

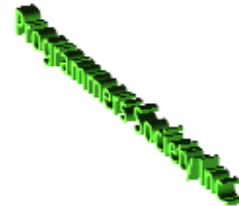
```

#define INCL_DOSPROCESS
#define INCL_DOSSESMGR
#define INCL_WIN
#include <os2.h>
#include <strstrea.h>
#include "logline.hpp"

void logline :: SetStartingOffsetSize(UINT size )
{
    StartingOffsetSize = size;
}
UINT logline :: GetStartingOffsetSize()
{
    return (StartingOffsetSize );
}
void logline :: SetNbSpacesFromOffsetToData( UINT size = 3)
{
    NbSpacesFromOffsetToData = size;
}
UINT logline :: GetNbSpacesFromOffsetToData( )
{
    return (NbSpacesFromOffsetToData );
}
void logline :: SetNbSpacesBetweenDataTokens( UINT size = 2)
{
    NbSpacesFromOffsetToData = size;
}
UINT logline :: GetNbSpacesBetweenDataTokens( )
{
    return ( NbSpacesBetweenDataTokens );
}
void logline :: SetDataTokenLen(UINT size = 8)
{
    DataTokenLen = size;
}
UINT logline :: GetDataTokenLen()
{
    return DataTokenLen;
}
void logline :: SetNbSpacesFromEOFDataToASCIIrep( UINT size = 3)
{
    NbSpacesFromEOFDataToASCIIrep = size;
}

void logline :: SetLineLength(UINT size = 80)
{
    LineLength = size;
}
UINT logline :: GetLineLength()
{
    return LineLength;
}
UINT logline :: GetNbSpacesFromEOFDataToASCIIrep( )
{
    return ( NbSpacesFromEOFDataToASCIIrep );
}

```



```

UINT logline :: GetNbofASCIIinALine() { return NbofASCIIinALine;}
void logline :: SetNbofASCIIinALine(UINT Nb) {NbofASCIIinALine = Nb;}
IString logline :: GetDumpFormattedLine() { return DumpFormattedLine;}
Boolean logline :: GetFeasibility() { return feasible;}
Boolean logline :: IsFormatFeasible(UINT NbDataSlabs)
{
    UINT NbofCharPerTkn = DataTokenLen / BaseDivider;
    UINT NbASCIIrep = NbDataSlabs * NbofCharPerTkn;

    return (
        (StartingOffsetSize + NbSpacesFromOffsetToData
         + NbSpacesFromEOFDataToASCIIrep
         + (NbDataSlabs * DataTokenLen)
         + ( ( NbDataSlabs -1) * NbSpacesBetweenDataTokens )
         + NbASCIIrep) <= LineLength      ? TRUE : FALSE );
}
const IString& logline :: FilterBase(IString& refDestStr, const IString& ref-
SrcStr)
{
    switch (Base) {
    case HEX:
        refDestStr = refSrcStr;
        refDestStr.c2x();
        break;
    case OCTAL:
        refDestStr = "";
        A2O(refDestStr, refSrcStr);
        break;
    case BINARY:
        break;
    default:
        // Error control || Silence
        break;
    } /* endswitch */
    return refDestStr;
}
//Take a String representing the value of the offset and fill StartingOffset
void logline :: BuildStartingoffset(const ULONG GivenValue)
{
    IString Value = (IString) GivenValue;
    switch (Base) {
    case HEX:
        (void) A2H(Value, Value);
        break;
    case OCTAL:
        (void) A2O(Value, Value);
        break;
    case BINARY:
        break;
    default:
        break;
    } /* endswitch */

    IString frontpad(0 /*NULL*/, StartingOffsetSize - Value.length() );

    // Now construct the StartingOffset string
    StartingOffset = frontpad + Value;
}

