

```
#include <algorithm>
```

```
/******
```

```
// s_VolumeLevelAtPrice
```

```
struct s_VolumeLevelAtPrice
```

```
{  
    //-- Members -----
```

```
    int PriceInTicks;
```

```
    // The maximum volume of a trade at price, above or equal to the set  
    // volume level.
```

```
    unsigned int MaxVolume;
```

```
    // The total volume of all trades at price, above or equal to the set  
    // volume level.
```

```
    unsigned int TotalVolume;
```

```
    // The total volume of all bid trades at price, above or equal to the set  
    // volume level.
```

```
    unsigned int BidTradeVolume;
```

```
    // The total volume of all ask trades at price, above or equal to the set  
    // volume level.
```

```
    unsigned int AskTradeVolume;
```

```
    //-- Methods -----
```

```
    s_VolumeLevelAtPrice();
```

```
};
```

```
/*=====*/
```

```
inline s_VolumeLevelAtPrice::s_VolumeLevelAtPrice()
```

```
    : PriceInTicks(0)  
    , MaxVolume(0)  
    , TotalVolume(0)  
    , BidTradeVolume(0)  
    , AskTradeVolume(0)
```

```
{  
}
```

```
/******
```

```
// s_VolumeAtPriceV2
```

```
struct s_VolumeAtPriceV2
```

```
{  
    //-- Members -----
```

```
    int PriceInTicks;
```

```
    unsigned int Volume;
```

```
    unsigned int BidVolume;
```

```
    unsigned int AskVolume;
```

```
    unsigned int NumberOfTrades;
```

```
    //-- Methods -----
```

```
    s_VolumeAtPriceV2();
```

```
    s_VolumeAtPriceV2
```

```
    ( const unsigned int Volume  
    , const unsigned int BidVolume  
    , const unsigned int AskVolume  
    , const unsigned int NumberOfTrades  
    );
```

```

s_VolumeAtPriceV2& operator += (const s_VolumeAtPriceV2& Rhs);
s_VolumeAtPriceV2& operator -= (const s_VolumeAtPriceV2& Rhs);

bool IsEmpty() const
{
    return PriceInTicks == 0 && Volume == 0;
}
};

/*=====*/
inline s_VolumeAtPriceV2::s_VolumeAtPriceV2()
: PriceInTicks(0)
, Volume(0)
, BidVolume(0)
, AskVolume(0)
, NumberOfTrades(0)
{
}

/*=====*/
inline s_VolumeAtPriceV2::s_VolumeAtPriceV2
( const unsigned int Volume
, const unsigned int BidVolume
, const unsigned int AskVolume
, const unsigned int NumberOfTrades
)
: PriceInTicks(0)
, Volume(Volume)
, BidVolume(BidVolume)
, AskVolume(AskVolume)
, NumberOfTrades(NumberOfTrades)
{
}

/*=====*/
inline s_VolumeAtPriceV2& s_VolumeAtPriceV2::operator +=
( const s_VolumeAtPriceV2& Rhs
)
{
    Volume += Rhs.Volume;
    BidVolume += Rhs.BidVolume;
    AskVolume += Rhs.AskVolume;
    NumberOfTrades += Rhs.NumberOfTrades;

    return *this;
}

/*=====*/
inline s_VolumeAtPriceV2& s_VolumeAtPriceV2::operator -=
( const s_VolumeAtPriceV2& Rhs
)
{
    if (Rhs.Volume > Volume)
        Volume = 0;
    else
        Volume -= Rhs.Volume;

    if (Rhs.BidVolume > BidVolume)
        BidVolume = 0;
    else
        BidVolume -= Rhs.BidVolume;

    if (Rhs.AskVolume > AskVolume)
        AskVolume = 0;
}

```

```

else
    AskVolume -= Rhs.AskVolume;

NumberOfTrades -= Rhs.NumberOfTrades;

return *this;
}

```

```

/*****
// c_VAPContainerBase

```

This is a container class related to bar data. Each bar may contain multiple elements of the given template type for specific prices within the bar.

The template elements are stored in a 1-dimensional dynamically allocated array. A 2nd 1-dimensional dynamically allocated array is used to keep track of which elements in the 1st array belong to which bar indexes.

```

-----*/
template<typename t_VolumeAtPrice>
class c_VAPContainerBase
{
    ///-- Public Methods -----
public:
    c_VAPContainerBase
        ( const unsigned int InitialAllocationElements = 1024
        );
    c_VAPContainerBase(const c_VAPContainerBase&) = delete;
    c_VAPContainerBase(c_VAPContainerBase&& rr_Source);
    ~c_VAPContainerBase();

    c_VAPContainerBase& operator = (const c_VAPContainerBase&) = delete;
    c_VAPContainerBase& operator = (c_VAPContainerBase&& rr_Right);

    void Clear();
    void ClearFromBarIndexToEnd(const unsigned int BarIndex);

    unsigned int GetNumberOfBars() const;

    unsigned int GetSizeAtBarIndex(const unsigned int BarIndex) const;

    void Swap(c_VAPContainerBase& r_That);

    bool GetVAPElement
        ( const int PriceInTicks
        , const unsigned int BarIndex
        , t_VolumeAtPrice** p_VAP
        , const bool AllocateIfNeeded = false
        );
    bool GetVAPElementAtIndex
        ( const unsigned int BarIndex
        , int VAPDataIndex
        , t_VolumeAtPrice** p_VAP
        , bool ReturnErrorOutOnOfBounds = false
        ) const;

    bool GetVAPElementAtIndex
        (const unsigned int BarIndex
        , int VAPDataIndex
        , const t_VolumeAtPrice** p_VAP
        , bool ReturnErrorOutOnOfBounds = false
        ) const;

    bool GetVAPElementForPriceIfExists
        ( const unsigned int BarIndex
        , const int PriceInTicks

