN: pixImpact.fill(sm->paletteBackgroundColor()); break;
16     case TORPILLEUR: pixImpact.fill(torp->paletteBackgroundColor()); break;
17     case CUIRASSE: pixImpact.fill(cuir->paletteBackgroundColor()); break;
18     case CROISEUR: pixImpact.fill(croi->paletteBackgroundColor()); break;
19     case PORTE_AVIONS: pixImpact.fill(pa->paletteBackgroundColor()); break;
20     default: pixImpact.fill(QColor(0,0,0)); break;
21     }
22   mer->setPixmap(ligneImpact, colImpact, pixImpact); //elle se teinte
23   }
```

Ce fragment de code suppose une fonction `CBataille::evalueConsequences()`renvoyant une description de l'effet d'un coup. Comme cet effet est complexe (il faut indiquer soit le type du bateau et la nature des dégâts, soit la distance du bateau le plus proche), la fonction renvoie une valeur de type `CVerdict`, une classe qu'il faudra créer. Nous exigeons que les fonctions `CVerdict::distance()` *(7)* et `CVerdict::victime()` *(8, 13)* fournissent l'information que leur nom suggère.

Dans la seconde partie de la fonction `f_feu()`, les cellules voisines du point d'impact doivent changer de couleur. Il faut traiter correctement le cas où l'impact est proche des limites du jeu, ainsi que celui où la zone à traiter comporte des traces de bateaux déjà coulés : pour préserver ces traces, seules les cellules qui ont encore la couleur de la mer sont blanchies.

```
   //blanchit la zone dont on vient d'apprendre qu'elle est vide
24 int x;
25 int y;
26 for (y = ligneImpact + 1 - rayon ; y < ligneImpact + rayon ; ++y)
27   if (y >= 0 && y < mer->numRows())
28     for (x = colImpact + 1 - rayon ; x < colImpact + rayon ; ++x)
29       if (x >= 0 && x < mer->numCols() && couleur(mer,y,x) == m_couleurMer)
30         mer->clearCell(y, x);
   //conclusion
31 message->setText(effet.message());
32 m_partieEnCours = ! effet.partieGagnee();
33 inventaire();
34 }
```

Ajoutez la fonction `f_feu()`à votre dialogue et notez que CVerdict devra proposer deux autres fonctions : `message()` doit renvoyer une `QString` décrivant la situation représentée par l'instance au titre de laquelle elle est exécutée et `partieGagnee()`doit renvoyer un booléen.

## La fonction couleur()

Déterminer la "couleur d'une cellule" de la `QTable` n'est pas immédiat, car il s'agit de la couleur de la pixmap qui est dans la cellule, et la classe `QPixmap` n'offre pas de fonction renvoyant "la" couleur de l'objet concerné (les pixmap ont habituellement plusieurs couleurs). Si la cellule contient une pixmap *(3)* , elle permet d'accéder à celle-ci, d'en demander la conversion en un

objet permettant d'accéder à son contenu, puis d'accéder à l'un des pixels (nous savons qu'ils sont tous identiques). La valeur représentant ce pixel est un `int` dont seuls les trois octets de poids faible nous intéressent (ce sont les composantes rouge, verte et bleue de la couleur du pixel). Pour obtenir un codage identique à celui d'une `QColor`, il faut forcer à 1 les bits de l'octet de poids fort.

