Traitement HTML avec C++ Builder 6

Par Gilles Louise

Date de publication : 1 octobre 2002

Ce tutoriel se propose d'ajouter ou de modifier avec C++ Builder 6 un répertoire d'accès aux éléments dans un fichier HTML pour une extension donnée

Introduction

Ce tutoriel se propose d'ajouter ou de modifier avec C++ Builder 6 un répertoire d'accès aux éléments dans un fichier HTML pour une extension donnée. Cette fonction peut être utile à ceux qui restructurent leur site et veulent mettre dans des répertoires particuliers des fichiers ayant la même extension par exemple des images, des sons, des textes. Cet utilitaire m'a été suggéré par un webmestre qui a eu cette difficulté.

1 - Position du problème

Imaginons un site donnant accès à des fichiers MIDI (Musical Instrument Digital Interface). Que ce soient des fichiers MIDI n'a aucune importance, ce qui importe est qu'il s'agisse de liens utilisant le mot clé HREF du HTML. Dans ces conditions, pour faire jouer un fichier, on aura dans le HTML une syntaxe de ce type :

```
<a href="Titre.mid">Titre</a>
```

Le mot « Titre » apparaît à l'écran et en cas de clic sur ce lien, le fichier MIDI Titre.mid sera joué par le MediaPlayer. Comme aucun chemin n'est indiqué dans l'exemple précédent, le fichier Titre.mid est censé se trouver dans le répertoire par défaut. Si maintenant l'utilisateur veut créer un répertoire nommé par exemple MIDI et y transférer tous ses fichiers MIDI, il devra modifier dans le HTML tous les liens c'est-à-dire rajouter la chaîne MIDI/ juste avant le nom du fichier. Ainsi, l'exemple précédent deviendra

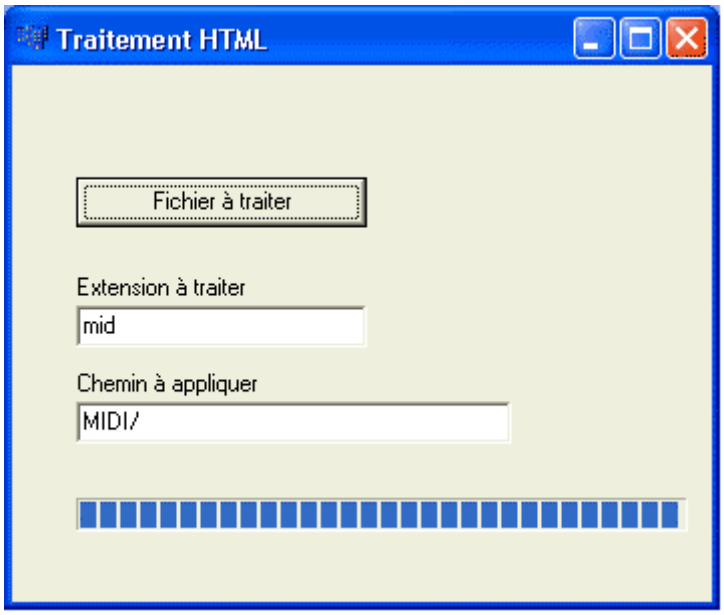
```
<a href="MIDI/Titre.mid">Titre</a>
```

La chaîne MIDI/ a été ajoutée avant le nom du fichier, Titre.mid est donc censé se trouver maintenant dans ce répertoire. Il est vrai que si le webmestre n'a que quelques liens à modifier sur sa page, il peut le faire à la main. Mais s'il a plusieurs centaines de liens voire des milliers, il serait très fastidieux de procéder manuellement et même risqué car la saisie manuelle est beaucoup moins fiable que l'automatisme informatique.
Il s'agit donc de programmer une écriture conditionnelle. Il faut repérer dans le fichier HTML la chaîne « .mid » puis reculer jusqu'à repérer le HREF correspondant au lien, extraire de ce segment le nom pur du MIDI, ajouter le chemin à ce nom et réécrire cette correction dans le fichier.

2 - Analyse

On initialise le programme au moment du constructeur principale de la fiche.

Notre application se compose d'une fenêtre avec cinq éléments, un bouton pour choisir le fichier à traiter via un TOpenDialog, deux LabeledEdit (nouveau composant propre à C++ Builder 6), un qui contiendra l'extension à rechercher (ici par défaut mid) et l'autre le chemin à donner aux liens trouvés (ici par défaut MIDI/) et enfin une barre de progression



Dans ces conditions, le programme va rechercher tous les « .mid » (contenu du LabeledEdit Extension) et appliquer au nom pur du MIDI le chemin donné à savoir MIDI/ (contenu du LabeledEdit Chemin). Voici les noms donnés aux composants (propriété Name) :

- La forme principale : **Traitement** ;
- Le TOpenDialog : **OuvreFichier** ;
- Le bouton : **Recherche** ;
- Le premier LabeledEdit : **Extension** ;
- Le deuxième LabeledEdit : **Chemin** ;
- La barre de progression : **Progression**.

La chaîne HREF a été déclarée comme constante juste après le pointeur de fenêtre principale

```
TTraitement *Traitement;
const AnsiString Borne="HREF";
```

On aurait pu utiliser aussi un **TLabeledEdit** pour ce champ mais comme c'est un mot clé standard du HTML, nous avons opté pour une constante.

Dans la classe principale **TTraitement**, nous avons déclaré dans la zone réservée à l'utilisateur quelques variables :

```
TStringList *Texte;
AnsiString EXT;
int NbModif;
bool MODIF;
int liml, limc;
int finl,finc;
```

- un pointeur de **TStringList** qui va nous servir à lire le fichier HTML ;
- un **AnsiString** qui contiendra l'extension à rechercher ;
- un entier qui va contenir par incrémentation le nombre de modifications opérées par le programme ;
- une variable booléenne qui nous dira pour une position donnée s'il y a eu ou non une modification dans le fichier ;
- deux entiers représentant en cas de modification la position à partir de laquelle on reprend l'analyse, c'est donc aussi la limite au-delà de laquelle il ne sera pas possible de reculer.

En effet, quand une extension sera trouvée, on recherchera le HREF correspondant mais on ne pourra pas reculer au-delà de la dernière intervention. Cette précaution est nécessaire car si l'extension trouvée ne correspond à aucun

HREF, on risquerait de traiter le HREF précédent, ce qui serait fautif. Et enfin deux autres entiers représentant la limite supérieure au-delà de laquelle il faudra stopper la recherche.