```

```

, t_VolumeAtPrice** p_VAP
, unsigned int& r_InsertionIndex
);

bool GetNextHigherVAPElement
( const unsigned int BarIndex
, int& r_PriceInTicks
, const t_VolumeAtPrice** p_VAP
) const;

bool GetNextLowerVAPElement
( const unsigned int BarIndex
, int& r_PriceInTicks
, const t_VolumeAtPrice** p_VAP
) const;

// This was the original function signature.
bool GetHighAndLowPriceTicksForBarIndex
( const unsigned int BarIndex
, int& r_HighPriceInTicks
, int& r_LowPriceInTicks
) const;

// Overloaded function allows for optional return parameters.
bool GetHighAndLowPriceTicksForBarIndex
( const unsigned int BarIndex
, int* p_HighPriceInTicks
, int* p_LowPriceInTicks
) const;

bool GetHighAndLowPriceTicksForBarIndexRange
( const unsigned int FirstBarIndex
, const unsigned int LastBarIndex
, int* p_HighPriceInTicks
, int* p_LowPriceInTicks
) const;

const t_VolumeAtPrice& GetVAPElementAtPrice
( const unsigned int BarIndex
, const int PriceInTicks
) const;

bool AddVolumeAtPrice
( const int PriceInTicks
, const unsigned int BarIndex
, const t_VolumeAtPrice& VolumeAtPrice
);

bool SubtractVolumeAtPrice
( const int PriceInTicks
, const unsigned int BarIndex
, const t_VolumeAtPrice& VolumeAtPrice
);

bool SetVolumeAtPrice
(const int PriceInTicks
, const unsigned int BarIndex
, const t_VolumeAtPrice& VolumeAtPrice
);

//--- Private Static Methods -----
private:
static bool PriceIsLess
( const t_VolumeAtPrice& First
, const t_VolumeAtPrice& Second
);

```

//--- Private Methods -----

private:

```
bool EnsureBarExists
( const unsigned int BarIndex
, const bool AllocateIfNeeded
);
bool InitialAllocation();
bool AllocateBar(const unsigned int BarIndex);
bool AllocateElement(const unsigned int ElementIndex);
```

```
unsigned int GetFirstDataElementIndexForBar
( const unsigned int BarIndex
) const;
```

```
bool GetBeginEndIndexesForBarIndex
( const unsigned int BarIndex
, unsigned int* p_BeginIndex
, unsigned int* p_EndIndex
) const;
```

```
void GetHighAndLowPriceTicks
( const unsigned int BeginIndex
, unsigned int EndIndex
, int* p_HighPriceInTicks
, int* p_LowPriceInTicks
) const;
```

//--- Private Members -----

private:

```
// Note: The order of these private members must not change in order
// to preserve compatibility with custom study DLLs that may have
// been built with a different version of this code. These private
// members are shared between those DLLs and Sierra Chart.
```

```
// This points to an array with an element for each bar. The value
// at each element is the index of the first element in the VAP data
// array that contains data for the bar.
```

```
unsigned int* m_p_BarIndexToFirstElementIndexArray = nullptr;
```

```
// The number of elements that are currently used in the above array.
```

```
unsigned int m_NumberOfBars = 0;
```

```
// The number of elements that are currently allocated for the above
// array. The elements allocated is usually greater than the
// elements used so that the memory for the array does not need to be
// reallocated for every new element.
```

```
unsigned int m_NumElementsAllocated = 0;
```

```
// This points to an array of all of the Volume at Price data
// elements that are stored in this container.
```

```
t_VolumeAtPrice* m_p_VAPDataElements = nullptr;
```

```
// The number of elements that are currently used in the above array.
```

```
unsigned int m_NumElementsUsed = 0;
```

```
// The number of elements that are currently allocated for the above
// array.
```

```
unsigned int m_NumBarsAllocated = 0;
```

```
// This is the number of bars that are initially allocated when the
// arrays for the container are initially allocated.
```

```
const unsigned int m_InitialAllocationElements = 0;
```

```
// These members are not used any longer but need to remain here
// because older compiled studies will alter these values. So they
```

```

    // must remain in the structure.
    int m_Unused_LastSortedBarIndex = 0;
    int m_Unused_LastSortedBarSize = 0;
};

/*=====*/
template<typename t_VolumeAtPrice>
inline c_VAPContainerBase<t_VolumeAtPrice>::c_VAPContainerBase
( const unsigned int InitialAllocationElements
) : m_InitialAllocationElements(InitialAllocationElements)
{

}

/*=====*/
template<typename t_VolumeAtPrice>
inline c_VAPContainerBase<t_VolumeAtPrice>::c_VAPContainerBase
( c_VAPContainerBase&& rr_Source
)
: c_VAPContainerBase(rr_Source.m_InitialAllocationElements)
{
    Swap(rr_Source);
}

/*=====*/
template<typename t_VolumeAtPrice>
inline c_VAPContainerBase<t_VolumeAtPrice>::~c_VAPContainerBase()
{
    if (m_p_VAPDataElements != nullptr)
    {
        free(m_p_VAPDataElements);
        m_p_VAPDataElements = nullptr;
    }

    if (m_p_BarIndexToFirstElementIndexArray != nullptr)
    {
        free(m_p_BarIndexToFirstElementIndexArray);
        m_p_BarIndexToFirstElementIndexArray = nullptr;
    }
}

/*=====*/
template<typename t_VolumeAtPrice>
inline c_VAPContainerBase<t_VolumeAtPrice>&&
c_VAPContainerBase<t_VolumeAtPrice>::operator =
( c_VAPContainerBase&& rr_Right
)
{
    Swap(rr_Right);

    return *this;
}

/*=====*/
template<typename t_VolumeAtPrice>
inline void c_VAPContainerBase<t_VolumeAtPrice>::Clear()
{
    // Swapping with a new temporary object will effectively destroy and
    // re-initialize this object.
    Swap(c_VAPContainerBase<t_VolumeAtPrice>(m_InitialAllocationElements));
}

/*=====*/
template<typename t_VolumeAtPrice>
inline void c_VAPContainerBase<t_VolumeAtPrice>::ClearFromBarIndexToEnd

