Même lorsqu'un programme est au point, certaines circonstances exceptionnelles peuvent compromettre la poursuite de son exécution ; il peut s'agir par exemple de données incorrectes ou de la rencontre d'une fin de fichier prématurée (alors que nous avons besoin d'informations supplémentaires pour continuer le traitement).

Les exceptions sont donc des anomalies qu'un programme détecte en cours d'exécution, telles des divisions par 0, un accès à l'extérieur des bornes d'un tableau ou l'épuisement de la mémoire. De telles exceptions sortent du fonctionnement normal du programme et requièrent de sa part une gestion immédiate.

Bien entendu, on peut toujours essayer d'examiner toutes les situations possibles au sein du programme et prendre les décisions qui s'imposent. Mais outre le fait que le concepteur du programme risque d'omettre certaines situations, la démarche peut devenir très vite fastidieuse et les codes quelque peu complexes. Le programme peut être rendu quasiment illisible si sa tâche principale est masquée par de nombreuses instructions de traitement de circonstances exceptionnelles.

Le langage C++ dispose d'un mécanisme très souple nommé gestion d'exception, qui permet à la fois :

1. de dissocier la détection d'une anomalie de son traitement,
2. de séparer la gestion des anomalies du reste du code, donc de contribuer à la lisibilité des programmes.

Ce mécanisme s'effectue toujours en deux temps.

1. Nous avons d'abord, la détection de l'anomalie. Le développeur doit alors avertir du disfonctionnement en provoquant une rupture de séquence par le déclenchement d'une exception correspondant à l'anomalie. Cette phase s'appelle souvent « *lever* » ou « *lancer* » une exception. Nous lançons une exception par la directive « *throw* ».
2. Ensuite, si nous le désirons, nous pouvons nous occuper de ce qui s'appelle la « *gestion d'exception* », qui consiste à proposer un certain nombre d'actions pour gérer le problème lié à une ou plusieurs anomalies. Généralement, nous tentons d'abord d'effectuer le traitement prévu, et si cela se passe mal, nous capturons l'exception « *lancée* » par la directive « *throw* ». Nous proposons alors une alternative correspondant au type de l'exception. En fait, et plus précisément, chaque exception est caractérisée par un type, et le choix du bon gestionnaire se fait en fonction de la nature de l'expression mentionnée à « *throw* ». Cette phase est réalisée par les directives « *try – catch* » (essayer et capturer).

Remarque

Généralement, ces deux phases sont traitées par deux développeurs différents. En effet, le premier construit les classes. Celui-ci doit alors prévoir tous les cas où la classe peut être mal utilisée. Il doit donc proposer un ensemble d'exceptions correspondant aux dysfonctionnements possibles. Le second est celui qui utilise les classes. Celui-ci doit gérer les exceptions suivant l'utilisation qu'il fait de ces classes. Ainsi, la gestion d'exception peut être totalement différente suivant l'utilisateur et surtout suivant le programme à traiter. Le fait d'avoir deux phases permet de simplifier considérablement la situation, chaque programmeur s'occupe de son propre problème.

Détecter les anomalies

La première démarche, et ce n'est pas toujours la plus facile, consiste à recenser toutes les anomalies possibles au sein d'une classe, dues généralement, à une mauvaise manipulation de la part du programmeur qui l'utilise. Pour illustrer ces propos, je vous propose de revenir sur l'étude d'un tableau d'entier.

Dans l'exemple ci-contre, nous voyons apparaître deux anomalies possibles :

1. D'une part, une mauvaise proposition d'indice qui nous envoie en dehors des limites du tableau.
2. D'autre part, une taille de tableau négative.

Quand ce genre de problème arrive, il est préférable de tout arrêter plutôt que de faire n'importe quoi et d'accéder à une partie de la mémoire qui n'est pas prévue.

```
2  class Tableau
3  {
4    int *valeur;
5    int taille;
6  public:
7    Tableau(int nombreElement)
8    {
9      valeur = new int[taille = nombreElement];
10   }
11   int& operator[] (int indice)
12   {
13     return valeur[indice];
14   }
15   ~Tableau() { delete[] valeur; }
16 };
17 //---------------------------------------------------------------
18 int main( int argc, char * argv[] )
19 {
20   Tableau tab(10);     // création d'un tableau de 10 cases
21   tab[5] = 3;          // tout se passe bien dans cette instruction
22   tab[-5] = tab[15];   // Attention, deux erreurs d'indice
23   Tableau faux(-12);   // Attention, erreur dans la création
24   return 0;
25 }
```

Lever une exception

Éventuellement, celui qui construit la classe pourrait envisager de proposer une solution dans le cas, par exemple, où l'utilisateur tente d'accéder à une case du tableau au delà des limites prévues. Mais alors, que choisir comme indice. Le concepteur de la classe ne sait pas ce que l'utilisateur désire réellement faire. Il est préférable que le concepteur de la classe laisse l'initiative à l'utilisateur et juste le prévenir qu'il y a un problème.