En effet, quand l'extension sera trouvée, on exigera qu'il y ait une fermeture de chaîne (c'est-à-dire le caractère « " »).

On cherchera donc le signe « " » en tolérant seulement la possibilité d'espaces avant ce signe. Ensuite on reculera jusqu'à trouver le HREF, on exigera qu'il y ait après HREF le signe « = » (avec tolérance d'espaces selon la syntaxe du HTML), on exigera ensuite un « " » mais la recherche de ces signes ne pourra pas se faire au-delà de la limite (finl,finc) et ce, pour ne pas fausser l'analyse dans le cas où le fichier HTML serait erroné avec une chaîne qui par exemple n'aurait pas le signe « " » ouvrant mais aurait le signe « " » fermant. Ainsi, quand on traitera l'occurrence, on sera certain

1 qu'il y a l'extension ;
2 qu'après l'extension il y a le signe fermant « " » (position mémorisée dans finl et finc) ;
3 qu'avant l'extension il y a HREF strictement avant la limite (liml,limc) qui représente la position de la dernière modification ;
4 q'après HREF il y a le signe « = » strictement avant la limite (finl,finc) ;
5 qu'après le signe « = » il y a le signe ouvrant « " » strictement avant la limite (finl,finc).

Dans ces conditions, l'occurrence est valide car on a dans l'ordre HREF, le signe « = », le signe ouvrant « " », l'extension, le signe fermant « " ». L'occurrence étant valide, on la traite en extrayant le nom pur du fichier (c'est-à-dire sans le chemin s'il existe) avec son extension, en ajoutant le chemin donné et en écrivant ce résultat à la place de ce qui se trouve entre les deux signes « " ».

2.1 - On initialise le programme au moment du constructeur principale de la fiche

```
__fastcall TTraitement::TTraitement(TComponent* Owner)
: TForm(Owner)
    Extension->Text="mid";
    Chemin->Text="MIDI/";
    Texte=new TStringList;
```

On propose une valeur par défaut aux deux LabeledEdit,

- le premier avec la chaîne « mid » (Extension->Text="mid";)
- le second avec « MIDI/ » (Chemin->Text="MIDI/";), l'utilisateur étant libre de modifier ces paramètres avant traitement.

On instancie la TStringList (Texte=new TStringList;).

2.2 - Voici ce qui se passe au moment du clic du bouton

```
void __fastcall TTraitement::RechercheClick(TObject *Sender)
    Progression->Position=0;
    if(OuvreFichier->Execute())
        TraiteFic(OuvreFichier->FileName);
```

On initialise la position de la barre de progression à zéro (Progression->Position=0;).
On ouvre le **TOpenDialog** en testant sa valeur de retour (if(OuvreFichier->Execute())).
Si l'utilisateur a choisi un fichier, on le traite (TraiteFic(OuvreFichier->FileName);).
Si l'utilisateur a annulé la recherche de fichier, il ne se passe rien.

2.3 - TraiteFic reçoit en argument le nom du fichier, il se contente de demander confirmation à l'utilisateur

```
void TTraitement::TraiteFic(AnsiString N)
    if(Confirme(N))
        TraiteFic1(N);
```

On appelle la fonction booléenne Confirme en testant sa valeur de retour (if(Confirme(N))), si la valeur est true, l'utilisateur confirme alors sa demande et donc on traite vraiment le fichier (TraiteFic1(N);).

2.4 - La fonction booléenne de confirmation est très simple

```
bool TTraitement::Confirme(AnsiString N)
    N="Nous allons traiter le fichier "+N;
    return (Application->MessageBox(N.c_str(), "Traitement", MB_OKCANCEL)==IDOK);
```

N étant le nom du fichier, on prépare dans N le message final (N="Nous allons traiter le fichier "+N;)
puis on retourne à l'appelant le résultat du **MessageBox**.

```
return ( Application-> MessageBox ( N.c_str(), "Traitement" , MB_OKCANCEL ) == IDOK);
```

Ce **MessageBox** contient un bouton **OK** et un bouton **Annuler**. Si le bouton **OK** est cliqué, la valeur de retour sera true et le traitement se fera, sinon le retour se fait avec false parce que l'utilisateur a cliqué **Annuler**.

Remarquez ce type de syntaxe très pratique qui consiste à utiliser return en relation avec une fonction avec une syntaxe du type return(fonction()). La fonction est ainsi appelée et ce que renvoie la fonction (ici une valeur booléenne) sera à son tour renvoyé à l'appelant.

2.5 - TraiteFic1 devient le programme maître en tant que tel, il reçoit en argument le nom du fichier et procède à ses lecture, modification et écriture sur disque dur

```
void TTraitement::TraiteFic1(AnsiString N)
    int i=0,j=1;
    Update();
    liml=0;
    limc=1;
    NbModif=0;
    EXT="."+UpperCase(Extension->Text);
    Texte->LoadFromFile(N);
    do
        TraitePos(i,j);
    while(CarSuiv(i,j));
    Application->MessageBox(CompteRendu(NbModif).c_str(),"Traitement", MB_OK);
    Texte->SaveToFile(OuvreFichier->FileName);
```

On initialise deux entiers i et j, le premier à 0 et le deuxième à 1 (int i=0,j=1;). En effet, i sera le compteur de lignes et j le compteur de caractères sur la ligne i. Comme **Texte** est notre **TStringList**, Texte->Strings[i] sera un **AnsiString** correspondant à la ligne d'indice i mais cet indice commence à 0 et donc va naviguer de 0 à Texte->Count-1. En revanche, l'indice des **AnsiString** à commence à 1, si A est un **AnsiString**, A[1] sera le premier caractère de l'**AnsiString** et A.Length() sa longueur. Quelles que soient à les valeurs de i et de j, le caractère pointé est Texte->Strings[i][j]. On peut donc considérer le couple (i,j) comme un à pseudo-pointeur qui pointe le caractère numéro j de la ligne d'indice i via la syntaxe d'accès Texte->Strings[i][j].

On redessine la forme principale (Update();). La raison en est qu'une boîte de dialogue a été affichée juste avant pour choisir le fichier et que cette fenêtre a toute chance d'avoir chevauché notre forme principale. Si on ne redessine pas la fenêtre principale maintenant, elle ne se redessinera pas correctement suite à la disparition de la boîte de dialogue, elle ne se redessinera qu'à la fin du traitement au moment où Windows peut prendre la main. Comme le

traitement peut être assez long si le fichier est grand, cette petite précaution améliore la présentation, la fenêtre est immédiatement redessinée après la disparition de l'**OpenDialog**.