```

```

( const unsigned int BarIndex
)
{
    if (BarIndex >= m_NumberOfBars)
        return;

    // Clear the VAP data array.

    const unsigned int BeginIndex
        = m_p_BarIndexToFirstElementIndexArray[BarIndex];

    const unsigned int NumElementsToClear = m_NumElementsUsed - BeginIndex;

    memset
        ( m_p_VAPDataElements + BeginIndex
        , 0
        , NumElementsToClear * sizeof(t_VolumeAtPrice)
        );

    m_NumElementsUsed = BeginIndex;

    // Clear the StartingVAPIndexesForBarIndexes array.

    const unsigned int NumBarsToClear = m_NumberOfBars - BarIndex;

    memset
        ( m_p_BarIndexToFirstElementIndexArray + BarIndex
        , 0
        , NumBarsToClear * sizeof(unsigned int)
        );

    m_NumberOfBars = BarIndex;
}

/*=====*/
template<typename t_VolumeAtPrice>
inline unsigned int c_VAPContainerBase<t_VolumeAtPrice>::GetNumberOfBars() const
{
    return m_NumberOfBars;
}

/*=====
Returns the number of Volume at Price elements for the bar at the given
BarIndex.
-----*/
template<typename t_VolumeAtPrice>
inline unsigned int c_VAPContainerBase<t_VolumeAtPrice>::GetSizeAtBarIndex
( const unsigned int BarIndex
) const
{
    unsigned int BeginIndex, EndIndex;
    if (!GetBeginEndIndexesForBarIndex(BarIndex, &BeginIndex, &EndIndex))
        return 0;

    return EndIndex - BeginIndex;
}

/*=====*/
template<typename t_VolumeAtPrice>
inline void c_VAPContainerBase<t_VolumeAtPrice>::Swap
( c_VAPContainerBase& r_That
)

```

```

{
    using std::swap;

    // Note: not swapping m_InitialAllocationElements.

    swap(m_p_VAPDataElements, r_That.m_p_VAPDataElements);
    swap(m_NumElementsAllocated, r_That.m_NumElementsAllocated);
    swap(m_NumElementsUsed, r_That.m_NumElementsUsed);

    swap
        ( m_p_BarIndexToFirstElementIndexArray
        , r_That.m_p_BarIndexToFirstElementIndexArray
        );
    swap(m_NumBarsAllocated, r_That.m_NumBarsAllocated);
    swap(m_NumberOfBars, r_That.m_NumberOfBars);
}

/* =====
This method sets the given p_VAP pointer to point to the element at the
given BarIndex and PriceInTicks. If the requested element does not exist,
and AllocateIfNeeded is true (default), the element will be automatically
allocated. A new element can only be allocated if it is at or beyond the
last bar in the container. Returns true unless the requested element
could not be created or accessed.
----- */
template<typename t_VolumeAtPrice>
inline bool c_VAPContainerBase<t_VolumeAtPrice>::GetVAPElement
( const int PriceInTicks
, const unsigned int BarIndex
, t_VolumeAtPrice** p_VAP
, const bool AllocateIfNeeded
)
{
    *p_VAP = nullptr;

    if (!EnsureBarExists(BarIndex, AllocateIfNeeded))
        return false;

    unsigned int InsertionIndex = -1;

    if (GetVAPElementForPriceIfExists(BarIndex, PriceInTicks, p_VAP, InsertionIndex))
        return true;

    // The requested element does not exist.

    if (!AllocateIfNeeded)
        return false;

    // We cannot add because we are not at the last bar index segment in
    // the VAP array.
    if (BarIndex + 1 < m_NumberOfBars)
        return false;

    if (!AllocateElement(++m_NumElementsUsed))
    {
        Clear();
        return false;
    }

    if (InsertionIndex == -1)
    {
        InsertionIndex = m_NumElementsUsed - 1;
    }
}