Pour prévenir l'utilisateur, il faut lever une exception qui correspond à l'anomalie. Pour cela, nous devons utiliser l'instruction « *throw* » suivi d'une valeur d'un type quelconque. Nous pouvons, par exemple, proposer une valeur numérique entière qui indique l'erreur correspondant à l'anomalie,

```
2  class Tableau
3  {
4    int *valeur;
5    int taille;
6  public:
7    Tableau(int nombreElement)
8    {
9      if (nombreElement<0) throw -1;
10     valeur = new int[taille = nombreElement];
11   }
12   int& operator[] (int indice)
13   {
14     if (indice<0 || indice>=taille) throw -2;
15     return valeur[indice];
16   }
17   ~Tableau() { delete[] valeur; }
18 };
```

comme *-1* pour le problème de construction, et *-2* pour le problème lié à l'indice. Ceci dit, cette démarche n'est pas très élégante.

Puisque nous avons le choix, nous pouvons fabriquer un nouveau type. Il est, en effet, préférable d'utiliser une énumération dont chacun des énumérateurs donne explicitement le type de l'erreur.

```
enum Erreur {Creation, Indice};
//---------------------------------------
class Tableau
{
  int *valeur;
  int taille;
public:
  Tableau(int nombreElement)
  {
    if (nombreElement<0) throw Creation;
    valeur = new int[taille = nombreElement];
  }
  int& operator[] (int indice)
  {
    if (indice<0 || indice>=taille) throw Indice;
    return valeur[indice];
  }
  ~Tableau() { delete[] valeur; }
```

Nous aurions pu aussi proposer une chaîne de caractères avec un message adapté au type de l'anomalie.

Bref, nous pouvons utiliser n'importe quel type, et en créer de nouveaux spécialement adaptés à la situation. Nous verrons d'ailleurs que le mieux sera de créer carrément une classe correspondant à chaque type d'erreur. Si vous le faites, la valeur à envoyer dans l'exception est alors un objet.

En prenant cette solution, il suffit de fabriquer des classes sans rien à l'intérieur, le but est juste de créer des nouveaux types adaptés aux anomalies rencontrées.

```
class ErreurCreation { };
class ErreurIndice { };
//---------------------------------------
class Tableau
{
  int *valeur;
  int taille;
public:
  Tableau(int nombreElement)
  {
    if (nombreElement<0) throw ErreurCreation();
    valeur = new int[taille = nombreElement];
  }
  int& operator[] (int indice)
  {
    if (indice<0 || indice>=taille) throw ErreurIndice();
    return valeur[indice];
  }
  ~Tableau() { delete[] valeur; }
};
```

Création d'un objet anonyme avec un appel du constructeur par défaut

Lorsque une exception est lancée, il faut créer un objet de la classe considérée. Il n'est absolument pas nécessaire de spécifier un nom à l'objet, puisque la plupart du temps cet objet sert uniquement d'élément de propagation de l'exception. Il peut donc être anonyme.

N'oubliez pas qu'une classe, même a priori sans rien dedans, offre un comportement minimum et possède donc un constructeur par défaut (qui ne fait rien). La création de l'objet passera par l'appel de ce constructeur.

Que se passe-t-il lorsqu'une exception est levée ?

En fait, tout dépend si nous gérons l'exception ou pas (à l'aide du bloc « *try–catch* »). Si ce n'est pas le cas, l'exception provoque l'arrêt pur et simple du programme. De toute façon, ce n'est pas la peine d'aller plus loin, puisque si une exception est levée sans être gérée, nous nous trouvons alors dans une situation plutôt catastrophique.

Conclusion

Le développeur qui fabrique les classes doit s'occuper de recenser l'ensemble des anomalies possibles et lance des exceptions correspondant à ces disfonctionnements. Pour cela, il fabrique une classe d'erreur par type d'anomalie. Ensuite, c'est tout, son travail est terminé.