```
1  QColor TD10DialogImpl::couleur(const QTable *t, int l, int c) const
2  {
3  if(t->item(l, c) != NULL && ! t->item(l,c)->pixmap().isNull())
4     return t->item(l,c)->pixmap().convertToImage().pixel(1,1) | 0xff000000;
5  return -1;
6  }
```

Ajoutez la fonction `couleur()`à votre classe de dialogue ☐

### Qu'avons-nous promis ?

Le tableau suivant récapitule les suppositions que nous avons étés conduits à faire pour pouvoir définir confortablement les fonctions membre de notre classe de dialogue :

| | Origine | Nature de la promesse |
|---|---|---|
| 1 | `f_tirage()` | `bool CBataille::installe(QMap<ENature, int>, QPoint)` |
| 2 | `inventaire()` | `int CBataille::nombreDeCoups()` |
| 3 | `inventaire()` | `int CBataille::nombreDe(ENature)` |
| 4 | `f_feu()` | `CVerdict CBataille::evalueConsequences(QPoint)` |
| 5 | `f_feu()` | `int CVerdict::distance()` |
| 6 | `f_feu()` | `ENature CVerdict::victime()` |
| 7 | `f_feu()` | `QString CVerdict::message()` |
| 8 | `f_feu()` | `bool CVerdict::partieGagnee()` |

## 5 - La classe `CBataille`

Ajoutez une classe `CBataille` à votre projet ☐.

### La fonction `installe()`

La fonction `installe()` commence par remettre à zéro un compteur de coups *(3)* et la collection de bateaux *(4)*. Cette collection est une simple `QValueList` de `CBateau`, une classe que nous allons créer pour représenter nos cibles. Dès que la zone de jeu n'est plus vide, il faut veiller à ce qu'un nouveau bateau ne prétende pas occuper une case déjà utilisée. La fonction `installe()` gère ce problème en mettant en place *(5)* une liste des positions qui ne sont plus libres :

9

```
1  bool CBataille::installe(const QMap<ENature, int> & effectifs, QPoint max)
2  {
3  m_nbCoups = 0;
4  m_laFlotte.clear();
5  QValueList <QPoint> occupes;
```

Le parcours de la `QMap effectifs` permet ensuite de savoir, pour chaque type de bateau, combien d'exemplaires doivent être positionnés, c'est à dire ajoutés à la flotte.

> La valeur transmise à `append()` est obtenue par appel explicite d'un constructeur de CBateau, qui choisira une position pour le nouveau bateau. Ce choix exige évidemment que l'on indique au constructeur la nature du bateau, les positions déjà occupées et les limites du jeu.

```
6  QMap<ENature, int>::ConstIterator it;
7  for(it = effectifs.begin() ; it != effectifs.end() ; ++it)
8     {
9     int n;
10    for(n=0 ; n < it.data() ; ++n)
11       m_laFlotte.append(CBateau(it.key(), occupes, max));
12    if(nombreDe(it.key()) != it.data())
13       return false; //on a manqué de place !
14    }
15 return true;
16 }
```

10

La mise en place d'un bateau peut échouer (par manque de place). Après avoir ajouté le nombre de bateaux prévus pour le type en cours de fabrication *(9-11)*, nous vérifions donc qu'ils sont tous opérationnels *(12-13)*. Cette vérification est simplifiée en exigeant que, en cas d'échec, le constructeur de `CBateau` laisse l'objet sur lequel il opérait dans un état qui le rend invisible pour `nombreDe()`.

Ajoutez à votre classe `CBataille` les variables `m_nbCoups` (de type `int`) et `m_laFlotte` ☐, ainsi que la fonction que nous venons de décrire ☐.

## La fonction `nombreDe()`

Cette fonction se contente de parcourir la flotte en incrémentant un compteur chaque fois que le bateau rencontré correspond au type sur lequel porte la question. Elle exige donc que la classe `CBateau` offre une fonction `nature()` renvoyant le type du bateau au titre duquel elle est appelée.

```
1  int CBataille::nombreDe(ENature leTypeQuiNousInteresse) const
2  {
3  int effectif = 0;
4  QValueList<CBateau>::ConstIterator it;
5  for (it = m_laFlotte.begin() ; it != m_laFlotte.end() ; ++it)
6    if((*it).nature() == leTypeQuiNousInteresse)
7      ++effectif;
8  return effectif;
9  }
```

11

Ajoutez la fonction `nombreDe()`à votre classe `CBataille` ☐

## La fonction `evalueConsequences()`

Le rôle principal de cette fonction est de transmettre les coordonnées du point d'impact à chacun des bateaux de la flotte *(8)*. Si un bateau est atteint (ie. TOUCHE ou COULE), la fonction s'achève immédiatement en renvoyant son verdict *(19)*.

La fonction `evalueConsequences()` doit aussi gérer la flotte *(15-17)* et, en cas de "coup dans l'eau", calculer la distance du bateau le plus proche. Ce calcul exige que chaque bateau place sa propre distance au point d'impact dans la réponse qu'il fournit à un tir : il suffit d'utiliser une variable *(4)* pour capturer *(12-13)* la plus petite des valeurs observées. C'est cette valeur qui doit, si aucun bateau n'est atteint, figurer dans le verdict final *(22)*.