On initialise la limite inférieure au-delà de laquelle on ne pourra pas reculer dans la recherche du HREF soit la position (0,1), premier caractère de la première ligne (liml=0;limc=1;). On initialise le nombre de modifications à 0 (NbModif=0;). Puis on lit dans l'AnsiString EXT ce que contient le **LabeledEdit Extension** en convertissant le tout en majuscules et en y ajoutant le point (EXT="."+UpperCase(Extension->Text);).

On a utilisé la fonction **UpperCase** pour convertir L'AnsiString en majuscules. La conversion en majuscules est nécessaire sinon il faudrait à chaque fois tester les minuscules et les majuscules. Il en sera de même au moment de la lecture des données, on convertira systématiquement les caractères lus en majuscules, les comparaisons ne se feront qu'en majuscules. EXT contient donc la chaîne que nous cherchons c'est-à-dire un point et l'extension en majuscules.

On lit maintenant le fichier choisi dans la **TStringList** (Texte->LoadFromFile(N);). Vous voyez qu'il n'y a rien de plus simple que de lire un fichier texte avec C++ Builder, une seule instruction suffit, il faut simplement donner le nom du fichier.

La boucle fondamentale du programme se trouve là. À partir de la position de départ contenue dans le couple (i,j), on traite cette position et on avance jusqu'à la fin du fichier (do TraitePos(i,j); while(CarSuiv(i,j));). **CarSuiv** est une fonction booléenne qui d'une part avance le pseudo-pointeur (i,j) d'un caractère et d'autre part dit si ce caractère existe ou non. Donc tant qu'il y a un caractère suivant, on traite la position (i,j) pointée.

À la fin de cette boucle magique qui est en fait le cœur du programme, on affiche un compte-rendu

```
Application->MessageBox( CompteRendu ( NbModif ).c_str() , "Traitement", MB_OK);
```

Le message est créé par la fonction CompteRendu qui renvoie un AnsiString, on applique donc à cet AnsiString la méthode .c_str() (conversion string) pour le convertir en char* car la fonction **MessageBox** ne fonctionne qu'avec un char*.

Pour terminer, on sauvegarde la TStringList sur disque dur (Texte->SaveToFile(OuvreFichier->FileName);), ce qui valide toutes les modifications qui y ont été apportées durant le traitement.

2.6 - Voici comment nous créons le message de compte-rendu

```
AnsiString __fastcall TTraitement::CompteRendu(int N)
    AnsiString M1="Traitement terminé! ";
    AnsiString M2="Aucune intervention à signaler";
    AnsiString M3="Il y a eu ";
    AnsiString M4="une";
    AnsiString M5=" intervention";
    AnsiString M6=IntToStr(N);
    AnsiString M7="s";
    if(N==0)
        return(M1+M2);

    if(N==1)
        return(M1+M3+M4+M5);

    return(M1+M3+M6+M5+M7);
}
```

On distingue trois cas, celui où il n'y a eu aucune modification ce qui signifie qu'une extension valide n'a jamais été trouvée dans le fichier HTML, celui où il n'y en a eu une seule et celui où il y en a eu plusieurs. Par concaténation de fragment de phrases, on obtient le bon message à renvoyer. Cela nous évite d'avoir à utiliser des affichages du genre « intervention(s) » qui laisse à penser qu'un ordinateur ne sait pas distinguer le singulier du pluriel.

2.7 - Voici comment nous allons traiter la position contenue dans le couple (i,j).

```
void __fastcall TTraitement::TraitePos(int& l,int& c)
    MODIF=false;
    if(Trouve(l,c,EXT))
        TraiteOcc(l,c);

    if(MODIF)
       l=liml;
       c=limc;
```

La fonction **TraitePos** reçoit deux arguments l et c, ce qui fait que i devient l (ligne) et j devient c (numéro de caractère). Ces arguments sont du type référence (opérateur &) et seront donc renvoyés à l'appelant. On commence par initialiser la variable MODIF à false (MODIF=false;). Si l'extension EXT est trouvée à partir de la position (l,c), on traite l'occurrence (if(Trouve(l,c,EXT)) TraiteOcc(l,c);). Si l'occurrence a été traitée, il se peut qu'il y ait eu modification du fichier, ce n'est pas sûr car il y a d'autres conditions à tester.

Quoi qu'il en soit, s'il y a eu modification (if(MODIF)) on renverra à l'appelant la position à partir de laquelle l'analyse doit se poursuivre soit le couple (liml,limc) (l=liml; c=limc;). Cette position correspond à la dernière lettre copiée. Par exemple si l'occurrence trouvée est href="Titre.mid", après correction on aura href="MIDI/Titre.mid", (liml,limc) pointera le d de mid, c'est à partir de là que l'analyse reprendra. Si aucune modification n'a été opérée, alors les arguments l et c ne sont pas touchés et sont donc renvoyés tels quels à l'appelant.

2.8 - Voici la fonction booléenne Trouve qui va nous dire si oui ou non on lit la chaîne EXT à partir de la position donnée (l,c).

```
bool TTraitement::Trouve(int l,int c,AnsiString Ext)
    int i=1;
    bool ok,rep;
    AnsiString Car;
    do
       Car=UpperCase(Texte->Strings[l][c]);
        rep=(Car==Ext[i]);
        if(rep)
            if(i++ == Ext.Length())
                ok=false;
            else
                ok=CarSuiv(l,c);
        else
            ok=false;
    while(ok);
    if(rep)
        rep=TrouveSigne(l,c,'"',false);

    if(rep)
        finl=l;
        finc=c;

return rep;
```

Il s'agit de savoir si la suite de caractères contenu dans l'AnsiString EXT correspond à la suite de caractères à partir de la position pointée par le couple (l,c). Comme EXT est un AnsiString et que l'indice des AnsiString commence à 1, on commence par créer une variable i en l'initialisant à 1 (int i=1;). L'entier i va donc être une sorte de pointeur de la chaîne EXT, on peut dire qu'il pointe le caractère numéro i de EXT soit EXT[i].
On crée ensuite deux variables booléennes, une première qui va contrôler la boucle en while et une deuxième qui contiendra la réponse au sortir de la boucle (bool ok,rep;).