```



```

else if (InsertionIndex < m_NumElementsUsed - 1)// Not inserting at the end.
{
    memmove(m_p_VAPDataElements + InsertionIndex + 1, m_p_VAPDataElements + InsertionIndex,
(m_NumElementsUsed - InsertionIndex - 1)* sizeof(t_VolumeAtPrice));
}

t_VolumeAtPrice& VolumeAtPrice = m_p_VAPDataElements[InsertionIndex];
VolumeAtPrice = t_VolumeAtPrice();
VolumeAtPrice.PriceInTicks = PriceInTicks;

*p_VAP = &m_p_VAPDataElements[InsertionIndex];

return true;
}

/*=====
VAPDataIndex is zero based, and is relative to the data for BarIndex.
-----*/

template<typename t_VolumeAtPrice>
inline bool c_VAPContainerBase<t_VolumeAtPrice>::GetVAPElementAtIndex
( const unsigned int BarIndex
, int VAPDataIndex
, t_VolumeAtPrice** p_VAP
, bool ReturnErrorOutOnOfBounds
) const
{
    *p_VAP = nullptr;

    unsigned int VAPSegmentBeginIndex, VAPSegmentEndIndex;
    if (!GetBeginEndIndexesForBarIndex
        ( BarIndex
        , &VAPSegmentBeginIndex
        , &VAPSegmentEndIndex
        )
    )
    {
        return false;
    }

    const int NumberOfElements = VAPSegmentEndIndex - VAPSegmentBeginIndex;

    if (VAPDataIndex < 0)
    {
        if (ReturnErrorOutOnOfBounds)
            return false;

        VAPDataIndex = 0;
    }

    if (VAPDataIndex > NumberOfElements - 1)
    {
        if (ReturnErrorOutOnOfBounds)
            return false;

        VAPDataIndex = NumberOfElements - 1;
    }

    *p_VAP = m_p_VAPDataElements + VAPSegmentBeginIndex + VAPDataIndex;

    return true;
}

/*=====
VAPDataIndex is zero based, and is relative to the data for BarIndex.
-----*/

```

```

-----*/
template<typename t_VolumeAtPrice>
inline bool c_VAPContainerBase<t_VolumeAtPrice>::GetVAPElementAtIndex
(const unsigned int BarIndex
, int VAPDataIndex
, const t_VolumeAtPrice** p_VAP
, bool ReturnErrorOutOnOfBounds
) const
{
    *p_VAP = nullptr;

    unsigned int VAPSegmentBeginIndex, VAPSegmentEndIndex;
    if (!GetBeginEndIndexesForBarIndex(BarIndex, &VAPSegmentBeginIndex, &VAPSegmentEndIndex ))
    {
        return false;
    }

    const int NumberOfElements = VAPSegmentEndIndex - VAPSegmentBeginIndex;

    if (VAPDataIndex < 0)
    {
        if (ReturnErrorOutOnOfBounds)
            return false;

        VAPDataIndex = 0;
    }

    if (VAPDataIndex > NumberOfElements - 1)
    {
        if (ReturnErrorOutOnOfBounds)
            return false;

        VAPDataIndex = NumberOfElements - 1;
    }

    *p_VAP = m_p_VAPDataElements + VAPSegmentBeginIndex + VAPDataIndex;

    return true;
}

/*=====
This function is for getting a specific Volume at Price element to modify.
Returns true unless the requested element does not exist.
-----*/
template<typename t_VolumeAtPrice>
inline bool c_VAPContainerBase<t_VolumeAtPrice>::GetVAPElementForPriceIfExists
( const unsigned int BarIndex
, const int PriceInTicks
, t_VolumeAtPrice** p_VAP
, unsigned int& r_InsertionIndex
)
{
    unsigned int BeginIndex = 0, EndIndex = 0;
    if (!GetBeginEndIndexesForBarIndex(BarIndex, &BeginIndex, &EndIndex))
    {
        return false;
    }

    // This is a temporary object only for the purpose of providing an object
    // of the same type for comparison.
    t_VolumeAtPrice Target;
    Target.PriceInTicks = PriceInTicks;

    t_VolumeAtPrice* p_BeginElement = m_p_VAPDataElements + BeginIndex;
    t_VolumeAtPrice* p_EndElement = m_p_VAPDataElements + EndIndex;

```

```

t_VolumeAtPrice* p_FoundElement = std::lower_bound(p_BeginElement, p_EndElement, Target, PriceIsLess);

if (p_FoundElement->PriceInTicks == PriceInTicks && p_FoundElement != p_EndElement)
{
    *p_VAP = p_FoundElement;
    return true;
}

r_InsertionIndex
= static_cast<unsigned int>
    ( p_FoundElement - p_BeginElement + BeginIndex
    );

return false;
}

/*=====
Searches for the VAP element in the bar at the given BarIndex at the next
price greater than the given r_PriceInTicks. Returns true unless there
is no higher price in the bar. r_PriceInTicks and p_VAP will be set
according to this next element.

If r_PriceInTicks is set to INT_MIN, this method will get the VAP element
with the lowest price for the bar.
-----*/
template<typename t_VolumeAtPrice>
inline bool c_VAPContainerBase<t_VolumeAtPrice>::GetNextHigherVAPElement
( const unsigned int BarIndex
, int& r_PriceInTicks
, const t_VolumeAtPrice** p_VAP
) const
{
    *p_VAP = nullptr;

    unsigned int BeginIndex, EndIndex;
    if (!GetBeginEndIndexesForBarIndex(BarIndex, &BeginIndex, &EndIndex))
        return false;

    // Return the element for the lowest price on the first iteration.
    if (r_PriceInTicks == INT_MIN)
    {
        t_VolumeAtPrice& r_LowestVAPElement = m_p_VAPDataElements[BeginIndex];

        *p_VAP = &r_LowestVAPElement;
        r_PriceInTicks = r_LowestVAPElement.PriceInTicks;

        return true;
    }

    // This is a temporary object only for the purpose of providing an object
    // of the same type for comparison.
    t_VolumeAtPrice Target;
    Target.PriceInTicks = r_PriceInTicks;

    const t_VolumeAtPrice* p_BeginElement = m_p_VAPDataElements + BeginIndex;
    const t_VolumeAtPrice* p_EndElement = m_p_VAPDataElements + EndIndex;

    // Note: upper_bound does a binary search (or something algorithmically
    // equivalent) and returns the lowest element that is greater than the
    // given Target.
    const t_VolumeAtPrice* p_NextHigherElement
        = std::upper_bound(p_BeginElement, p_EndElement, Target, PriceIsLess);

    // Return false if we have reached the end of the data for the bar.

