



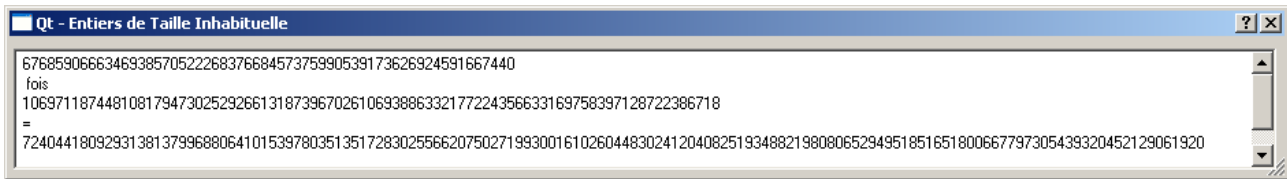
Centre Informatique pour les Lettres
et les Sciences Humaines

C++ - TD 17

Entiers de Taille Inhabituelle

1 - Réflexions préalables.....	2
2 - Interface et classe de dialogue.....	2
3 - La classe ETI et ses variables membre.....	3
4 - Création et destruction d'Entiers de Taille Inhabituelle.....	3
La construction par défaut et la fonction baseAlloc().....	3
La construction par copie et la fonction nbOctetsUtils().....	4
Transtypage à partir d'une séquence de chiffres.....	5
Transtypage à partir d'un int.....	5
La fonction statique hasard() et la fonction redim().....	6
Destruction.....	6
5 - Affectation d'une valeur à un Entier de Taille Inhabituelle.....	7
6 - Comparaisons entre Entiers de Taille Inhabituelle.....	7
Les tests d'égalité et de différence.....	7
Les tests d'infériorités et la fonction abs().....	8
Les tests de supériorité.....	9
7 - Additions d'Entiers de Taille Inhabituelle.....	9
L'addition sans effet et la fonction ajoute().....	9
Les additions à effet.....	10
8 - Soustractions d'Entiers de Taille Inhabituelle.....	10
La soustraction sans effet et la fonction enleve().....	10
Les soustractions à effet.....	11
9 - Multiplications d'Entiers de Taille Inhabituelle.....	11
La multiplication sans effet et la fonction foisDeux().....	11
La multiplication à effet.....	12
10 - Divisions d'Entiers de Taille Inhabituelle.....	13
Les divisions sans et avec effet et l'opérateur %.....	13
La fonction divise().....	13
11 - Représentation textuelle d'un Entier de Taille Inhabituelle.....	14
12 - Le fichier eti.h.....	15

Le programme que nous allons réaliser au cours de ce TD tire deux nombres entiers au hasard et les affiche, ainsi que leur produit :



Si ce résultat manque évidemment d'intérêt, la mise au point du programme va nous conduire à définir une classe ETI, capable de gérer des valeurs entières largement supérieures à celles manipulables à l'aide des types prédéfinis `int` ou `long int`. Cette classe est un exemple d'encapsulation de l'allocation dynamique et peut être utilisée dans d'autres projets. Notez, toutefois, qu'il s'agit d'un **simple exercice** et que le code proposé ici ne prétend offrir ni la fiabilité ni l'efficacité des bibliothèques (telles que la [GNU Multiple Precision Arithmetic Library](#), par exemple) destinées à être utilisées dans de véritables applications.

L'absence de prétentions de la classe ETI est soulignée par son nom, clin d'oeil aux RTI (Rongeurs de Taille Inhabituelle) du film "[The Princess Bride](#)".

1 - Réflexions préalables

La représentation d'une valeur entière exige une quantité de mémoire qui dépend de la taille de cette valeur. Dans le cas des types prédéfinis, la taille est fixée arbitrairement et tous les objets du même type ont la même taille. La représentation d'une petite valeur laisse donc un certain nombre de bits inutilisés, et la taille choisie impose une limite aux valeurs qui peuvent être représentées (4 294 967 295 pour un entier non signé représenté sur 4 octets, par exemple).

Pour un type visant à permettre de manipuler de très grands nombres, cette approche n'est pas très prometteuse : une taille fixe conduit à la fois à accepter une limite (qui risque de s'avérer tôt ou tard trop basse) et à mobiliser beaucoup de mémoire sans l'utiliser réellement lorsque les valeurs représentées sont très inférieures à cette limite.

La classe ETI adopte donc un fonctionnement plus proche de celui des `QString` que de celui des `int` : la place utilisée pour représenter une valeur est variable et s'ajuste automatiquement en fonction des besoins.

2 - Interface et classe de dialogue

Utilisez la procédure habituelle (cf. [aide mémoire](#)) pour créer un projet nommé TD17 ☐ et effacez tous les widgets présents dans le dialogue ☐.

Placez sur le dialogue un `QTextEdit` (Menu Tools, catégorie Display) ☐ nommé `ecran` ☐.

N'oubliez pas d'enregistrer votre travail avant de revenir à Visual C++ ☐.