```
1  CVerdict CBataille:: evalueConsequences(QPoint impact)
2  {
3  ++m_nbCoups;
4  int distanceMini = INT_MAX;
5  QValueList<CBateau>::Iterator it;
6  for (it = m_laFlotte.begin() ; it != m_laFlotte.end() ; ++it)
7    {
8    CVerdict resultat = (*it).encaisse(impact);
9    switch(resultat.gravite())
10     {
11     case RATE :
12       if(resultat.distance() < distanceMini)
13         distanceMini = resultat.distance();
14       break;
15     case COULE :
16       m_laFlotte.remove(it);
17       resultat.partieGagnee(m_laFlotte.isEmpty());
18     case TOUCHE :
19       return resultat;
20     }
21   }
22 return CVerdict(AUCUN, RATE, distanceMini);
23 }
```

12

13

La classe `CBateau` doit offrir une fonction `encaisse()`renvoyant un `CVerdict` qui décrit l'effet d'un tir dans la case désignée par son paramètre sur le bateau au titre duquel elle est invoquée.

Ajoutez la fonction `encaisse()`à votre classe `CBataille` ☐.

### Le fichier bataille.h

Une fois ajoutée la définition *(7)* de la fonction `nombreDeCoups()`, la définition de la classe `CBataille` prend finalement l'allure suivante :

```
#include "qmap.h"
#include "typesenum.h"
#include "bateau.h"
#include "verdict.h"
1  class CBataille
2  {
3  public:
4     bool installe(const QMap<ENature, int> & effectifs, QPoint max);
5     CVerdict evalueConsequences(QPoint impact);
6     int nombreDe(ENature lequel) const;
7     int nombreDeCoups()const {return m_nbCoups;}
8  protected:
9     int m_nbCoups;
10    QValueList <CBateau> m_laFlotte;
11 };
```

Vérifiez que l'interface de la classe `CBataille` tient toutes les promesses faites à son sujet .

### Qu'avons-nous promis ?

| | Origine | Nature de la promesse |
|---|---|---|
| 9 | `m_laFlotte` | `CBateau::CBateau()` |
| 10 | `installe()` | `CBateau::CBateau(ENature, QValueList<QPoint> &, QPoint)` |
| 11 | `nombreDe()` | `ENature CBateau::nature()` |
| 12 | `encaisse()` | `CVerdict CBateau::encaisse(QPoint)` |
| 13 | `encaisse()` | `void CVerdict::partieGagnee(bool)` |

## 6 - La classe `CBateau`

Ajoutez une classe `CBateau` à votre projet .

Un `CBateau` devant être capable d'indiquer son type (promesse 11), il semble naturel de munir la classe d'une variable membre `m_nature` de type `ENature` qui permettra de stocker cette information . La liste des cases qu'il occupe doit, elle aussi, être stockée dans un `CBateau`. Ajoutez donc à cette classe une variable de type `QValueList` de `QPoint` nommée `m_lesCases` .

### Le constructeur à trois paramètres

Cette fonction est celle qui effectue l'opération la plus délicate : le positionnement d'un bateau dans l'espace de jeu. La stratégie utilisée est de tirer au hasard *(17)* une orientation (horizontale ou verticale) et une case *(18-20)*, puis de vérifier que toutes les cases convoitées sont disponibles *(21-26)*. Si ce n'est pas le cas, une nouvelle case est tirée (dans la limite de 100 tentatives). Etant donné qu'un bateau occupe plusieurs cases, il serait vain d'essayer d'en positionner un en partant trop près des limites de l'espace de jeu. Le tirage de la position s'effectue donc dans un espace réduit d'une `marge` dont la taille dépend du type du bateau à installer *(3-11)*. Cette marge est soustraite à la largeur de l'espace de jeu si une orientation horizontale a été choisie, et à sa hauteur dans le cas contraire.