On crée L'AnsiString **Car** pour lire caractère après caractère la **TStringList**. On rentre dans la boucle en do et on lit le caractère pointé par le couple (l,c) en le convertissant en majuscule (Car=UpperCase(Texte->Strings[l][c]);). Puis on renseigne la variable booléenne **rep** en fonction du fait que **Car** est ou non égal au caractère de EXT pointé par

i (rep=(Car==Ext[i]);). Si **rep** est vrai, les deux caractères sont égaux sinon ils sont différents. Si **rep** est vrai (if(rep)) on regarde si i est égal à la longueur de l'extension cherchée EXT.

On en profite pour incrémenter i mais comme il s'agit d'une post-incrémentation, la comparaison s'effectue avec la valeur de i non encore incrémenté (if(i++==Ext.Length())). Si c'est vrai, on met ok à false (ok=false;) ce qui aura pour effet de stopper la boucle en while et comme **rep** est vrai, le programme conclura qu'on a trouvé l'extension, ce qui est vrai puisque les Ext.Length() comparaisons successives ont toutes été positives. Si i n'est pas égal à la longueur de l'extension EXT, alors on se positionne au caractère suivant et on positionne ok en fonction de son existence ou non (ok=CarSuiv(l,c);). Ici, si ok est vrai, il y a un caractère suivant sinon on est arrivé à la fin de la TStringList. Si ok est vrai, on est prêt pour la comparaison suivante avec i incrémenté et le couple (l,c) pointant lui aussi le caractère suivant.

Si **rep** est faux, on se contente de mettre ok à false (else ok=false;), ce qui va arrêter la boucle en while et comme **rep** est à false, l'appelant conclura que l'extension n'a pas été trouvée. On teste maintenant la variable ok qui dira s'il faut ou non continuer la boucle (while(ok);). Si l'extension a été trouvée (if(rep)), on cherche à savoir si le prochain caractère autre qu'un espace est une fermeture de chaîne de caractères « " » et on met ce résultat dans **rep** (rep=TrouveSigne(l,c,"",false);). Ici on cherche le signe « " » sans limite autre que la fin du fichier, ce pourquoi le dernier argument booléen est à false (sans limite). Si **rep** est de nouveau vrai, on a trouvé l'extension et le signe « " » fermant, on mémorise alors dans finl et finc la position de ce signe fermant (finl=l; finc=c;) et ce, parce que quand on cherchera le signe « = » et le signe « " » ouvrant après le HREF, ils devront se situer strictement avant cette limite (finl,finc).

2.9 - La procédure CarSuiv pointe le caractère suivant et dit s'il existe.

```
bool __fastcall TTraitement::CarSuiv(int& l,int& c)
bool existe,vide;
if(c++ < Texte->Strings[l].Length())
    return true;

c=1;
do
    existe=(++l!=Texte->Count);
    if(existe)
        vide=!Texte->Strings[l].Length();
while(existe && vide);
return existe;
```

Comme cette fonction va renvoyer la nouvelle position du couple (l,c), ces deux entiers sont envoyés en référence avec l'opérateur & qui permet de communiquer avec l'appelant, ce signe signifiant simplement que l et c sont comme renvoyés à l'appelant. En réalité il ne sont pas à proprement parler renvoyés mais comme on procède par adresse grâce au signe « & », cela revient à dire que leurs valeurs sont renvoyées à l'appelant.

On commence par créer deux variables booléennes, (bool existe,vide;). La variable existe dira si oui ou non il y a un caractère suivant et la variable vide sera utilisée pour savoir si la ligne suivante est vide ou non. En effet, si c pointe le dernier caractère de la ligne l à savoir le caractère numéro Texte->Strings[l].Length(), il faut alors incrémenter l pour pointer la ligne suivante. Si cette ligne existe, il faut qu'elle soit non nulle et passer encore à la suivante en cas de nullité car si la ligne l est vide, la lecture d'un caractère par l'instruction Texte->Strings[l][c] donnera un **Out_Of_Range** (**ERangeError**). En lisant une **TStringList** caractère après caractère par une syntaxe du type Texte->Strings[l][c], il faut toujours que l soit compris entre 0 et Text->Count-1 et c entre 1 et Texte->Strings[l].Length() sinon il y aura un **OutOfRange**.

On commence par regarder si c est inférieur au nombre de caractères de la ligne en cours (if(c++ < Texte->Strings[l].Length())). On en profite pour incrémenter c mais comme il s'agit d'une post-incrémentation, on teste bien la valeur actuelle de c avant incrémentation. Si c'est le cas, on retourne immédiatement à l'appelant avec true (return true;). Sinon, on réinitialise c à 1 (c=1;) car on va maintenant pointer le premier caractère de la ligne suivante.

Ensuite nous allons boucler jusqu'à trouver une ligne non vide ou arriver en fin de texte. La boucle en do commence, on regarde si l+1 est différent du nombre de lignes de la **TStringList** et on met ce résultat booléen dans la variable

existe (existe=(++l!=Texte->Count);). Ici on utilise une pré-incrémentation de l, la comparaison se fait donc avec l incrémenté. Si l incrémenté est différent de Texte->Count, c'est que la ligne suivante existe et la variable sera à true.

N'oubliez pas que l'indice Text->Count est OutOfRange puisque les indices vont de 0 à Text->Count-1.

Si **existe** est vrai à (if(existe)) donc si la ligne suivante existe, encore faut-il qu'elle contienne quelque chose. On positionne donc la à variable booléenne vide en fonction du fait que la longueur de la ligne l est nulle ou non à (vide=!Texte->Strings[l].Length();). Une valeur numérique peut toujours servir de booléen puisque 0 équivaut à faux et non zéro à vrai. On exploite ici ce fait. Si n est une valeur, n équivaut à !0 (non zéro) donc if(n) correspondra à if(n !=0) à et !n équivaut à 0 donc if(!n) équivaut à if(n==0). La boucle se fait tant qu'une ligne existe et qu'elle est vide à (while(existe && vide);).