Le code nécessaire est très bref et, comme nous souhaitons simplement qu'il s'exécute lorsque le programme est lancé, nous le plaçons directement dans le constructeur du dialogue :

```
1 TD17DialogImpl::TD17DialogImpl(QWidget* parent, const char* name, bool modal, WFlags f )
   : TD17Dialog( parent, name, modal, f )
2 {
3   srand(time(0));
4   ETI a = ETI::hasard(25);
5   ETI b = ETI::hasard(35);
6   ETI produit = a * b;
7   écran->setText(a.texte() + "\nfois\n" + b.texte() + "\n=\n" + produit.texte());
8 }
```

Remarquez que :

- Les variables `a` et `b` sont initialisées avec des valeurs aléatoires renvoyées par une **fonction statique** dont le paramètre indique le **nombre d'octets** que cette valeur peut occuper en mémoire. Il faut donc que la classe ETI fournisse un **constructeur par copie** et une fonction **hasard()**.

- La variable `produit` est initialisée avec une valeur de type ETI obtenue en multipliant `a` par `b`. Il faut donc que la classe ETI fournisse un opérateur de multiplication.
- Les trois instances de ETI sont utilisées pour appeler une fonction qui renvoie la séquence de chiffres qui représente leur valeur dans la notation positionnelle en base 10 utilisant les chiffres arabes.

Donnez à votre constructeur de dialogue le contenu suggéré ci-dessus ☐.

3 - La classe ETI et ses variables membre

Ajoutez à votre projet une classe nommée ETI et ne dérivant d'aucune autre ☐.

La gestion d'une zone de mémoire de taille variable pour représenter la valeur de l'objet impose l'utilisation de deux variables membre : l'une devra stocker la taille de la zone, l'autre son adresse. La première de ces variables, nommée `m_taille`, est de type `unsigned int`.

D'un point de vue purement théorique, adopter un type prédéfini pour cette variable revient à fixer une limite aux valeurs représentables par la classe ETI. Pratiquement, toutefois, aucun ordinateur actuel ne pourrait faire quoi que ce soit avec une quantité entière dont la représentation occuperait 4 294 967 295 octets...

L'adresse de la zone où est représentée la valeur doit évidemment être stockée dans un pointeur, mais de quel type ? Lorsque, comme c'est le cas ici, nous voulons que le langage considère les données comme de simples états électriques et laisse au programme l'entière responsabilité de leur donner une interprétation, le plus simple est de considérer que les données sont une collection de valeurs de type `unsigned char`.

Il n'existe pas de type `byte` (octet) en C++, et les programmeurs ont depuis longtemps pris l'habitude d'utiliser des `unsigned char` à la place. Le seul réel inconvénient de cette façon de procéder est la connotation textuelle qu'a le mot `char` pour la plupart des débutants.

Si nous souhaitons pouvoir manipuler des quantités négatives, il nous faut également mettre en place un système de mémorisation du signe. Une approche simple est d'introduire une variable membre de type booléen, qui indiquera si le nombre est positif.

Les types prédéfinis ne procèdent habituellement pas à un stockage séparé du signe et de la valeur absolue. La technique qu'ils utilisent (cf. [Leçon 1](#)) ne présente toutefois guère d'intérêt dans le cas de valeurs qui ne sont pas toutes représentées sur le même nombre d'octets.

Ajoutez à votre classe ETI les trois variables membre suivantes ☐.

```
1 unsigned int m_taille;  
2 unsigned char * m_valeur;  
3 bool m_positif;
```

4 - Création et destruction d'Entiers de Taille Inhabituelle

Même si l'état actuel du programme ne l'exige pas explicitement, il est évident qu'un constructeur par défaut est pratiquement indispensable à la classe ETI (ne serait-ce que pour nous permettre de ranger des valeurs de ce type dans une `QMap` ou une `QValueList`). Il est également vraisemblable que la plupart des utilisations de la classe ETI impliqueront des variables initialisées avec des valeurs non aléatoires. Deux méthodes d'initialisation semblent particulièrement attrayantes : à partir d'un entier (lorsque la valeur le permet) et à partir d'une chaîne de chiffres décrivant la valeur (lorsque celle-ci excède les possibilités des types prédéfinis). Deux constructeurs de transtypage nous permettront d'écrire des choses comme :

```
1 ETI uneValeur = 1; //transtypage à partir d'un int  
2 ETI uneAutre = "999 999 999 999"; //transtypage à partir d'une chaîne
```

La construction par défaut et la fonction `baseAlloc()`

La construction par défaut produit une instance nulle en mettant à zéro (4-6) tous les octets de la zone consacrée à la représentation de la valeur. Cette zone est réservée en faisant appel à une fonction de service (qui sera également utilisée par d'autres constructeurs ou fonctions membre).

```

1 ETI::ETI() : m_positif(true), m_taille(0), m_valeur(NULL)
2 {
3     baseAlloc();
4     unsigned int p;
5     for(p = 0 ; p < m_taille ; ++p)
6         m_valeur[p] = 0;
7 }

```

La fonction `baseAlloc()` procède à une allocation dynamique (3) entourée des précautions habituelles (4-9). Si cette opération réussit, elle stocke dans `m_taille` le nombre d'octets disponibles dans la zone de données :

```

1 void ETI::baseAlloc(unsigned int t)
2 {
3     m_valeur = new unsigned char[t];
4     if(m_valeur == 0)
5     {
6         QMessageBox::critical(0, "Classe ETI", "Mémoire insuffisante");
7         qApp->exit(-1); //met brutalement fin à l'exécution du programme
8         return;
9     }
10    m_taille = t;
11 }

```

La construction par copie et la fonction `nbOctetsUtils()`

La construction par copie ne se distingue de la construction par défaut que par le fait que la zone de données reçoit une copie de la zone de données du modèle (5-7). Il faut donc que l'instance en cours de construction dispose d'une zone d'une taille adaptée, et cette taille est obtenue en invoquant une [fonction prévue à cet effet](#).