```
1  CBateau::CBateau(ENature quoi, QValueList<QPoint> & interdits, QPoint max) :
                  m_nature(AUCUN)//ça ne sera un "quoi" que si on arrive à le construire...
2  {
3  int marge; //les bateaux occupent marge+1 cases
4  switch(quoi)
5     {//les bateaux de plus d'une case doivent commencer assez loin du bord
6     case PORTE_AVIONS : marge = 4; break;
7     case CROISEUR : marge = 3; break;
8     case CUIRASSE : marge = 2; break;
9     case TORPILLEUR : marge = 1; break;
10    default : marge = 0; break; //les SOUS_MARIN et les imprévus...
11    }
```

```
12  int tirages = 0;
13  while (m_lesCases.count() < 1 + marge)
14     {
15     if(tirages++ > 100)
16        return; //on n'y arrive pas !
17     bool horizontal = rand() % 2;
       //choix de la case de départ
18     int x = rand() % (max.x() - (horizontal ? marge : 0));
19     int y = rand() % (max.y() - (horizontal ? 0 : marge));
20     QPoint position(x,y);
       //vérifie que toutes les cases envisagées sont libres
21     m_lesCases.clear();
22     while(m_lesCases.count() < 1 + marge && !interdits.contains(position))
23        {
24        m_lesCases.append(position);
25        position += horizontal ? QPoint(1,0) : QPoint(0,1);
26        }
27     }
28  interdits += m_lesCases; //les cases choisies sont maintenant occupées
29  m_nature = quoi; //la mise en place a réussi !
30  }
```

Les cases occupées par le bateau sont enfin ajoutées à la liste des cases qui ne sont plus disponibles *(28)* et la réussite de la construction est signalée en conférant au bateau son véritable type *(31)*.

> Si le bateau n'a pas trouvé de place, l'exécution du constructeur s'achève à la ligne *16*, et le `CBateau` reste de type AUCUN, ce qui garantit que la fonction `nombreDe()` ne le prendra pas en compte lors du dénombrement des exemplaires du type qu'il aurait dû avoir.

Ajoutez cette fonction ☐ à la classe `CBateau`.

## La fonction `encaisse()`

Lorsqu'un tir survient, un bateau vérifie s'il est concerné en parcourant la liste des cases qu'il occupe. Il mémorise, au passage, la distance minimale observée entre le point d'impact et ces cases, car cette information doit faire partie de la conclusion qu'il renvoie.

```
1   CVerdict CBateau::encaisse(QPoint impact)
2   {
3   int distanceMini = INT_MAX;
4   QValueList <QPoint>::Iterator it;
5   for(it = m_lesCases.begin() ; it != m_lesCases.end() ; ++it)
6      {
7      int deltaX = abs(impact.x() - (*it).x());    //différence des abscisses
8      int deltaY = abs(impact.y() - (*it).y());    //différence des ordonnées
9      int d = (deltaX > deltaY) ? deltaX : deltaY; //la plus grande des deux
10     if (d == 0)
11        {//enfer et damnation !
12        m_lesCases.remove(it);   //retire l'élément détruit
13        return CVerdict(m_nature, m_lesCases.isEmpty() ? COULE : TOUCHE, 0);
14        }
15     if (d < distanceMini)
16        distanceMini = d;
17     }
18  return CVerdict(AUCUN, RATE, distanceMini);
19  }
```

## Le fichier bateau.h

Une fois ajoutée la fonction `encaisse()` ☐, il ne reste qu'à définir *(8)* la fonction `nature()` ☐ pour obtenir la version finale de la classe `CBateau` :