*Remarquez que si la ligne n'existe pas, vide n'est pas renseigné mais cela n'a aucune importance puisque existe étant faux, le ET logique donnera faux quelle que soit la valeur de vide et la boucle s'arrêtera à en renvoyant existe à savoir false. La boucle est terminée, on renvoie à l'appelant la valeur de la variable **existe** à (return existe;).*

2.10 - Voici comment nous allons traiter une occurrence

```
void __fastcall TTraitement::TraiteOcc(int l,int c)
    bool ok;
    do
        ok=CarPrec(l,c) && Ordre(liml,limc,l,c);
        if(ok)
            if (TrouveBorne(l,c))
                TraiteBorne(l,c);
                ok=false;
    while (ok);
```

On exécute cette fonction parce qu'une occurrence de l'extension a été trouvée avec son signe « " » fermant (fonction booléenne Trouve) et cette occurrence commence à partir de la position (l,c) c'est-à-dire qu'à partir de (l,c), on a tous les caractères de l'extension cherchée EXT. On va d'abord chercher en reculant le HREF correspondant et si on trouve par reculs successifs HREF, on opérera le traitement. On va utiliser un booléen pour contrôler la boucle en do-while (bool ok;).

La boucle commence, on regarde si on peut reculer à partir de la position (l,c) sans dépasser la limite (liml,limc) qui représente la position de la dernière modification (ok=CarPrec(l,c) && Ordre(liml,limc,l,c);). **CarPrec** est une procédure booléenne qui nous fait reculer d'un caractère et qui dit si ce caractère existe, c'est le symétrique de CarSuiv. Ordre vérifie que la première position est strictement inférieure à la deuxième c'est-à-dire que l'on ait dans l'ordre première position, deuxième position donc que (liml,limc) < (l,c).

Cette précaution est nécessaire. Imaginons qu'une première extension ait été correcte donc traitée et qu'une deuxième soit trouvée par hasard. Si on allait au-delà par recul de la limite (liml,limc), on accéderait au HREF précédant et le programme croirait qu'il s'agit du HREF correspondant à l'extension trouvée par hasard. Il faut donc segmenter la recherche.

Le HREF doit se trouver après cette limite (liml,limc) sinon l'occurrence n'est pas traitée. Si le caractère précédent existe et qu'on est dans nos limites (if(ok)), on cherche la borne en reculant, on entend par borne la chaîne HREF (constante Borne). Si cette borne a été trouvée par recul (if (TrouveBorne(l,c))), on va alors traiter cette borne (TraiteBorne(l,c);). Puis on met ok à false pour arrêter la boucle (ok=false;). La boucle se fait tant que ok est vrai (while (ok);). Il n'y a rien d'autre, la fonction ne renvoie rien, l'arrêt a lieu soit parce que une borne a été trouvée, soit parce qu'un recul a été refusé.

2.11 - Voici comment nous reculons d'un caractère

```
bool __fastcall TTraitement::CarPrec(int& l,int& c)
    bool existe;
    if(--c)
        return true;

    do
        existe = (--l >= 0);
        if (existe)
            c=Texte->Strings[l].Length();
    while(existe && !c);
    return existe;
```

Comme pour **CarSuiv**, les variables l et c sont envoyées comme référence (opérateur &), ce qui permet de communiquer ces valeurs à l'appelant. On crée une variable booléenne (bool existe;). Si c décrémenté est non nul, on retourne tout de suite à l'appelant avec true (if(--c) return true;). Ce sera le cas le plus fréquent, c n'est pas égal à 1 (indice minimal des **AnsiString**), sa seule décrémentation suffit à pointer le caractère précédent.

Sinon, c est nul après décrémentation, il faut donc reculer à la ligne précédente. Mais comme elle peut être vide, il faut boucler jusqu'à pointer une ligne non nulle. On entre donc dans une boucle en do. On positionne la variable booléenne existe en fonction du fait que l décrémenté est positif ou nul (existe=(--l>=0);). En effet, l'indice minimal de l est zéro (l va de 0 à Texte->Count-1). Si la ligne précédente existe (if (existe)), on lit dans c le nombre de caractères de cette ligne (c=Texte->Strings[l].Length();) et on boucle tant que la ligne existe et que son nombre de caractères est nul (while(existe && !c);).

La boucle s'arrêtera donc soit parce que existe est faux (il n'y a pas de ligne précédente), soit parce que existe étant vrai, !c est **faux** donc c **vrai** donc c non nul (se souvenir que a signifie « a non nul » et !a signifie « a nul », if(a) peut se lire « si a vaut quelque chose » donc si a!=0 et if(!a) peut se lire « si a ne vaut pas quelque chose » donc si a=0). La variable existe contient la réponse, on la renvoie à l'appelant (return existe;). Notez que dans la plupart des cas, on ne rentre pas du tout dans cette boucle mais le programme fonctionnerait toujours en cas de présentation aberrante. Imaginons qu'un lien HTML soit écrit comme ceci sur plusieurs lignes avec des lignes vides.

```
<a href="Titre.mid">Titre</a>
```

notation acceptée d'ailleurs par le HTML, notre programme marcherait parfaitement. Pointant le m de mid, il verrait grâce à notre fonction **CarPrec** que le caractère précédent est le point trois lignes plus haut et que le caractère précédent du r est le t deux lignes plus haut et ainsi de suite. Dans un tel cas, le programme va d'ailleurs procéder à un nettoyage en virant tout ce qui se trouve entre les deux « " », on va alors obtenir d'abord.

```
<a href="">Titre</a>
```

puis il va rajouter le nom avec chemin donc

```
<a href="MIDI/Titre.mid">Titre</a>
```

C'est la raison pour laquelle, la position de reprise d'analyse se décide a posteriori. Il ne s'agit pas de la position de d de mid au moment où il est rencontré mais de la position du d de mid au moment de la réparation de l'occurrence, sinon on risquerait de sauter des lignes et même de se trouver OutOfRange vers la fin du fichier car après suppression de lignes, l'indice l peut très bien être trop grand. C'est donc après réparation de l'occurrence qu'on renseigne via les deux variables liml et limc de la classe principale la position de reprise d'analyse.

2.12 - Test de l'ordre des positions

```
bool __fastcall TTraitement::Ordre(int l1,int c1, int l2, int c2)
    if(l1==l2)
        return (c1<c2);
```

```
    return (l1<l2);
```

On cherche à savoir si la position (l1,c1) est strictement inférieure à la position (l2,c2). Si les lignes sont égales, c'est c1 et c2 qui décident (if(l1==l2) return (c1<c2);). Sinon, ce sont les lignes qui décident (return (l1<l2);).