Le constructeur par copie se contente d'une zone équivalente à l'[espace effectivement occupé](#) par la valeur contenue dans le modèle au moment de la copie. Il aurait aussi été possible d'allouer une zone de la même taille que celle utilisée par le modèle.

```

1 ETI::ETI(const ETI & modele): m_positif(modele.m_positif), m_taille(0),
                                m_valeur(NULL)
2 {
3     const unsigned int TAILLE = modele.nbOctetsUtils();
4     baseAlloc(TAILLE);
5     unsigned int p;
6     for(p = 0 ; p < TAILLE ; ++p)
7         m_valeur[p] = modele.m_valeur[p];
8 }

```

Il est aussi pratique de disposer d'un constructeur par "copie partielle", qui adopte la valeur absolue du modèle, mais permet de [spécifier le signe](#) de l'instance construite :

```

1 ETI::ETI(const ETI & valeur, bool signe): m_positif(signe), m_taille(0),
                                            m_valeur(NULL)
2 {
3     const unsigned int TAILLE = valeur.nbOctetsUtils();
4     baseAlloc(TAILLE);
5     unsigned int p;
6     for(p = 0 ; p < TAILLE ; ++p)
7         m_valeur[p] = valeur.m_valeur[p];
8 }

```

La fonction `nbOctetsUtils()` parcourt la zone de données, en commençant par la fin, jusqu'à ce qu'elle rencontre un octet non-nul.

Les octets nuls en fin de zone de données n'ajoutent rien à la valeur représentée, exactement comme les zéros de gauche dans le montant d'un chèque : 000010 € == 10 €.

```

1 unsigned int ETI::nbOctetsUtils() const
2 {
3   unsigned int p = m_taille-1; //m_valeur[p] est le dernier octet de la zone de données
4   while(p >= 0 && m_valeur[p] == 0)
5     --p; //on recule jusqu'au premier octet non nul
6   return p+1;
7 }

```

Transtypage à partir d'une séquence de chiffres

La construction d'une valeur numérique à partir d'une séquence de chiffres reprend la logique utilisée pour la réalisation de la calculatrice du [TD 3](#) : les chiffres sont lus de la gauche vers la droite et la prise en compte d'un nouveau chiffre a donc pour effet de multiplier la valeur actuelle par 10 avant de lui ajouter la valeur correspondant au nouveau chiffre.

```

1 ETI::ETI(const char * s)
2 {
3   m_positif = (s[0] != '-');
4   ETI laValeurAbsolue;
5   int p = (m_positif ? 0 : 1); //saute le '-' initial éventuel
6   while(s[p] >= '0' && s[p] <= '9')
7   {
8     laValeurAbsolue *= 10;
9     laValeurAbsolue += s[p] - '0'; //'7' - '0' donne 7, par exemple
10    while(s[++p] == ' '); //ignore les espaces intercalés entre les chiffres
11  }
12  m_taille = laValeurAbsolue.m_taille;
13  m_valeur = laValeurAbsolue.m_valeur;
14  laValeurAbsolue.m_valeur = NULL; //laValeurAbsolue va être détruite !
15 }

```

La valeurAbsolue est calculée dans un **ETI local** dont l'instance en construction va conserver la zone de données (13). Comme le destructeur de ETI va être exécuté au titre de cette instance locale dès la fin de l'exécution de la fonction, il faut veiller à empêcher (14) ce destructeur de libérer la zone en question (appliquer l'opérateur delete[] à un pointeur NULL est sans effet ni danger).

Transtypage à partir d'un int

Pour construire un ETI à partir d'un int, il faut **mémoriser le signe du modèle** et représenter sa valeur absolue dans une zone de donnée (6-11) d'une taille suffisante (5).

```

1 ETI::ETI(int v) : m_positif(v >= 0)
2 {
3   if(v < 0)
4     v = -v;
5   baseAlloc(sizeof(int));
6   unsigned int p=0;
7   for(p = 0 ; p < m_taille ; ++p)
8   {
9     m_valeur[p] = v % 256;
10    v /= 256;
11  }
12 }

```

Les octets qui représentent la valeur sont calculés (9-10) plutôt que lus directement dans la variable v, pour éviter de faire des hypothèses hasardeuses sur la façon dont sont représentés les int (octet de poids faible en tête ou en queue ?). Ce calcul est facilement compréhensible si l'on considère que la séquence d'octets recherchée correspond à une représentation de la valeur en base 256, les chiffres utilisés étant les 256 valeurs possibles pour un octet.