Interception et gestion des exceptions

Lorsqu'une exception est levée, plutôt que d'avoir un programme qui se termine de façon abrupte, il serait souhaitable de maîtriser la situation et de proposer une alternative de fonctionnement. Pour cela, il faut mettre en œuvre ce que l'on appelle une gestion d'exception qui se déroule finalement en trois phases :

1. *Tentative* d'exécution d'un ensemble d'instructions,
2. *Capture* de l'exception, si un problème est rencontré durant cette tentative,
3. *Gestion* de l'exception en proposant une nouvelle suite d'instructions.

Tentative

Pour intercepter et gérer les exceptions possibles, vous devez d'abord entourer les instructions qui sont susceptibles de lever des exceptions par un bloc *try*. Un bloc *try* commence par le mot clé *try* suivi d'une séquence d'instructions entourées d'accolades.

Le bloc *try* est suivi d'une liste de gestionnaires appelés clauses *catch*. En fait, le bloc *try* regroupe un ensemble d'instructions et leur associe un ensemble de gestionnaires pour gérer les exceptions que peuvent lever les instructions.

Dans notre exemple, deux clauses *catch* sont associées au bloc *try*. Le nombre de clauses *catch* dépend du nombre de type d'erreur possible lors de l'exécution d'un ensemble d'instructions. Nous avons recensé deux anomalies possibles. Nous devons donc pouvoir capturer ces deux types d'anomalies grâce à des clauses *catch* adaptées.

Si aucune exception ne survient, l'ensemble du code à l'intérieur du bloc *try* est exécuté et les gestionnaires associés au bloc *try* sont ignorés. Le programme exécute ensuite les instructions qui sont placées à la suite des clauses *catch*.

Si une exception est levée à l'intérieur d'un bloc *try*, les instructions qui suivent l'instruction lançant l'exception ne sont pas exécutées. L'exécution du programme reprend dans la clause *catch* gérant l'exception.

Interception et gestion d'une exception

Un gestionnaire d'exception C++ est une clause *catch*. Quand une exception est levée depuis des instructions dans un bloc *try*, la liste des clauses *catch* qui suit le bloc *try* est recherchée afin d'y trouver une clause *catch* qui soit capable de gérer l'exception.

Une clause *catch* se compose de trois parties :

1. le mot clé *catch*,
2. la déclaration d'un type unique ou d'un objet unique entre parenthèses (appelée déclaration d'exception),
3. et un ensemble d'instructions dans une instruction composée (accolades).

Si la clause *catch* est sélectionnée pour gérer une exception, l'instruction composée est exécutée.

Dès qu'une clause *catch* a terminée son travail, l'exécution du programme continue sur l'instruction qui suit la dernière clause *catch* de la liste.

Le mécanisme de gestion des exceptions du C++ est dit sans reprise ; une fois l'exception gérée, l'exécution du programme ne reprend pas là où l'exception a été levée.

```
4  class ErreurCreation { };
5  class ErreurIndice { };
6  //-----------------------------------------------------------
7  class Tableau
8  {
9    int *valeur;
10   int taille;
11 public:
12   Tableau(int nombreElement)
13   {
14     if (nombreElement<0) throw ErreurCreation();
15     valeur = new int[taille = nombreElement];
16   }
17   int& operator[] (int indice)
18   {
19     if (indice<0 || indice>=taille) throw ErreurIndice();
20     return valeur[indice];
21   }
22   ~Tableau() { delete[] valeur; }
23 };
24 //-----------------------------------------------------------
25 int main( int argc, char * argv[] )
26 {
27   try {
28     Tableau tab(10);
29     tab[5] = 3;
30     Tableau faux(-12);
31     tab[-5] = tab[15];
32   }
33   catch (ErreurCreation) { cerr << "Problème à la création" << endl; }
34   catch (ErreurIndice)   { cerr << "Mauvais indice" << endl; }
35   cout << "Le programme se termine";
36   return 0;
37 }
```

Déclenchement de l'exception
Tentative
Arrêt de lecture dans le bloc try puisqu'une exception est déclenchée
Cette ligne n'est pas lue
Gestion de l'exception
Le programme se poursuit ici
Capture de l'exception