```

```

    if (p_NextHigherElement >= p_EndElement)
        return false;

    *p_VAP = p_NextHigherElement;
    r_PriceInTicks = p_NextHigherElement->PriceInTicks;

    return true;
}

/*=====
Searches for the VAP element in the bar at the given BarIndex at the next
price less than the given r_PriceInTicks. Returns true unless there is
no lower price in the bar. r_PriceInTicks and p_VAP will be set
according to this next element.

If r_PriceInTicks is set to INT_MAX, this method will get the VAP element
with the highest price for the bar.
-----*/
template<typename t_VolumeAtPrice>
inline bool c_VAPContainerBase<t_VolumeAtPrice>::GetNextLowerVAPElement
( const unsigned int BarIndex
, int& r_PriceInTicks
, const t_VolumeAtPrice** p_VAP
) const
{
    *p_VAP = nullptr;

    unsigned int BeginIndex, EndIndex;
    if (!GetBeginEndIndexesForBarIndex(BarIndex, &BeginIndex, &EndIndex))
        return false;

    // Return the element for the highest price on the first iteration.
    if (r_PriceInTicks == INT_MAX)
    {
        t_VolumeAtPrice& r_HighestVAPElement
            = m_p_VAPDataElements[EndIndex - 1];

        *p_VAP = &r_HighestVAPElement;
        r_PriceInTicks = r_HighestVAPElement.PriceInTicks;

        return true;
    }

    // This is a temporary object only for the purpose of providing an object
    // of the same type for comparison.
    t_VolumeAtPrice Target;
    Target.PriceInTicks = r_PriceInTicks;

    const t_VolumeAtPrice* p_BeginElement = m_p_VAPDataElements + BeginIndex;
    const t_VolumeAtPrice* p_EndElement = m_p_VAPDataElements + EndIndex;

    // Note: lower_bound does a binary search (or something algorithmically
    // equivalent) and returns the lowest element that is greater than or
    // equal to the given Target. This means the next lower element will be
    // the element just before the returned element.
    const t_VolumeAtPrice* p_NextLowerElement
        = std::lower_bound(p_BeginElement, p_EndElement, Target, PricesLess);

    // Move the returned pointer down to the next lower element.
    --p_NextLowerElement;

    // Return false if we have reached the end of the data for the bar.
    if (p_NextLowerElement < p_BeginElement)
        return false;

```

```

    *p_VAP = p_NextLowerElement;
    r_PriceInTicks = p_NextLowerElement->PriceInTicks;

    return true;
}

/*=====*/
template<typename t_VolumeAtPrice>
inline bool c_VAPContainerBase<t_VolumeAtPrice>::GetHighAndLowPriceTicksForBarIndex
( const unsigned int BarIndex
, int& r_HighPriceInTicks
, int& r_LowPriceInTicks
) const
{
    return GetHighAndLowPriceTicksForBarIndex
        ( BarIndex
        , &r_HighPriceInTicks
        , &r_LowPriceInTicks
        );
}

/*=====*/
template<typename t_VolumeAtPrice>
inline bool c_VAPContainerBase<t_VolumeAtPrice>::GetHighAndLowPriceTicksForBarIndex
( const unsigned int BarIndex
, int* p_HighPriceInTicks
, int* p_LowPriceInTicks
) const
{
    unsigned int BeginIndex, EndIndex;
    if (!GetBeginEndIndexesForBarIndex(BarIndex, &BeginIndex, &EndIndex))
        return false;

    GetHighAndLowPriceTicks
        ( BeginIndex
        , EndIndex
        , p_HighPriceInTicks
        , p_LowPriceInTicks
        );

    return true;
}

/*=====*/
template<typename t_VolumeAtPrice>
inline bool c_VAPContainerBase<t_VolumeAtPrice>::GetHighAndLowPriceTicksForBarIndexRange
( const unsigned int FirstBarIndex
, const unsigned int LastBarIndex
, int* p_HighPriceInTicks
, int* p_LowPriceInTicks
) const
{
    unsigned int BeginIndex = GetFirstDataElementIndexForBar(FirstBarIndex);
    unsigned int EndIndex = GetFirstDataElementIndexForBar(LastBarIndex + 1);

    if (BeginIndex >= EndIndex)
        return false;

    GetHighAndLowPriceTicks
        ( BeginIndex
        , EndIndex
        , p_HighPriceInTicks
        , p_LowPriceInTicks
        );
}