```



```

//Builds a single string for the data tokens in a single dump formatted line
void logline :: BuildDataTokens(IString& InputStr)
{
    IString tempStr ;
    tempStr = InputStr;
    IString spaces(NULL, NbSpacesBetweenDataTokens);
    InputStr = "";

    for (UINT i=0 ; i < NbofDataTokens ;i++ ) {
        InputStr += tempStr.subString(1+ i* DataTokenLen, DataTokenLen);
        InputStr += (i == (NbofDataTokens -1) ) ? (IString)" " : spaces;
    } /* endfor */
}

void logline :: MakePrintable(IString& PrintableLine, const IString& Input-
Line)
{
    if (InputLine.isPrintable()) {
        PrintableLine = InputLine;          //Identity transform
    } else {
        PrintableLine = InputLine;
        CHAR *ptrChar = (char *) PrintableLine;
        while (*ptrChar) {
            if (*ptrChar < ' ' || *ptrChar > '~') {
                *ptrChar++ = '.';
            } /* endif */
        } /* endwhile */
    }
}

// It should take only a line's worth of DataToken(ASCII), so it can format
only a line
void logline :: DoFormat(const IString& line, const ULONG offsetValue)
{
    IString BasedLine, DataTokens, ASCIIPrintableLine ;
    if (GetFeasibility()) {
        //Format is feasible, take the info from class member and format line
        // Change the representation ASCII ---> [OCTAL | HEX | WHAT]
        BasedLine = FilterBase(BasedLine, line);

        //Build the offset string
        BuildStartingoffset(offsetValue);

        // Build the data tokens as single string
        DataTokens = BasedLine;
        BuildDataTokens(DataTokens);

        // Build the Ascii rep - The inputline may contain non-printables !!
        MakePrintable(ASCIIPrintableLine, line);

        //Now construct the whole line
        IString SpacesFromOffsetToData (" ", NbSpacesFromOffsetToData );
        IString SpacesFromEOFDataToASCIIrep (" ",
            NbSpacesFromEOFDataToASCIIrep);

```

```

        DumpFormattedLine = StartingOffset + SpacesFromOffsetToData
                          + DataTokens + SpacesFromEOFDataToASCIIrep
                          + ASCIIPrintableLine;

    } else {
        DumpFormattedLine="Line is not dump formattable as per specification ";
    } /* endif */
}
logline :: logline(IString& InputStr, UINT RequestedBase)
    : //Do default Initialization --- Make these hardcode as default
      // The defaults are Hex dependent
      Base(RequestedBase),
      StartingOffsetSize(5),           //Free to choose
      NbSpacesFromOffsetToData(3),     //Free to choose
      NbSpacesBetweenDataTokens(2),   //Free to choose
      DataTokenLen(8),                 // Hex dependent
      NbSpacesFromEOFDataToASCIIrep(5),
      feasible(0),                      // Internal State
      NbofDataTokens(4),                // Free to choose
      BaseDivider(HEX),                 // Hex dependent
      NbofASCIIinALine((DataTokenLen / BaseDivider) * NbofDataTo-
kens),

      LineLength(80)

{
    // How do we cope with user supplied requested sizes
    // How do we test the feasibility of the requested format

    switch (Base) {
    case HEX:
        BaseDivider = HEX;
        feasible = (DataTokenLen % BaseDivider) ? 0 : 1 ;
        break;
    case OCTAL:
        BaseDivider = OCTAL;
        feasible = (DataTokenLen % BaseDivider) ? 0 : 1 ;
        break;
    case ASCII:
        BaseDivider = ASCII;
        feasible = (DataTokenLen % BaseDivider) ? 0 : 1 ;
        break;
    case BINARY:
        BaseDivider = BINARY;
        feasible = (DataTokenLen % BaseDivider) ? 0 : 1 ;
        break;
    default:
        break;
    } /* endswitch */

    // Now test whole line feasibility
    feasible = IsFormatFeasible(NbofDataTokens);
}

```

Programmers Society Inc

```

const IString& logline :: A2H(IString& refDestStr, const IString& refSrcStr)
{
    if (refDestStr == refSrcStr) {
        IString temp = refSrcStr;
        return ( refDestStr = temp.c2x() );
    } else {
        refDestStr = refSrcStr;
        return ( refDestStr.c2x() );
    } /* endif */
}
// Given an ASCII string map it to an octally represented string
const IString& logline :: A2O(IString& refDestStr, const IString& refSrcStr)
{
    IString bp;
    // buf is an output stream with refDestStr is attached as a stream buffer
    ostrstream buf((char *) refDestStr, refDestStr.length()+1, ios::out);
    for (UINT i=0 ; i <refSrcStr.length() ; i++ ) {
        bp = refSrcStr.subString(i+1, 1);
        bp.c2d(); //decimally represented ASCII char
        if (atol(bp) < 64) {
            if (atol(bp) < 8) {
                buf << oct << "00" <<atol(bp);
            }
            else {
                buf << oct << "0" << atol(bp) ;
            }
        }else
            buf << oct <<atol(bp) ;
    } /* endfor */

    //cout <<endl << "In A2O refDestStr=" << refDestStr <<endl;
    return ( refDestStr) ;
}

logline :: ~logline()
{
}

```