Dans notre exemple, notre gestion d'exception consiste juste à un affichage des erreurs. Vous pouvez, bien entendu, placer vos blocs *try-catch* n'importe où, notamment à l'intérieur d'une itérative. Ainsi vous serez à même de faire une véritable gestion et de proposer éventuellement d'autres valeurs d'indices pour l'occurrence suivante. Il est également possible de proposer des blocs *try-catch* imbriqués les uns dans les autres, soit dans la même méthode, soit par le fait qu'une méthode appelle une autre méthode, chacune disposant d'un bloc *try-catch*.

```
39 int main()
40 {
41   bool correcte;
42   try {
43     int dimension;
44     cout << "Dimension de votre tableau : ";
45     cin >> dimension;
46     Tableau tableau(dimension);
47     do {
48       correcte = true;
49       try {
50         int indice, valeur;
51         cout << "Dans quelle case voulez-vous introduire la valeur : ";
52         cin >> indice;
53         cout << "Valeur : "; cin >> valeur;
54         tableau[indice] = valeur;
55       }
56       catch (ErreurIndice) {
57         cerr << "Attention, votre indice n'est pas correct" << endl;
58         correcte = false;
59       }
60     } while (!correcte);
61   }
62   catch (ErreurCreation) { cerr << "Dimension du tableau négative"; }
63   return 0;
64 }
```

```
Dimension de votre tableau : 10
Dans quelle case voulez-vous introduire la valeur : -2
Valeur : 12
Attention, votre indice n'est pas correct
Dans quelle case voulez-vous introduire la valeur : 15
Valeur : 45
Attention, votre indice n'est pas correct
Dans quelle case voulez-vous introduire la valeur : 5
Valeur : 45
```

Déroulement de pile

La recherche d'une clause *catch* pour gérer une exception levée se déroule ainsi. Si l'expression *throw* se trouve dans un bloc *try*, les clauses *catch* associées à ce bloc sont examinées pour voir si l'une d'elles peut gérer l'exception.

Si une clause *catch* est détectée, l'exception est gérée. Si aucune clause *catch* n'est détectée, la recherche se poursuit dans le bloc *try-catch* de niveau supérieur (celui qui englobe le *try-catch* imbriqué).

Si une clause *catch* est trouvée dans ce nouveau bloc, l'exception est gérée sinon la recherche se poursuit à un niveau encore supérieur. Ce processus se poursuit en remontant l'imbrication des blocs *try-catch* jusqu'à ce qu'une clause *catch* pour l'exception soit trouvée. Dès qu'une clause *catch* pouvant gérer l'exception est rencontrée, on entre dans la clause *catch* et l'exécution du programme continue dans ce gestionnaire.

Si aucun gestionnaire n'est trouvé, le programme appelle la fonction *terminate()* définie dans la bibliothèque du C++ standard. Cette fonction propose un comportement par défaut, qui appelle notamment la fonction *abort()* qui elle-même indique que le programme se termine anormalement « *Abnormal program termination* ».

Propager une exception

Il est possible qu'une clause unique ne puisse pas gérer une exception complètement. Après quelques actions correctives, une clause *catch* peut décider que l'exception sera gérée par un bloc *try-catch* de niveau supérieur. Il suffit pour cela de propager l'exception. Dans un gestionnaire, l'instruction *throw* (sans expression) retransmet (propage) l'exception au niveau englobant.

```
catch (ErreurIndice) {
  cerr << "Attention, votre indice n'est pas correct" << endl;
  correcte = false;
  throw;
}
```

Propagation vers le niveau englobant par redéclenchement de l'exception

Si nous utilisons cette technique, il faut, bien entendu, que le bloc supérieur soit capable de capturer ce type d'exception et qu'il dispose donc du même gestionnaire.

Gestionnaire pour toutes les exceptions

Au lieu de proposer un gestionnaire par type d'anomalies possibles, vous pouvez capturer toutes les exceptions dans une seule clause *catch*. Cette clause *catch* possède une déclaration d'exception de la forme *(…)*, où les trois points sont une *ellipse*.

```
try {
   Tableau tab(10);
   tab[5] = 3;
   Tableau faux(-12);
   tab[-5] = tab[15];
 }
 catch (ErreurCreation) { cerr << "Problème à la création" << endl; }
 catch (ErreurIndice)   { cerr << "Mauvais indice" << endl; }
 catch (...)            { cerr << "Autre problèmes" << end; }
```

Vous pouvez aussi combiner les exceptions en gérant quelques unes plus précisément et les autres de façon globale en utilisant alors la clause avec *l'ellipse*. Cela implique qui si un *catch (…)* est combiné avec d'autres clauses *catch*, il sera toujours placé en dernier de la liste des gestionnaires d'exception. En effet, les clauses *catch* sont examinées à tour de rôle, dans l'ordre où elles apparaissent à la suite du bloc *try*. Si les clauses *catch* particulières se trouvaient après la clause *catch* comportant *l'ellipse*, elles ne seraient jamais atteintes.