```

```

    return true;
}

/*=====
Returns an element with all zeros if there is no element for the
requested BarIndex and PriceInTicks.
-----*/
template<typename t_VolumeAtPrice>
inline const t_VolumeAtPrice&
c_VAPContainerBase<t_VolumeAtPrice>::GetVAPElementAtPrice
( const unsigned int BarIndex
, const int PriceInTicks
) const
{
    static const t_VolumeAtPrice ZeroVAP;

    unsigned int BeginIndex, EndIndex;
    if (!GetBeginEndIndexesForBarIndex(BarIndex, &BeginIndex, &EndIndex))
        return ZeroVAP;

    // This is a temporary object only for the purpose of providing an object
    // of the same type for comparison.
    t_VolumeAtPrice Target;
    Target.PriceInTicks = PriceInTicks;

    t_VolumeAtPrice* p_BeginElement = m_p_VAPDataElements + BeginIndex;
    t_VolumeAtPrice* p_EndElement = m_p_VAPDataElements + EndIndex;

    t_VolumeAtPrice* p_FoundElement = std::lower_bound(p_BeginElement, p_EndElement, Target, PriceIsLess);

    if (p_FoundElement->PriceInTicks == PriceInTicks && p_FoundElement != p_EndElement)
    {
        return *p_FoundElement;
    }

    return ZeroVAP;
}

/*=====
Adds the given VolumeAtPrice to the current volume at the given BarIndex
and PriceInTicks. If no current volume exists, the volume will be added
to a new default element, if possible. Returns true unless the requested
element could not be accessed or created.
-----*/
template<typename t_VolumeAtPrice>
inline bool c_VAPContainerBase<t_VolumeAtPrice>::AddVolumeAtPrice
( const int PriceInTicks
, const unsigned int BarIndex
, const t_VolumeAtPrice& VolumeAtPrice
)
{
    t_VolumeAtPrice* p_VAPElement = nullptr;

    const bool ElementFound = GetVAPElement(PriceInTicks, BarIndex, &p_VAPElement, true);

    if (!ElementFound || p_VAPElement == nullptr)
        return false;

    *p_VAPElement += VolumeAtPrice;

    return true;
}

/*=====
Subtracts the given VolumeAtPrice from the current volume at the given

```

BarIndex and PriceInTicks. If no current volume exists, the volume will be subtracted from a new default element, if possible. Returns true unless the requested element could not be accessed or created.

```
-----*/
template<typename t_VolumeAtPrice>
inline bool c_VAPContainerBase<t_VolumeAtPrice>::SubtractVolumeAtPrice
( const int PriceInTicks
, const unsigned int BarIndex
, const t_VolumeAtPrice& VolumeAtPrice
)
{
    t_VolumeAtPrice* p_VAPElement = nullptr;

    const bool ElementFound
        = GetVAPElement(PriceInTicks, BarIndex, &p_VAPElement, true);

    if (!ElementFound || p_VAPElement == nullptr)
        return false;

    *p_VAPElement -= VolumeAtPrice;

    return true;
}
/*=====*/
template<typename t_VolumeAtPrice>
bool c_VAPContainerBase<t_VolumeAtPrice>::SetVolumeAtPrice(const int PriceInTicks, const unsigned int BarIndex,
const t_VolumeAtPrice& VolumeAtPrice)
{
    t_VolumeAtPrice* p_VAPElement = nullptr;

    const bool ElementFound
        = GetVAPElement(PriceInTicks, BarIndex, &p_VAPElement, true);

    if (!ElementFound || p_VAPElement == nullptr)
        return false;

    *p_VAPElement = VolumeAtPrice;

    return true;
}
/*=====
Returns true only if the price of the first element is less than the
price of the second element. This can be used as a comparison predicate
for sorting and searching.
-----*/
template<typename t_VolumeAtPrice>
inline bool c_VAPContainerBase<t_VolumeAtPrice>::PriceIsLess
( const t_VolumeAtPrice& First
, const t_VolumeAtPrice& Second
)
{
    return (First.PriceInTicks < Second.PriceInTicks);
}
/*=====
Checks if the given BarIndex is currently in the container. If
AllocatelfNeeded is true (default) and the bar does not exist, new bar
elements will be allocated and initialized so that the bar may exist.
Returns true unless the bar does not exist or could not be allocated.
-----*/
template<typename t_VolumeAtPrice>
inline bool c_VAPContainerBase<t_VolumeAtPrice>::EnsureBarExists
( const unsigned int BarIndex
, const bool AllocatelfNeeded
```

```

)
{
    // Make certain the arrays are initially allocated.
    if (!InitialAllocation())
        return false;

    // Return true if the given BarIndex is currently within this container.
    if (BarIndex < m_NumberOfBars)
        return true;

    // The given BarIndex is not currently within this container.

    // Return false if we cannot allocated the requested bar.
    if (!AllocateIfNeeded)
        return false;

    // Add the requested bar.

    const unsigned int PriorNumberOfBars = m_NumberOfBars;

    m_NumberOfBars = BarIndex + 1;

    const unsigned int NumberOfBarsAdded
        = m_NumberOfBars - PriorNumberOfBars;

    // Allocate space for more bars if necessary.
    if (!AllocateBar(BarIndex))
    {
        Clear();
        return false;
    }

    // Get the index that is just beyond the end of the VAP elements.
    const unsigned int VAPEndIndex
        = static_cast<unsigned int>
          ( (m_p_VAPDataElements + m_NumElementsUsed) - m_p_VAPDataElements
            );

    // Set the newly added bars to the end of the VAP data.
    for (unsigned int NewIndex = PriorNumberOfBars;
         NewIndex < m_NumberOfBars;
         ++NewIndex
        )
    {
        m_p_BarIndexToFirstElementIndexArray[NewIndex] = VAPEndIndex;
    }

    return true;
}

/*=====*/
template<typename t_VolumeAtPrice>
inline bool c_VAPContainerBase<t_VolumeAtPrice>::InitialAllocation()
{
    if (m_p_BarIndexToFirstElementIndexArray == nullptr)
    {
        m_NumBarsAllocated = m_InitialAllocationElements;

        m_p_BarIndexToFirstElementIndexArray
            = static_cast<unsigned int*>
              ( realloc
                ( nullptr
                  , m_NumBarsAllocated * sizeof(unsigned int)
                )
              );
    }
}