2.13 - Recherche de la borne

```
bool __fastcall TTraitement::TrouveBorne(int l,int c)
    int i=Borne.Length();
    bool ok,rep;
    AnsiString Car;
    do
        Car=UpperCase(Texte->Strings[l][c]);
        rep=(Car==Borne[i]);
        if(rep)
            if(!--i)
                ok = false;
            else
                ok = CarPrec(l,c);
        else
            ok=false;
    while(ok);
    return rep;
```

On cherche ici à savoir si à partir de la position (l,c) envoyée en argument, on a la borne HREF. On va donc faire les comparaisons en reculant puisqu'on cherche cette borne par reculs successifs (fonction **TraiteOcc**). On crée la variable i ne l'initialisant au nombre de caractères de cette borne (int i=Borne.Length();), i pointe donc le dernier caractère de la borne soit Borne[i]. On crée deux variables booléennes, une pour contrôler la boucle et l'autre pour contenir la réponse (bool ok,rep;). On crée également un AnsiString qui contiendra les caractères successifs (AnsiString Car;).

La boucle en do commence. On lit dans **Car** le caractère pointé avec conversion en majuscule (Car=UpperCase(Texte->Strings[l][c]);). Ensuite on positionne la variable **rep** en fonction du fait que le caractère lu correspond au caractère de la borne pointé par i (rep=(Car==Borne[i]);). Si ces caractères coïncident (if(rep)) on met ok à false si i prédécrémenté est nul (if(!--i) ok=false;).

En effet, i va de Borne.Length() à 1 par décrémentation, si sa décrémentation donne zéro, cela signifie que les Borne.Length() comparaisons ont été positives, en conséquence de quoi la borne a été trouvée, donc on met ok à false pour stopper la boucle et comme **rep** est à true, on renverra vrai à l'appelant. Sinon, si i n'est pas nul après décrémentation, on positionne ok en fonction du fait qu'il existe un caractère précédent (ok=CarPrec(l,c);). Si **rep** est faux, on se contente de mettre ok à false pour arrêter la boucle (else ok=false;). On continue tant que ok est vrai (while(ok);). La variable **rep** contient la réponse, on la retourne à l'appelant (return rep;).

2.14 - Traitement de la borne

```
void __fastcall TTraitement::TraiteBorne(int l,int c)
    if(TrouveSigne(l,c,'=', true))
        if(TrouveSigne(l,c,'"',true))
            if(Ordre(l,c,finl,finc))
                RepareOcc(l,c,finl,finc);
```

On exécute cette fonction parce que HREF a été trouvé par reculs successifs à partir de l'extension elle aussi trouvée avec son signe « " » fermant en position (finl,finc) (fonction booléenne Trouve). La position envoyée en argument pointe le F de HREF. On vérifie la validité de cette borne en exigeant que le premier signe à partir de ce F qui ne soit pas un espace soit le signe « = » (if(TrouveSigne(l,c,'=', true))). On appelle ici la fonction booléenne **TrouveSigne** avec son dernier argument à true, ce qui signifie que cette recherche ne doit pas se faire au-delà de (finl,finc) qui est la position du signe « " » fermant de l'extension trouvée.

Si cette précaution n'était pas prise, on pourrait, en cas de fichier HTML farfelu, trouver le signe « = » au-delà de cette limite, ce qui ne serait pas interprétable.

Si cette première condition est réalisée, on exige une seconde chose à savoir qu'il y ait après le signe « = » le signe « " » ouvrant (if(TrouveSigne(l,c,"",true))). On est sûr ici de trouver ce signe puisqu'il se trouve après l'extension (fonction booléenne **Trouve**). Cela dit, il faut s'assurer qu'il ne s'agit pas du signe fermant mais du même signe situé strictement avant. Si cette deuxième condition est réalisée à savoir que le signe « " » a été trouvé, on sait qu'il se trouve à la position (l,c) car **TrouveSigne** communique avec l'appelant (opérateur &). Dans ce cas il faut une ultime condition pour que l'occurrence soit valide à savoir que la position (l,c) soit strictement inférieure à (finl,finc) car en cas d'égalité, cela signifierait qu'il n'y a qu'une seule fois le signe « " » et non deux.

Si cette ultime condition est respectée, l'occurrence est vraiment valide et on la répare (if(Ordre(l,c,finl,finc)) RepareOcc(l,c,finl,finc);). On ne répare donc l'occurrence qu'en cas de certitude : l'extension a été trouvée, il y a le signe « " » après en position (finl,finc) (avec tolérance illimitée d'espaces), il y a HREF avant l'extension mais après la dernière intervention située à (liml,limc), il y a un signe « = » après le F de HREF (avec tolérance d'espaces), il y a le signe « " » encore après (avec tolérance d'espaces) et ce signe est strictement avant (finl,finc) qui est la position du signe « " » fermant. Si toutes ces conditions sont remplies, l'occurrence est valide et on procède à sa réparation, sinon on ne fait rien.

2.15 - Recherche d'un signe avec ou sans limite

```
bool TTraitement::TrouveSigne(int& l, int& c, AnsiString Signe, bool AvecLimite)
    AnsiString Car;
    while(true)
    if(!CarSuiv(l,c))
        return false;

    if(AvecLimite && !Ordre(l,c,finl,finc))
        return false;

    Car=Texte->Strings[l][c];
    if(Car==Signe)
        return true;

    if(Car!=' ')
        return false;
```

On cherche à partir de la position (l,c) le signe envoyé en argument. Le dernier argument dit s'il faut tenir compte de la limite (finl,finc) ou non. On déclare un **AnsiString** pour lire les caractères (AnsiString Car). On entre dans une boucle infinie de type while(true). C'est le corps de la boucle qui décidera de tout arrêter. Si le caractère suivant n'existe pas, on retourne immédiatement avec false (if(!CarSuiv(l,c)) return false;). Si la demande de vérification avec limite est à true et si l'ordre entre la position (l,c) et la limite (finl,finc) n'est pas respectée parce que (l,c) ne serait pas strictement avant (finl,finc), on retourne avec false (if(Avec Limite && !Ordre(l,c,finl,finc)) return false;).

Ici nous sommes dans les limites et le caractère suivant existe, on lit le caractère pointé (Car=Texte->Strings[l][c];). Si ce signe est celui qu'on cherche, on retourne immédiatement avec true (if(Car==Signe) return true;). Si ce n'est pas le cas, on ne tolère que l'espace comme autre possibilité sinon on retourne avec false (if(Car!=' ') return false;). Tout cela continue jusqu'à ce qu'une décision soit prise, soit fin de texte rencontré, soit limite demandée et rencontrée, soit signe trouvé, soit signe non trouvé et caractère autre qu'espace.