Conclusion

Le développeur qui utilise les classes ne s'occupe pas du tout de recenser les anomalies possibles. Il doit juste proposer un certain nombre d'alternatives en gérant les exceptions qui peuvent être levées suivant les tentatives qu'il propose. Vous remarquez que par cette disposition, chacun s'occupe de son propre domaine, ce qui simplifie notablement le travail.

Spécification d'exception

```
 7 class Tableau
 8 {
 9   int *valeur;
10   int taille;
11 public:
12   Tableau(int nombreElement);
13   int& operator[] (int indice);
14   ~Tableau();
15 };
```

Les exceptions ne sont plus visibles

Dans la déclaration de la classe *Tableau* que j'ai proposé, les méthodes sont directement définies, ce qui permet de visualiser les exceptions qui sont levées. Toutefois, la plupart du temps, les définitions des méthodes se font à l'extérieur de la déclaration de la classe, ce qui offre d'ailleurs une meilleure lisibilité. Dans ce cas là, malheureusement, il n'est plus possible de déterminer que ces méthodes peuvent éventuellement lever une exception.

```
 7 class Tableau
 8 {
 9   int *valeur;
10   int taille;
11 public:
12   Tableau(int nombreElement) throw (ErreurCreation);
13   int& operator[] (int indice) throw (ErreurIndice);
14   ~Tableau();
15 };
```

Spécifications d'exception

La spécification d'exception offre une solution pour lister les exceptions qu'une méthode peut lever en même temps que la déclaration de la méthode. Elle assure (vérifié par le compilateur) que la méthode ne lance aucun autre type d'exception.

Une spécification d'exception suit la liste des paramètres de la méthode. Elle est déclarée avec le mot clé *throw*, suivi d'une liste des types d'exception entourée de parenthèses.

Une spécification d'exception est un contrat entre la méthode et le reste du programme. Elle garantit que la méthode ne lèvera pas d'exception non listée dans sa spécification d'exception.

Les objets exception

Nous avons vus que la déclaration d'exception d'une clause *catch* peut être soit une déclaration de type, soit une déclaration d'objet. Quand la déclaration d'exception dans une clause *catch* déclare-t-elle un objet ? Un objet sera déclaré lorsque nous devons obtenir la valeur ou manipuler l'objet exception créé par l'expression *throw*.

En effet, jusqu'à présent, les classes d'exception que nous avons créées étaient réduites à leurs plus simples expressions. Mais il s'agit de classes à part entière, comme les autres, et rien n'empêche de les créer de façon beaucoup plus sophistiquées avec un certain nombre d'attributs et de méthodes. Il est même possible de structurer tout une hiérarchie de classes d'erreur.
Il peut être utile de fabriquer des classes d'erreur plus complètes afin de stocker, par exemple, la valeur qui a provoqué l'erreur ainsi que les valeurs limites qu'impose le bon fonctionnement des classes normales.

```
Dimension de votre tableau : 5
Dans quelle case voulez-vous introduire la valeur : 10
Valeur : -55
Attention, votre valeur 10 n'est pas correcte. Elle doit être comprise entre 0 et 5
Dans quelle case voulez-vous introduire la valeur :
```

```
class ErreurCreation
{
  int valeurIntroduite;
public:
  ErreurCreation(int valeur)   { valeurIntroduite = valeur; }
  int getValeurIntroduite()    { return valeurIntroduite; }
};
//-----------------------------------------------------------
class ErreurIndice
{
  int valeurIntroduite;
  int limite;
public:
  ErreurIndice(int valeur, int taille) {
    valeurIntroduite = valeur;
    limite = taille;
  }
  int getValeurIntroduite()    { return valeurIntroduite; }
  int getLimite()              { return limite; }
};
```

```
class Tableau
{
  int *valeur;
  int taille;
public:
  Tableau(int nombreElement) throw (ErreurCreation)
  {
    if (nombreElement<0) throw ErreurCreation(nombreElement);
    valeur = new int[taille = nombreElement];
  }
  int& operator[] (int indice) throw (ErreurIndice)
  {
    if (indice<0 || indice>=taille) throw ErreurIndice(indice, taille);
    return valeur[indice];
  }
  ~Tableau() { delete[] valeur; }
};
```

Cette fois-ci, pendant la création de l'objet d'erreur, nous précisons le nombre d'éléments souhaités

Ici, nous récupérons l'indice souhaité, et la taille maximale que peut contenir le tableau d'entier

Cette fois-ci, lorsque nous capturons un objet d'erreur, il est beaucoup plus complet. Nous pouvons donc faire une analyse plus fine et avertir l'utilisateur de son erreur par rapport au contexte.