```



```

    if (m_p_BarIndexToFirstElementIndexArray == nullptr)
        return false;

    // Zero New elements.
    memset
    ( m_p_BarIndexToFirstElementIndexArray
    , 0
    , m_NumBarsAllocated * sizeof(unsigned int)
    );
}

if (m_p_VAPDataElements == nullptr)
{
    m_NumElementsAllocated = m_InitialAllocationElements*10;

    m_p_VAPDataElements
    = static_cast<t_VolumeAtPrice*>
    ( realloc
    ( nullptr
    , m_NumElementsAllocated * sizeof(t_VolumeAtPrice)
    )
    );

    if (m_p_VAPDataElements == nullptr)
        return false;

    // Zero New elements.
    memset
    ( m_p_VAPDataElements
    , 0
    , m_NumElementsAllocated * sizeof(t_VolumeAtPrice)
    );
}

return true;
}

/*=====
Makes any necessary allocations and initializations to ensure bar data at
the requested BarIndex is safe to access. Returns true unless the bar
could not be allocated.
-----*/
template<typename t_VolumeAtPrice>
inline bool c_VAPContainerBase<t_VolumeAtPrice>::AllocateBar
( const unsigned int BarIndex
)
{
    // Do nothing but return true if the requested bar is already allocated.
    if (BarIndex < m_NumBarsAllocated)
        return true;

    const int PriorNumBarsAllocated = m_NumBarsAllocated;

    m_NumBarsAllocated = m_NumBarsAllocated * 2;

    if (BarIndex >= m_NumBarsAllocated)
        m_NumBarsAllocated = m_NumberOfBars;

    // Note: If there is not enough available memory to expand the block to
    // the given size, the original block is left unchanged, and nullptr is
    // returned.
    void* p_NewBarIndexToFirstElementIndexArray
    = realloc
    ( m_p_BarIndexToFirstElementIndexArray

```

```

        , m_NumBarsAllocated * sizeof(unsigned int)
    );

    if (p_NewBarIndexToFirstElementIndexArray == nullptr)
        return false;

    m_p_BarIndexToFirstElementIndexArray
        = static_cast<unsigned int*>(p_NewBarIndexToFirstElementIndexArray);

    const int NumAllocatedBarsAdded
        = m_NumBarsAllocated - PriorNumBarsAllocated;

    // Initialize the new bar elements to zero.
    memset
        ( m_p_BarIndexToFirstElementIndexArray + PriorNumBarsAllocated
        , 0
        , NumAllocatedBarsAdded * sizeof(unsigned int)
        );

    return true;
}

/*=====
    Makes any necessary allocations and initializations to ensure VAP data at
    the requested ElementIndex is safe to access. Returns true unless the
    element could not be allocated.
    -----*/
template<typename t_VolumeAtPrice>
inline bool c_VAPContainerBase<t_VolumeAtPrice>::AllocateElement
( const unsigned int ElementIndex
)
{
    if (ElementIndex < m_NumElementsAllocated)
        return true;

    // The array needs to be increased in size.

    const int PriorNumElementsAllocated = m_NumElementsAllocated;

    m_NumElementsAllocated = static_cast<int>(m_NumElementsAllocated * 1.25);

    // Note: If there is not enough available memory to expand the block to
    // the given size, the original block is left unchanged, and nullptr is
    // returned.
    void* p_NewVAPDataElements
        = realloc
        ( m_p_VAPDataElements
        , m_NumElementsAllocated * sizeof(t_VolumeAtPrice)
        );

    if (p_NewVAPDataElements == nullptr)
        return false;

    m_p_VAPDataElements = static_cast<t_VolumeAtPrice*>(p_NewVAPDataElements);

    const int NumAllocatedElementsAdded = m_NumElementsAllocated - PriorNumElementsAllocated;

    // Initialize new elements to zero.
    memset
        ( m_p_VAPDataElements + PriorNumElementsAllocated
        , 0
        , NumAllocatedElementsAdded * sizeof(t_VolumeAtPrice)
        );

    return true;
}