2.16 - Réparation de l'occurrence

```
void TTraitement::RepareOcc(int l1,int c1, int l2, int c2)
    AnsiString A, Lien=Chemin->Text+NomPur(l1,c1);
    CarSuiv(l1,c1);
    CarPrec(l2,c2);
    if(l1==l2)
        SupCas1(l1,c1,c2);
     else if(l2==l1+1)
```

```
            SupCas2(l1,c1,c2);
         else
            SupCas3(l1,c1,l2,c2);

      A=Texte->Strings[l1];
      A.Insert(Lien,c1);
      Texte->Strings[l1]=A;
      MODIF=true;
      liml=l1;
      limc=c1+Lien.Length()-1;
      NbModif++;
      Progression->Max=Texte->Count-1;
      Progression->Position=l1;
```

On répare ici l'occurrence dont on est absolument certain qu'elle est valide. On reçoit en argument deux positions (l1,c1) et (l2,c2) pointant respectivement le signe « " » ouvrant et le signe « " » fermant. On crée un AnsiString intermédiaire A pour l'insertion des caractères et un autre qui contiendra le chemin et le nom pur du lien c'est-à-dire ce qu'on devra écrire entre les signes « " » en remplacement de ce qui s'y trouve (AnsiString A, Lien=Chemin->Text+NomPur(l1,c1);).

On appelle ici une fonction qui lit ce qui se trouve entre les signes « " » et en déduit le nom pur du lien. Comme (l1,c1) pointe le signe « " » ouvrant, il faut avancer d'un caractère pour pointer le premier caractère du lien (CarSuiv(l1,c1);) et comme (l2,c2) pointe le signe « " » fermant, il faut reculer d'un caractère pour pointer le dernier caractère du lien (CarPrec(l2,c2);).

Notez qu'on ne teste pas la valeur booléenne de retour puisqu'on est là certain que ces caractères existent, on se contente d'appeler la fonction. Ici on procède au nettoyage du lien ancien.

On distingue trois cas.

- Premier cas, tout se passe sur une seule ligne, ce qui normalement sera le cas le plus fréquent (if(l1==l2) SupCas1(l1,c1,c2);).
- Deuxième cas, le lien à réparer se trouve sur deux lignes consécutives (else if(l2==l1+1) SupCas2(l1,c1,c2);).
- Troisième et dernier cas, l'ancien lien à modifier tient sur plus de deux lignes (else SupCas3(l1,c1,l2,c2);). À ce stade, le lien est nettoyé, on a viré tout ce qui se trouve entre les signes « " » ouvrant et fermant, lesquels sont maintenant consécutifs (sur une ou deux lignes).

On lit dans l'**AnsiString** intermédiaire A prévu à cet effet la ligne à modifier (A=Texte->Strings[l1];). Le caractère d'insertion du nouveau lien est c1 donc on insère le nouveau lien à partir de c1 (A.Insert(Lien,c1);) puis on stocke ce résultat au même endroit (Texte->Strings[l1]=A;), validant ainsi la modification.

*À noter qu'une syntaxe directe du type Texte->Strings[l1].Insert(Lien,c1) est à déconseiller, l'expérience montre que dans certains cas, l'instruction ne réagit pas, rien ne se passe, il est donc préférable de décomposer en utilisant une variable intermédiaire. Il en ira de même plus loin au moment de la suppression du lien via la méthode **Delete**.*

On renseigne maintenant les nouvelles limites au-delà desquelles on ne pourra pas reculer pour chercher un HREF, pour la ligne il s'agit de l1 (liml=l1;) et pour le caractère, il s'agit du dernier caractère du lien (limc=c1+Lien.Length()-1;). Si l'extension est mid, il s'agit du d de mid dont on est sûr qu'il se trouve sur la ligne puisqu'on vient de l'écrire. On incrémente le compteur de modifications (NbModif++;). On met à jour la barre de progression. Comme le nombre de lignes est susceptible de changer, l'indice maximal l'est aussi, on le choisit comme maximum (Progression->Max=Texte->Count-1;). Et on renseigne la position de la progression qui n'est autre que l'indice de la ligne traitée (Progression->Position=l1;).

2.17 - Premier cas de suppression

```
void __fastcall TTraitement::SupCas1(int l,int c1, int c2)
    AnsiString A=Texte->Strings[l];
    A.Delete(c1,c2-c1+1);
    Texte->Strings[l]=A;
```

Premier cas de nettoyage du lien : le lien se trouve sur une seule ligne entre c1 et c2. Donc on efface c2-c1+1 caractères à partir de la position c1 en passant par une variable intermédiaire. On lit donc d'abord le lien dans un AnsiString A (AnsiString A=Texte->Strings[l];), on procède à la suppression (A.Delete(c1,c2-c1+1);) et on valide par une réécriture au même emplacement (Texte->Strings[l]=A;). Si le lien tient sur n caractères à partir du caractère c, ce lien s'écrit de c à c+n-1 donc pour obtenir le nombre de caractères du lien, il faut ajouter 1 à leur différence puisque (c+n-1)-(c)+1=n.

2.19 - Deuxième cas de suppression

```
void __fastcall TTraitement::SupCas2(int l,int c1, int c2)
    AnsiString A=Texte->Strings[l];
    A.Delete(c1,Texte->Strings[l].Length()-c1+1);
    Texte->Strings[l]=A;
    A=Texte->Strings[l+1];
    A.Delete(1,c2);
    Texte->Strings[l+1]=A;
```

Ici le lien tient sur deux lignes consécutives soit l et l+1. Il faut donc effacer tout ce qui se trouve au-delà de la position c1 (A.Delete(c1,Texte->Strings[l].Length()-c1+1);) et pour la ligne suivante tout ce qui se trouve jusqu'à c2 inclus (A.Delete(1,c2);), toujours via un AnsiString intermédiaire. Dans ce cas, la ligne l+1 commence par le signe « " » fermant.

2.20 - Troisième cas de suppression

```
void TTraitement::SupCas3(int l1,int c1, int l2,int c2)
    for(int i=0;i<l2-l1-1;i++)
        Texte->Delete(l1+1);

    SupCas2(l1,c1,c2);
```

Ici le lien tient sur plus de deux lignes. On vire toutes les lignes situées entre la première et la dernière c'est-à-dire l2-l1-1 lignes (for(int i=0;i<l2-l1-1;i++) Texte->Delete(l1+1);). À ce stade, on retombe dans le cas de lignes consécutives, on appelle ce cas qui finalise ce nettoyage (SupCas2(l1,c1,c2);).