Dans cet exemple, nous montrons à l'utilisateur la valeur que lui-même a saisie et ce que le système attend réellement.

La déclaration d'exception au niveau des clauses *catch* ressemble à un paramètre d'une méthode.

Pour prévenir les copies inutiles d'objet de classes de grande taille, il est préférable que les déclarations d'exception soient déclarées en tant que référence.

```
int main()
{
  bool correcte;
  try {
    int dimension;
    cout << "Dimension de votre tableau : ";
    cin >> dimension;
    Tableau tableau(dimension);
    do {
      correcte = true;
      try {
        int indice, valeur;
        cout << "Dans quelle case voulez-vous introduire la valeur : ";
        cin >> indice;
        cout << "Valeur : "; cin >> valeur;
        tableau[indice] = valeur;
      }
      catch (ErreurIndice &erreur) {
        cerr << "Attention, votre valeur " << erreur.getValeurIntroduite();
        cerr << " n'est pas correcte. Elle doit être comprise";
        cerr << " entre 0 et " << erreur.getLimite() << endl;
        correcte = false;
      }
    } while (!correcte);
  }
  catch (ErreurCreation &erreur) {
    cerr << "La dimension du tableau doit être positive" << endl;
    cerr << "La valeur que vous avez passé est la suivante : ";
    cerr << erreur.getValeurIntroduite();
  }
  return 0;
}
```

Hiérarchie de classes d'erreur

Je vais continuer mes investigations en proposant, cette fois-ci, une hiérarchie de classes polymorphiques, juste pour montrer toutes les possibilités et la souplesse du langage C++.

Dans l'exemple qui suit, nous construisons une classe de base abstraite où il sera nécessaire de redéfinir la méthode *getMessage* qui délivrera le message correspondant à l'objet levé.

Pour lever une exception, rien ne change, il suffit de créer l'objet relatif à la classe qui correspond au défaut détecté.

Pour la capture, cela peut être, finalement, beaucoup plus simple. Il suffit, en effet, de faire une capture par rapport à une référence sur la classe de base uniquement. Le mécanisme du polymorphisme permettra de récupérer le bon objet exception.

```
int main( int argc, char * argv[] )
{
  try {
    Tableau tab(10);
    tab[5] = 3;
    Tableau faux(-12);
    tab[-5] = tab[15];
  }
  catch (ErreurTableau &erreur) { cerr << erreur.getMessage(); }
  catch (...)   { cerr << "Pas assez de mémoire" << endl; }
  cout << "Le programme se termine";
  return 0;
}
```

Cette fois-ci, nous pouvons globaliser les erreurs issues du tableau par une référence à la classe de base. C'est l'objet levé par l'exception qui déterminera le type d'erreur.

```
class ErreurTableau
{
  int valeurIntroduite;
public:
  ErreurTableau(int valeur)   { valeurIntroduite = valeur; }
  int getValeurIntroduite()   { return valeurIntroduite; }
  virtual string getMessage() = 0;
};
//-----------------------------------------------------------
class ErreurCreation : public ErreurTableau
{
public:
  ErreurCreation(int valeur) : ErreurTableau(valeur) { }
  string getMessage() {
    ostringstream message;
    message << "La dimension du tableau doit être positive" << endl;
    message << "La valeur que vous avez passé est la suivante : ";
    message << getValeurIntroduite() << endl;
    return message.str();
  }
};
//-----------------------------------------------------------
class ErreurIndice : public ErreurTableau
{
  int limite;
public:
  ErreurIndice(int valeur, int taille) : ErreurTableau(valeur) {
    limite = taille;
  }
  int getLimite() { return limite; }
  string getMessage() {
    ostringstream message;
    message << "Attention, votre valeur " << getValeurIntroduite();
    message << " n'est pas correcte. Elle doit être comprise";
    message << " entre 0 et " << limite << endl;
    return message.str();
  }
};
```

Classe abstraite

Redéfinition de la méthode adaptée à la situation

Redéfinition de la méthode adaptée à la situation