La fonction statique hasard() et la fonction redim()

Bien qu'il ne s'agisse pas d'un constructeur, hasard() est néanmoins une fonction dont l'invocation génère une valeur de type ETI, et son usage se rapprochera donc de celui d'un constructeur. Elle doit renvoyer une instance de la classe ETI de la taille spécifiée par son paramètre, et elle se procure un tel objet en redimensionnant, à l'aide d'une **fonction de service**, une **instance issue d'une construction par défaut**. La valeur "aléatoire" est obtenue en tirant au hasard chacun des octets de la zone de données (5-7), ainsi que le signe (8).

```

1 ETI ETI::hasard(unsigned long taille)
2 {
3     ETI tirage;
4     tirage.redim(taille, false);
5     unsigned int p;
6     for (p = 0 ; p < taille ; ++p)
7         tirage.m_valeur[p] = rand() % 256;
8     tirage.m_positif = rand() % 2;
9     return tirage;
10 }

```

La fonction redim() dispose de deux paramètres : le premier spécifie la nouvelle taille souhaitée, et le second indique si l'ancienne valeur doit être conservée. Lorsque c'est le cas, il est évidemment impossible d'adopter une taille inférieure au nombre d'octets nécessaires pour représenter l'ancienne valeur (3-4). Une fois qu'une zone d'une taille adéquate a été créée et initialisée correctement, l'ancienne zone de données est restituée au système (18) et la nouvelle est adoptée (19-20) par l'instance "redimensionnée" :

```

1 void ETI::redim(unsigned int nouvelleTaille, bool conserverValeur)
2 {
3     if(conserverValeur && (nbOctetsUtils() > nouvelleTaille) )
4         return;
5     //création de la nouvelle zone de données
6     unsigned char * nouvelleZone = new unsigned char[nouvelleTaille];
7     if(nouvelleZone == 0)
8     {
9         QMessageBox::critical(0, "Classe ETI", "Mémoire insuffisante");
10        qApp->exit(-1); //met brutalement fin à l'exécution du programme
11    }
12    //initialisation de la nouvelle zone de données
13    unsigned int p;
14    for(p = 0 ; p < nouvelleTaille ; ++p)
15        if (conserverValeur && p < m_taille)
16            nouvelleZone[p] = m_valeur[p]; //copie l'ancienne valeur dans la nouvelle zone
17        else
18            nouvelleZone[p] = 0;
19    //adoption de la nouvelle zone de données
20    delete[] m_valeur;
21    m_valeur = nouvelleZone;
22    m_taille = nouvelleTaille;
23 }

```

Destruction

Comme nous l'avons vu dans la [Leçon 17](#), le rôle fondamental du destructeur d'une classe encapsulant l'allocation dynamique est la restitution de la mémoire allouée :

```

1 ETI::~~ETI()
2 {
3     delete[] m_valeur;
4 }

```

5 - Affectation d'une valeur à un Entier de Taille Inhabituelle

Comme souvent, l'opérateur d'affectation ressemble beaucoup à un constructeur par copie. La seule différence est qu'il existe déjà une zone de données attribuée à l'instance qui doit prendre une nouvelle valeur. Lorsque la taille de cette zone ne suffit pas pour accueillir la valeur affectée, il faut l'ajuster (4-5). Cet exercice comporte un piège : si le même objet figure des deux côtés de l'opérateur d'affectation, la restitution de l'ancienne zone de données effectuée par `redim()` rend invalide le **pointeur** utilisé ensuite pour accéder à la "nouvelle" valeur. Dans le cas présent, l'auto-affectation ne pose pas de problème, puisqu'il est inconcevable que l'objet concerné dispose d'un **espace** insuffisant pour stocker la valeur... qu'il contient déjà !

L'auto-affectation est un phénomène bien plus fréquent qu'on pourrait le croire. Non pas parce que certains programmeurs s'amusent à écrire des chose du genre

```
uneVariable = uneVariable; //??????
```

mais parce que les objets ne sont pas toujours désignés par leur nom, et qu'il n'est pas impossible que, par exemple, deux pointeurs (ou deux références) désignent le même objet.

```
1 ETI & ETI::operator =(const ETI & autre)
2 {
3     const int TAILLE_COPIE = autre.nbOctetsUtils();
4     if(m_taille < TAILLE_COPIE)
5         redim(TAILLE_COPIE, false); //serait désastreux en cas d'auto affectation !
6     m_positif = autre.m_positif;
7     unsigned int p;
8     for (p = 0 ; p < m_taille ; ++p)
9         m_valeur[p] = (p < TAILLE_COPIE) ? autre.m_valeur[p] : 0;
10    return * this;
11 }
```

Remarquez que, du fait de la disponibilité de constructeurs de transtypage, l'opérateur d'affectation que nous venons d'écrire suffit à autoriser des choses comme :

```
1 ETI unGrandNombre; //initialisé à 0 par défaut
2 unGrandNombre = 1; //construction à partir d'int et affectation
3 unGrandNombre = "9 876 543 210"; //construction à partir de char * et affectation
```

6 - Comparaisons entre Entiers de Taille Inhabituelle

Les opérateurs de comparaison sont très facile à programmer et seront vraisemblablement indispensables à tout programme manipulant des ETI.