```

```
}
```

```
/*=====
Returns the index of the first element in the VAP data array for the bar
at the given BarIndex. If the bar is not within the container, the 'end'
index, or the size of the VAP data array, is returned.
-----*/
```

```
template<typename t_VolumeAtPrice>
inline unsigned int c_VAPContainerBase<t_VolumeAtPrice>::GetFirstDataElementIndexForBar
( const unsigned int BarIndex
) const
{
    if (BarIndex >= m_NumberOfBars)
        return m_NumElementsUsed;

    if (m_p_BarIndexToFirstElementIndexArray == nullptr)
        return m_NumElementsUsed;

    return m_p_BarIndexToFirstElementIndexArray[BarIndex];
}
```

```
/*=====
Sets the given *p_BeginIndex and *p_EndIndex parameters to the indexes of
the first element and one index beyond the last element of VAP data for
the bar at the given BarIndex. *p_BeginIndex and *p_EndIndex are only
set when the function returns true, and the function only returns true if
there is data for the bar.
-----*/
```

```
template<typename t_VolumeAtPrice>
inline bool c_VAPContainerBase<t_VolumeAtPrice>::GetBeginEndIndexesForBarIndex
( const unsigned int BarIndex
, unsigned int* p_BeginIndex
, unsigned int* p_EndIndex
) const
{
    unsigned int BeginIndex = GetFirstDataElementIndexForBar(BarIndex);
    unsigned int EndIndex = GetFirstDataElementIndexForBar(BarIndex + 1);

    if (BeginIndex >= EndIndex)
        return false;

    if (p_BeginIndex != nullptr)
        *p_BeginIndex = BeginIndex;

    if (p_EndIndex != nullptr)
        *p_EndIndex = EndIndex;

    return true;
}
```

```
/*=====*/
```

```
template<typename t_VolumeAtPrice>
inline void c_VAPContainerBase<t_VolumeAtPrice>::GetHighAndLowPriceTicks
( const unsigned int BeginIndex
, unsigned int EndIndex
, int* p_HighPriceInTicks
, int* p_LowPriceInTicks
) const
{
    if (m_p_VAPDataElements == nullptr)
        return;

    if (p_HighPriceInTicks == nullptr && p_LowPriceInTicks == nullptr)
        return;
```

```

    if (BeginIndex == EndIndex)
        return;

    if (EndIndex > m_NumElementsUsed)
        EndIndex = m_NumElementsUsed;

    if (p_HighPriceInTicks != nullptr)
        *p_HighPriceInTicks = m_p_VAPDataElements[EndIndex - 1].PriceInTicks;

    if (p_LowPriceInTicks != nullptr)
        *p_LowPriceInTicks = m_p_VAPDataElements[BeginIndex].PriceInTicks;
}

/*****
// c_VAPContainer

class c_VAPContainer
: public c_VAPContainerBase<s_VolumeAtPriceV2>
{
    ///-- Public Methods -----
public:
    c_VAPContainer(const unsigned int InitialAllocationElements = 1024);

    unsigned int GetVolumeAtPrice
        ( const unsigned int BarIndex
        , const int PriceInTicks
        ) const;

    unsigned int GetBidVolumeAtPrice
        ( const unsigned int BarIndex
        , const int PriceInTicks
        ) const;

    unsigned int GetAskVolumeAtPrice
        ( const unsigned int BarIndex
        , const int PriceInTicks
        ) const;
};

/*=====*/
inline c_VAPContainer::c_VAPContainer
( const unsigned int InitialAllocationElements
)
: c_VAPContainerBase(InitialAllocationElements)
{
}

/*=====
Returns 0 if there is no element for the requested BarIndex and
PriceInTicks.
-----*/
inline unsigned int c_VAPContainer::GetVolumeAtPrice
( const unsigned int BarIndex
, const int PriceInTicks
) const
{
    return GetVAPElementAtPrice(BarIndex, PriceInTicks).Volume;
}

/*=====
Returns 0 if there is no element for the requested BarIndex and
PriceInTicks.
-----*/
inline unsigned int c_VAPContainer::GetBidVolumeAtPrice
( const unsigned int BarIndex

```

```

, const int PriceInTicks
) const
{
    return GetVAPElementAtPrice(BarIndex, PriceInTicks).BidVolume;
}

/*=====
Returns 0 if there is no element for the requested BarIndex and
PriceInTicks.
-----*/
inline unsigned int c_VAPContainer::GetAskVolumeAtPrice
( const unsigned int BarIndex
, const int PriceInTicks
) const
{
    return GetVAPElementAtPrice(BarIndex, PriceInTicks).AskVolume;
}

/*****
// c_VolumeLevelAtPriceContainer

class c_VolumeLevelAtPriceContainer
: public c_VAPContainerBase<s_VolumeLevelAtPrice>
{
    //-- Public Methods -----
public:
    c_VolumeLevelAtPriceContainer
    ( const unsigned int InitialAllocationElements = 1024
    );

    unsigned int GetMaxVolumeAtPrice
    ( const unsigned int BarIndex
    , const int PriceInTicks
    ) const;

    unsigned int GetTotalVolumeAtPrice
    ( const unsigned int BarIndex
    , const int PriceInTicks
    ) const;
};

/*=====*/
inline c_VolumeLevelAtPriceContainer::c_VolumeLevelAtPriceContainer
( const unsigned int InitialAllocationElements
)
: c_VAPContainerBase(InitialAllocationElements)
{
}

/*=====
Returns 0 if there is no element for the requested BarIndex and
PriceInTicks.
-----*/
inline unsigned int c_VolumeLevelAtPriceContainer::GetMaxVolumeAtPrice
( const unsigned int BarIndex
, const int PriceInTicks
) const
{
    return GetVAPElementAtPrice(BarIndex, PriceInTicks).MaxVolume;
}

/*=====
Returns 0 if there is no element for the requested BarIndex and
PriceInTicks.
-----*/

```

```
inline unsigned int c_VolumeLevelAtPriceContainer::GetTotalVolumeAtPrice
( const unsigned int BarIndex
, const int PriceInTicks
) const
{
    return GetVAPElementAtPrice(BarIndex, PriceInTicks).TotalVolume;
}
```

```
/*=====*/
```