2.21 - Extraction du nom pur du fichier

```
AnsiString __fastcall TTraitement::NomPur(int l,int c)
    bool ok=true;
    AnsiString Nom="", Car;
    do
        CarSuiv(l,c);
        Car=Texte->Strings[l][c];
        if(Car=='/')
            Nom="";
         else
            if (Car!='"')
                Nom=Nom+Car;
            else
                ok=false;
    while(ok);
    return Nom;
```

On extrait ici le nom pur du lien. On entend par nom pur le nom sans le chemin s'il existe. On utilise ici un booléen initialisé à true pour contrôler la boucle (bool ok=true;). On concaténera le nom dans un AnsiString et on lira les caractères à successifs dans un autre **AnsiString** (AnsiString Nom="", Car;). On entre dans une boucle en do. Comme la position de à départ donnée pour le signe « " » ouvrant, il faut déjà avancer d'un caractère pour pointer le premier caractère du lien à (CarSuiv(l,c);). Ici on ne fait aucun test, on sait d'une part que tous les caractères existent et on sait qu'il y a à un signe « " » fermant.

On lit maintenant le caractère pointé (Car=Texte->Strings[l][c];). Si ce caractère est un /, on à réinitialise la variable Nom à vide (if(Car=='/') Nom="";) car on ne veut que le nom pur du lien et non le chemin à susceptible de l'accompagner. Sinon, si ce n'est pas /, on regarde si c'est le signe « " » fermant. Si ce n'est pas le cas, on à ajoute à Nom le caractère trouvé (if (Car!="") Nom=Nom+Car;) sinon le signe « " » fermant a été trouvé, on met ok à false à pour arrêter la boucle (else ok=false;). La boucle se poursuit tant que ok est vrai (while(ok);). Au sortir de à cette boucle, Nom contient le nom pur du lien, on le renvoie à l'appelant (return Nom;).

2.22 - Evénement OnDestroy

```
void __fastcall TTraitement::FormDestroy(TObject *Sender)
    delete Texte;
```

Nous n'avons instancié par new qu'un **TStringList**, nous restituons ici la mémoire (delete Texte;)

2.23 - Variables et méthodes ajoutées à la classe principale

```
TStringList *Texte;
AnsiString EXT;
int NbModif;
bool MODIF;
int liml, limc;
int finl,finc;
void TraiteFic(AnsiString);
void TraiteFic1(AnsiString);
bool Trouve(int,int,AnsiString);
bool Confirme(AnsiString);
void RepareOcc(int,int,int,int);
void SupCas3(int,int,int,int);
bool TrouveSigne(int&,int&,AnsiString,bool);
void __fastcall TraitePos(int&,int&);
bool __fastcall CarSuiv(int&,int&);
bool __fastcall CarPrec(int&,int&);
void __fastcall TraiteOcc(int,int);
bool __fastcall ChaineFermee(int l,int c);
bool __fastcall Ordre(int,int,int,int);
bool __fastcall TrouveBorne(int,int);
void __fastcall TraiteBorne(int,int);
AnsiString __fastcall NomPur(int,int);
void __fastcall SupCas1(int,int,int);
void __fastcall SupCas2(int,int,int);
AnsiString __fastcall CompteRendu(int);
```

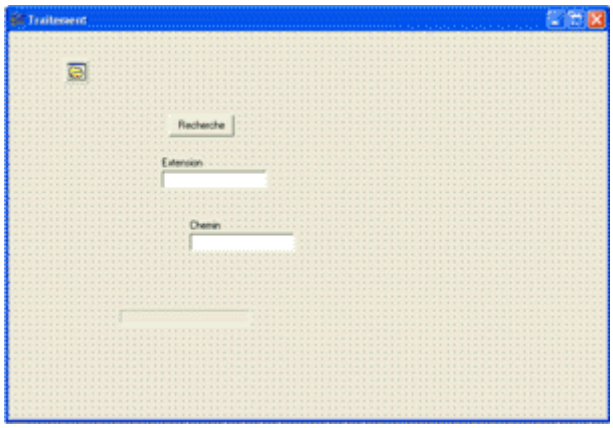
Nous avons supprimé la directive __fastcall si le nombre d'arguments est supérieur à 3 ou s'il contient un **AnsiString**. Cette finesse n'est d'ailleurs pas obligatoire au sens où si le passage des arguments par les registres est impossible, le compilateur ignore cette directive.

3 - Programme complet à télécharger

Pour faire fonctionner ce tutoriel :

- 1) Créez un répertoire HTML pour ce projet ;
- 2) **Téléchargez** la source dans le répertoire HTML (c'est un simple fichier texte) ;
- 3) Entrez dans C++ Builder 6 ;

- 4) Faites « Enregistrez le projet sous » ;
- 5) Au lieu de Unit1 entrez html ;
- 6) Rendez-vous au nouveau répertoire HTML nouvellement créé et enregistrez ;
- 7) Au lieu de **Project1** entrez chemin et enregistrez ;
- 8) Modifiez propriété **Name** de **Form1**, au lieu de Form1 entrez Traitement ;
- 9) Double-cliquez l'événement **OnDestroy** de cette fenêtre principale ;
- 10) Posez sur la fenêtre le composant **TOpenDialog** (onglet **Dialogues de la palette**) ;
- 11) Donnez à ce composant le nom OuvreFichier (propriété **Name**) ;
- 12) Posez un bouton sur la fenêtre (palette **Standard**) ;
- 13) Donnez au bouton le nom Recherche (propriété **Name**) ;
- 14) Double-cliquez sur le bouton pour créer l'événement **OnClick** ;
- 15) Posez sur la fenêtre un **TlabeledEdit** (palette Supplément) ;
- 16) Donnez-lui le nom Extension (propriété **Name**) ;
- 17) Posez un autre **TlabeledEdit** ;
- 18) Donnez-lui le nom Chemin (propriété **Name**) ;
- 19) Posez une barre de progression (palette **Win32**) ;
- 20) Donnez-lui le nom Progression (propriété **Name**) ;



- 21) Effacez tout ce qui se trouve dans **html.cpp** (source cpp créée par C++ Builder) ;
- 22) Remplacez cette source par le fichier texte téléchargé au début ;
- 23) Ajoutez dans la zone public:// Déclarations de l'utilisateur de html.hpp (juste après __fastcall TTraitement(TComponent* Owner);) les variables et les méthodes données précédemment ;
- 24) Sauvegardez le tout (disquettes en cascade) ;
- 25) Faites F9 pour compiler et exécuter.