Les tests d'égalité et de différence

```
1 bool ETI::operator ==(const ETI & autre) const
2 {
3     const unsigned int TAILLE = nbOctetsUtils();
4     const unsigned int AUTRE_TAILLE = autre.nbOctetsUtils();
5     if(TAILLE == 0 && AUTRE_TAILLE == 0)
6         return true;
7     if(m_positif != autre.m_positif || TAILLE != AUTRE_TAILLE)
8         return false;
9     unsigned int position;
10    for (position = 0 ; position < TAILLE ; ++position)
11        if(m_valeur[position] != autre.m_valeur[position])
12            return false;
13    return true;
14 }
```

Deux valeurs sont égales si elles sont nulles (5-6), mais ne peuvent être égales si leurs signes diffèrent, si leurs représentations n'exigent pas la même quantité de mémoire (7-8) ou si ces représentations ne comportent pas des octets identiques à toutes les positions (9-13).

Deux valeurs sont différentes si... elles ne sont **pas** égales :

```

1 bool ETI::operator != (const ETI & autre) const
2 {
3     return ! (*this == autre);
4 }

```

Les tests d'infériorités et la fonction abs()

Lorsque les deux valeurs ne sont pas de même signe, leur comparaison est immédiate car elle n'implique pas les valeurs absolues (3-6). Si les deux nombres sont négatifs, ils s'ordonnent à l'inverse de leurs valeurs absolues (7-8). Le seul cas à examiner en détail est donc celui où les deux valeurs sont positives (9-23). Si les représentations sont de tailles différentes, celle qui exige le plus d'octets est évidemment celle qui correspond au nombre le plus grand (9-12). Si les deux représentations sont de la même taille, il faut les comparer en commençant (13-14) par l'octet de poids maximum (200 est supérieur à 199, par exemple) : dès qu'une **divergence** apparaît, **l'ordre des octets concernés donne l'ordre des valeurs** :

```

1 bool ETI::operator < (const ETI & autre) const
2 {
3     if(m_positif && !autre.m_positif)
4         return false; //un positif est supérieur à un négatif
5     if(! m_positif && autre.m_positif)
6         return true; //un négatif est inférieur à un positif
7     if (!m_positif && !autre.m_positif)
8         return ! (abs() < autre.abs()); //ordre inverse des valeurs absolues
9     //ils sont tous deux positifs
10    unsigned int n1 = nbOctetsUtils();
11    unsigned int n2 = autre.nbOctetsUtils();
12    if (n1 != n2)
13        return n1 < n2; //le plus court est plus petit
14    //ils occupent le même nombre d'octets en mémoire
15    unsigned char * ptr = m_valeur + n1 - 1;
16    //ptr pointe sur le dernier octet significatif de m_valeur
17    unsigned char * autrePtr = autre.m_valeur + n2 - 1;
18    //autrePtr pointe sur le dernier octet significatif de autre.m_valeur
19    while (ptr >= m_valeur)
20    {
21        if(*ptr != *autrePtr)
22            return *ptr < *autrePtr;
23        --ptr;
24        --autrePtr;
25    }
26    return false; //ils sont égaux !
27 }

```

Remarquez que le parcours des zones de données (15-21) est effectué en manipulant explicitement deux pointeurs et non à l'aide d'un index entre crochets. Comme nous feignons de croire que la taille de ces zones doit être stockée dans un **unsigned** int, nous ne pouvons en effet pas écrire :

```

unsigned int p;
for (p = n1-1 ; p >= 0 ; --p) //ERREUR : p ne peut être inférieur à 0
    if(m_valeur[p] != autre.m_valeur[p])
        return m_valeur[p] < autre.m_valeur[p];

```

Le test d'infériorité au sens large est facilement obtenu à l'aide des fonctions précédentes :

```

1 bool ETI::operator <= (const ETI & autre) const
2 {
3     if(*this == autre)
4         return true;
5     return (*this < autre);
6 }

```


Du fait de la disponibilité du constructeur par "copie partielle", la fonction `abs()` est triviale :

```
1 ETI ETI::abs() const
2 {
3     return ETI(*this, true);
4 }
```

Les tests de supériorité

Les tests de supériorité se laissent facilement réduire à des tests d'infériorité :

```
1 bool ETI::operator > (const ETI & autre) const
2 {
3     if(*this == autre)
4         return false;
5     return ! (*this < autre);
6 }
```

```
1 bool ETI::operator >= (const ETI & autre) const
2 {
3     if(*this == autre)
4         return true;
5     return ! (*this < autre);
6 }
```

7 - Additions d'Entiers de Taille Inhabituelle

Comme les comparaisons, les opérations arithmétiques élémentaires sont manifestement requises pour une classe prétendant représenter des valeurs numériques. La gestion du signe implique que certaines additions se traduisent par des soustractions.