```
#include "qvaluelist.h"
#include "qpoint.h"
#include "typesEnum.h"
```

```
#include "verdict.h"
class CBateau
{
public:
   CBateau() : m_nature(AUCUN) {}
protected:
   CBateau(ENature quoi, QValueList<QPoint> & interdits, QPoint max);
   CVerdict encaisse(QPoint impact);
   ENature nature() const {return m_nature;}
   QValueList <QPoint> m_lesCases;
   ENature m_nature;
friend class CBataille;
};
```

Remarquez que seules les fonctions membre de `CBataille` ont vocation à manipuler des `CBateau`. On peut matérialiser cette restriction en réduisant à l'extrême l'interface de la classe et en déclarant amie la classe `CBataille`. La création d'une `QValueList` de `CBateau` exige toutefois qu'un constructeur par défaut reste public et, comme la présence du constructeur à trois arguments inhibe la création automatique d'un constructeur par défaut, il est indispensable d'en définir explicitement un *(4)*.

### Qu'avons-nous promis ?

| | Origine | Nature de la promesse |
|---|---|---|
| 15 | `encaisse()` | `CVerdict::CVerdict(ENature,EDegats,int)` |
| 16 | `encaisse()` | `void CVerdict::partieGagnee(EDegats)` |

## 7 - La classe `CVerdict`

Ajoutez une classe `CVerdict` à votre projet ☐.

### La fonction `message()`

Ajoutez à cette classe la fonction suivante ☐ (qui se passe aisément de commentaires...)

```
QString CVerdict::message() const
{
QString degats = (m_gravite == TOUCHE) ? "%1 touché" : "%1 coulé";
if(m_partieGagnee)
   degats += "\nPARTIE GAGNEE !";
switch(m_victime)
   {
   case AUCUN : return "Dans l'eau !";
   case SOUS_MARIN : return degats.arg("sous-marin");
   case TORPILLEUR : return degats.arg("torpilleur");
   case CUIRASSE : return degats.arg("cuirassé");
   case CROISEUR : return degats.arg("croiseur");
   case PORTE_AVIONS : return degats.arg("porte-avions");
   default : return "type de bateau inconnu";
   }
}
```

### Le fichier verdict.h

Par nature, les instances de `CVerdict` sont des objets qui ne permettent que la lecture des informations qu'ils contiennent. L'interface de cette classe ne contient donc, logiquement, que des fonctions de lecture. Il se trouve cependant que, dans notre programme, l'instanciation de cette classe est effectuée par quelqu'un (la fonction `f_feu()` de la classe de dialogue, en l'occurrence) qui n'est évidemment pas en mesure de fixer par initialisation l'état définitif de l'objet créé. Cet état est calculé d'une part par l'unique instance de la classe `CBataille` (qui est seule à pouvoir déterminer quel est le bateau le plus proche d'un point d'impact et si la partie est terminée...) et d'autre part par l'instance de `CBateau` qui est éventuellement atteinte par le tir (cet objet est le seul à connaître son propre type et à pouvoir déterminer la gravité de sa situation). Pour rendre ces opérations possibles sans compromettre l'interdiction faite aux fonctions membre de la classe

de dialogue de modifier l'état d'un `CVerdict`, les classe `CBateau` et `CBataille` sont déclarées amies *(16-17)*.

```
#include "qstring.h"
#include "typesenum.h"
1  class CVerdict
2  {
3  public:
4     int distance() const {return m_distance;}
5     bool partieGagnee() const {return m_partieGagnee;}
6     ENature victime() const {return m_victime;}
7     EDegats gravite() const {return m_gravite;}
8     QString message() const;
9  protected:
10    CVerdict(ENature v, EDegats dg, int d) :
                m_victime(v), m_gravite(dg), m_distance(d), m_partieGagnee(false){}
11    void partieGagnee(bool v) {m_partieGagnee = v;}
12    ENature m_victime;
13    EDegats m_gravite;
14    int m_distance;
15    bool m_partieGagnee;
16 friend class CBataille;
17 friend class CBateau;
18 };
```

Complétez la définition de la classe `CVerdict` ☐, compilez le programme ☐, et amusez-vous bien ☐...

## 8 - Exercices

1) A quel moment la classe `CBataille` est-elle instanciée ? Quel est le constructeur qui est exécuté à cette occasion ?

2) Combien la fonction `CVerdict::partieGagnee()` a-t-elle de paramètres ?

3) Le constructeur à trois paramètres de `CVerdict` est protégé et n'est donc accessible qu'aux amies de cette classe. Comment la fonction `TD10DialogImpl::f_feu()` peut-elle donc disposer d'une variable locale de type `CVerdict` ? Quelles sont les opérations que cette fonction est en mesure d'effectuer sur la variable en question ?

4) Modifiez le programme de façon à ce que l'utilisateur puisse choisir le nombre de bateaux de chaque type installés en début de partie ainsi que les nombres de lignes et de colonnes de l'espace de jeu.