L'addition sans effet et la fonction `ajoute()`

Une fois écartés les cas d'opérandes de signes opposés (3-6), l'addition revient à faire la somme des valeurs absolues. Comme le résultat doit figurer dans une nouvelle instance de ETI, celle-ci est créée par copie de l'opérande de droite, puis chaque **octet de l'opérande de gauche** est ajouté (en propageant la retenue éventuelle) à l'octet qui figure à la **même position** dans la nouvelle instance.

```
1 ETI ETI::operator + (const ETI &autre) const
2 {
3     if(m_positif && ! autre.m_positif)
4         return *this - autre.abs(); // 1 + -2 == 1 - 2
5     if(!m_positif && autre.m_positif)
6         return autre - abs(); //-1 + 2 == 2 - 1
7     //les valeurs sont de signes identiques, le résultat aura le même : -1 + -2 == -(1+2)
8     ETI resultat (autre); //on va faire la somme des valeurs absolues
9     unsigned int p;
10    for (p = 0 ; p < m_taille ; ++p)
11        resultat.ajoute(m_valeur[p], p);
12    return resultat;
13 }
```

L'ajout de la **valeur d'un octet** à une certaine **position** dans la représentation du résultat (avec propagation de la retenue si le résultat dépasse 255) est confié à une **fonction de service** :

```
1 void ETI::ajoute(int quantite, int position)
2 {
3     assert(quantite >= 0 && quantite < 256);
4     if(m_taille < position+1) //on travaille sur m_valeur[p]
5         redim(position+1, true); //"agrandit" la zone en conservant la valeur
6     while (quantite > 0 && position < m_taille)
7     {
8         quantite += m_valeur[position]; //ce qu'on doit représenter
```

```

9   m_valeur[position++] = quantite % 256; //on représente tout ce qu'on peut
10  quantite /= 256;                      //on va mettre le reste plus loin
11  }
12  if(quantite > 0)
13  { //il en reste encore, mais on n'a plus de place car position == m_taille
14    redim(m_taille+1, true);             //augmente m_taille
15    m_valeur[position] = 1; //propage la retenue dans le nouvel octet
16  }
17  }

```

Les additions à effet

```

1 void ETI::operator += (const ETI & autre) //l'addition d'un opérande avec effet
2 { *this = *this + autre; }

```

```

1 ETI & ETI::operator ++ () //incrémentatation préfixé
2 {
3   if(m_positif)
4     ajoute(1,0);
5   else
6     enleve(1,0);
7   return *this;
8 }

```

Lorsque la valeur est négative, l'incrémenter revient à enlever 1 à sa valeur absolue, ce qui peut être effectué en appelant la [fonction de service](#) qui joue, pour la soustraction, un rôle analogue à celui rempli par `ajoute()` dans le cas de l'addition.

Comme bien souvent, l'incrémentatation postfixée fait appel à l'opérateur préfixé :

```

1 ETI ETI::operator ++ (int) //incrémentatation postfixé
2 {
3   ETI avant(*this);
4   ++*this ;
5   return avant;
6 }

```

8 - Soustractions d'Entiers de Taille Inhabituelle

La gestion du signe implique que certaines soustractions se traduisent par des additions.

La soustraction sans effet et la fonction `enleve()`

```

1 ETI ETI::operator - (const ETI &autre) const
2 {
3   if(m_positif && ! autre.m_positif)
4     return *this + autre.abs();           // 1 - -2 == 1 + 2
5   if(!m_positif && autre.m_positif)
6     return ETI (abs() + autre, false);    // -1 - 2 = -(1+2)
7   if(!m_positif && !autre.m_positif)
8     return ETI (autre.abs() - abs(), false); // -1 - -2 = -1 + 2 == 2 - 1
9   //les deux valeurs sont positives
10  if (*this < autre)
11    return ETI (autre - *this, false);    // 1 - 2 = -(2-1)
12  ETI resultat (*this);
13  unsigned int p;
14  for (p=0 ; p < autre.nbOctetsUtils() ; ++p)
15    resultat.enleve(autre.m_valeur[p], p);
16  return resultat;

```

La soustraction de la [valeur d'un octet](#) à une certaine [position](#) dans la représentation du résultat (avec propagation d'une retenue éventuelle) est confié à une [fonction de service](#) :

```

1 void ETI::enleve(int quantite, int position)
2 {
3     assert(quantite >= 0 && quantite < 256);
4     assert(m_taille >= position);
5     while (quantite > 0)
6         if (m_valeur[position] >= quantite)
7             { //il n'y aura pas de retenue
8                 m_valeur[position] -= quantite;
9                 quantite=0; //donc c'est fini
10            }
11        else
12            { //on "emprunte" 256 à l'octet suivant
13                m_valeur[position++] += 256 - quantite;
14                quantite = 1; //propage la retenue
15            }
16 }

```

Les soustractions à effet

```

1 void ETI::operator -= (ETI autre) //la soustraction d'un opérande avec effet
2 {
3     *this = *this - autre;
4 }

```

```

1 ETI & ETI::operator -- () //la décrémentation préfixée
2 {
3     if(m_positif)
4         enleve(1,0);
5     else
6         ajoute(1,0); //décrémenter un nombre négatif augmente sa valeur absolue
7     return *this;
8 }

```

```

1 ETI ETI::operator -- (int) //la décrémentation postfixée
2 {
3     ETI avant(*this);
4     --*this ;
5     return avant;
6 }

```

9 - Multiplications d'Entiers de Taille Inhabituelle

La multiplication est légèrement plus complexe que l'addition et la soustraction.

La multiplication sans effet et la fonction foisDeux()

Bien que les valeurs soient représentées en binaire, la procédure habituelle de multiplication peut être employée quasiment telle quelle. Illustrons, par exemple, la multiplication de 25 par 13 :

	1 1001	autre (25, en binaire)
*	1101	*this (13, en binaire)
<hr/>		
	1 1001	
	11 0010	Valeurs successives aAjouter
	110 0100	
	1100 1000	
<hr/>		
	1 0100 0101	resultat (325, en binaire)

Comme les chiffres du multiplicateur (*this) ne peuvent être que des 0 et des 1 (on est en binaire...), les valeurs successives aAjouter peuvent être obtenues par doublement (multiplier par deux revient à ajouter un 0 à droite, comme multiplier par 10 en base 10). Bien entendu, la

somme qui donne le resultat ne doit pas inclure les valeurs de aAjouter qui correspondent à un bit nul dans le multiplicateur.

Comme nous disposons déjà des opérateurs d'addition, il ne nous manque qu'un moyen de vérifier la non-nullité de chacun des bits du multiplicateur et une fonction doublant la valeur de l'instance au titre de laquelle elle est appelée. Pour isoler la valeur d'un bit particulier d'un octet, on fait appel à un opérateur (cf. Leçon 24) qui effectue un ET logique bit à bit : chaque bit du résultat sera nul, sauf si les bits correspondants des opérandes sont tous deux non-nuls. Il suffit donc que le second opérande ait un seul bit non-nul (à la position qui nous intéresse) pour que la non nullité du résultat indique que le premier opérande comporte, lui aussi, un bit non nul à cette position.

```

1 ETI ETI::operator * (const ETI &autre) const
2 {
3     const unsigned char ISOLE[8] = {1, 2, 4, 8, 16, 32, 64, 128};
4     ETI aAjouter = autre;
5     ETI resultat;
6     unsigned int p;
7     int bit;
8     for (p = 0 ; p < nbOctetsUtils() ; ++p)
9         for (bit = 0 ; bit < 8 ; ++bit)
10            {
11                if(m_valeur[p] & ISOLE[bit])
12                    resultat += aAjouter;
13                aAjouter.foisDeux();
14            }
15     resultat.m_positif = (m_positif == autre.m_positif);
16     return resultat;
17 }

```

Doubler la valeur d'un ETI revient à doubler (8) chacun des octets qui représentent sa valeur (5-12), en veillant à propager les retenues éventuelles (7, 9-11, 13-18) :

```

1 void ETI::foisDeux()
2 {
3     int p;
4     bool ilYaUneRetenue = false;
5     for(p = 0 ; p < nbOctetsUtils() ; ++p)
6         {
7             bool debordement = (m_valeur[p] >= 128); //ie. si le bit de gauche vaut 1
8             m_valeur[p] *= 2; //ajoute un zéro à droite (le bit de gauche est perdu)
9             if(ilYaUneRetenue)
10                 ++m_valeur[p]; //le bit de droite passe de 0 à 1
11             ilYaUneRetenue = debordement;
12         }
13     if(ilYaUneRetenue)
14         { //le doublement du dernier octet a généré une retenue
15             if (p == m_taille) //tous les octets disponibles étaient utilisés
16                 redim(p+1, true);
17             m_valeur[p] = 1;
18         }
19 }

```

La multiplication à effet

```

1 void ETI::operator *= (ETI autre)
2 {
3     *this = *this * autre;
4 }

```

10 - Divisions d'Entiers de Taille Inhabituelle

La division entière produit *deux* résultats : le quotient et le reste. Le quotient est renvoyé par l'opérateur /, alors que le reste est renvoyé par l'opérateur %, mais l'opération sous-jacente est la même. Il semble donc logique d'implémenter ces opérateurs en faisant appel à une seule et unique fonction de division, qui renvoie le quotient et utilise un premier paramètre pour spécifier le **diviseur** et un second pour **stocker le reste**.

Les divisions sans et avec effet et l'opérateur %

L'opérateur de division ne s'intéresse pas au reste mais doit, pour satisfaire aux exigences syntaxiques de la véritable fonction de division, disposer d'une variable capable de le stocker :

```
1 ETI ETI::operator / (const ETI &autre) const
2 {
3     ETI reste;
4     return divide(autre, reste);
5 }
```

La division avec effet utilise simplement la division sans effet et l'affectation :

```
1 void ETI::operator /= (const ETI &autre)
2 {
3     *this = *this / autre;
4 }
```

L'opérateur modulo néglige la valeur renvoyée par `divide()` et renvoie la valeur placée par cette fonction dans la variable `reste` :

```
1 ETI ETI::operator % (const ETI &autre) const
2 {
3     ETI reste;
4     divide(autre, reste);
5     return reste;
6 }
```

La fonction `divide()`

La division est confrontée à un problème auquel échappent les autres opérations arithmétiques : elle n'est pas toujours définie. La conduite à tenir en cas de tentative de division par zéro est une question délicate sur laquelle nous ne nous étendrons pas trop ici (en particulier parce que la gestion des exceptions ne sera abordée qu'au cours de la Leçon 23). La fonction `divide()` se contente de calculer l'inverse du nombre d'octets nécessaires pour représenter le diviseur (3). Lorsque le diviseur est nul, la fonction `nbOctetsUtils()` renvoie 0, ce qui déclenche le mécanisme prévu pour gérer la division par 0 dans le cas des `int`. Plutôt que de prévoir sa propre procédure, la classe `ETI` emprunte donc aveuglément aux `int` leur gestion de la division par zéro.

Les compilateurs actuels procèdent à une analyse assez poussée du texte source, dans le but d'optimiser le code qu'ils génèrent. Une variable qui n'est pas réellement utilisée risque à cette occasion de disparaître purement et simplement, avec les calculs qui auraient été nécessaires pour l'initialiser. Spécifier qu'une variable est **volatile** permet d'indiquer au compilateur qu'il doit se garder de toute tentative d'optimisation la concernant.

Une fois écartée l'hypothèse de la division par 0, le problème est de savoir combien de fois on peut soustraire le diviseur du dividende (ce qui nous donnera le quotient) et combien il restera après la dernière soustraction (ce qu'on appelle, justement, le reste).

Abordée ainsi, la division est facile à programmer, mais très lente du fait du nombre souvent considérable de soustractions requises. La version que nous allons utiliser limite le nombre de soustractions en utilisant une opération que nous savons effectuer relativement rapidement : le doublement d'une valeur. Plutôt que de soustraire le diviseur, elle soustrait (21) la plus grande valeur possible obtenue par doublements successifs (12-19) de ce diviseur (le nombre à ajouter au quotient (22) pour refléter cette soustraction n'est donc pas 1, mais 1 doublé (18) autant de fois

que l'a été le diviseur). Le reste obtenu après cette soustraction est en général plus grand que le diviseur, et il faut lui ré-appliquer l'opération autant de fois que nécessaire (7-23).

```

1 ETI ETI::divise(const ETI &autre, ETI &reste) const
2 {
3     //run-time error délibérée en cas de division par 0 :
4     volatile int x = 1/autre.nbOctetsUtils();
5     reste = abs();
6     const ETI diviseur(autre.abs());
7     ETI quotient;
8     while(reste >= diviseur)
9     {
10        ETI petite(diviseur);
11        ETI grande = petite;
12        ETI ajout(1);
13        while (grande <= reste)
14        {
15            grande.foisDeux();
16            if(grande <= reste)
17            {
18                petite = grande;
19                ajout.foisDeux();
20            }
21        }
22        reste -= petite;
23        quotient += ajout;
24    }
25    return ETI(quotient, m_positif == autre.m_positif);
26 }

```

11 - Représentation textuelle d'un Entier de Taille Inhabituelle

La présentation des valeurs de type ETI sous une forme intelligible pour un être humain exige la création d'une chaîne de caractères. Cette création exige une succession de divisions par 10 qui peut conduire à des temps d'exécution impressionnants...


```

1 QString ETI::texte() const
2 {
3     const ETI DIX(10);
4     QString chaine;
5     ETI val(*this);
6     ETI reste;
7     while (val.nbOctetsUtils() != 0)
8     {
9         val = val.divise(DIX, reste);
10        QChar chiffre = reste.m_valeur[0] + '0';
11        chaine = chiffre + chaine;
12    }
13    return m_positif ? chaine : "-" + chaine;
14 }

```

Cette fonction est le seul membre de la classe ETI qui dépende de la librairie Qt. Pour exploiter les ETI en l'absence de Qt, il faudra donc modifier le type de la fonction `texte()` et celui de ses variables `chaine` et `chiffre`.

12 - Le fichier eti.h

Une fois définies toutes les fonctions membre que nous venons de décrire , votre fichier eti.h devrait avoir l'aspect suivant :

```

1 #include "qstring.h"
2 class ETI
3 {
4 public:
5     //construction et Cie
6     ETI();
7     ETI(const ETI & modele);
8     ETI(const ETI & valeur, bool signe);
9     ETI(const char * s);
10    ETI(int v);
11    ~ETI();
12    static ETI hasard(unsigned long t);
13    //affectation
14    ETI & operator =(const ETI & autre) ;
15    //comparaisons et valeur absolue
16    bool operator ==(const ETI & autre) const;
17    bool operator !=(const ETI & autre) const;
18    bool operator <(const ETI & autre) const;
19    bool operator <=(const ETI & autre) const;
20    ETI abs() const;
21    bool operator >(const ETI & autre) const;
22    bool operator >=(const ETI & autre) const;
23    //additions
24    ETI operator +(const ETI &autre) const;
25    void operator += (const ETI &autre);
26    ETI & operator ++ (); //préfixé
27    ETI operator ++ (int); //postfixé
28    //soustractions
29    ETI operator -(const ETI &autre) const;
30    void operator -= (ETI &autre);
31    ETI & operator -- (); //préfixé
32    ETI operator -- (int); //postfixé
33    //multiplications
34    ETI operator *(const ETI &autre) const;
35    void operator *= (ETI &autre);
36    //divisions
37    ETI operator /(const ETI &autre) const;
38    ETI operator %(const ETI &autre) const;
39    void operator /= (const ETI &autre);
40    //représentation textuelle
41    QString texte() const;
42 protected:
43    unsigned int m_taille;
44    unsigned char * m_valeur;
45    bool m_positif;
46    //fonctions de service
47    void baseAlloc(unsigned int t=1);
48    unsigned int ETI::nbOctetsUtils() const;
49    void redim(unsigned int nouvelleTaille, bool conserverValeur);
50    void ajoute(int quantite, int position);
51    void enleve(int quantite, int position);
52    void foisDeux();
53    ETI divise(const ETI &autre, ETI & reste) const;
54 };

```