TBaseArray: Alphabetic list of methods

Methods listed by category

Append	Append new elements to an array
AssignTo	Copy all the data to another array object
BlockCopy	Copy elements from another array
ClearFlag	Clear an array flag
Clone	Make a copy of this oject and its data
CopyFrom	Copy several elements from a passed variable
CopyTo	Copy several elements to a passed variable
Create	Constructor to set array and element size
DefineProperties	Define data items to a filer object
Delete	Delete elements from an array
Destroy	Destructor
Find	Search array for an element
FirstThat	Iterate over the array using an object method to find the first element matching a condition
FirstThatProc	Iterate over the array using a non-object function to find the first element matching a condition
ForEach	Iterate over all elements using an object method
ForEachProc	Iterate over all elements using a non-object procedure
GetCapacity	Get the number of elements the array can hold
GetCount	Get the number of used slots in the array
GetItem	Get data from one element
GetItemPtr	Obtain pointer to an array element
GetMaxCapacity	Get the maximal number of elements possible for an array of this class
HasFlag	Test for an array flag
Insert	Insert new elements into an array
InvalidateItems	Cleanup after deleted elements
LastThat	Iterate over the array using an object method to find the last element matching a condition
LastThatProc	Iterate over the array using a non-object function to find the last element matching a condition
Load	Load the objects data from a filer object
LoadFromFile	Load the array elements from a file
LoadFromStream	Load the array elements from a stream
PutItem	Copy data to one element
ReDim	Resize the array
SaveToFile	Save the array elements to a file
SaveToStream	Save the array elements to a stream
SetCompareProc	Define the comparision function
SetFlag	Set an array flag
Store	Store the objects data to a filer object
Sort	Sort the array
ValidIndex	Check index value
ValidateBounds	Check index and an element count
Zap	Delete all elements in the array and fill it with 0

AsInteger[Index: Cardinal]: LongInt **(Property)**

See Also

Class TPCharArray, TPStringArray
Visibility: public
Access: read/write

Description:

This is a conversion property; on write access it will convert the assigned number to a string representation using IntToString and store a pointer to that string in the *index-th* item of the array. On read access it will return the number stored in the string at *index*, or 0, if the string does not contain a valid integer number.

The property is implemented via the GetAsInteger and PutAsInteger methods.

AsReal[Index: Cardinal]: Extended **(Property)**

See Also

Class TPCharArray, TPStringArray
Visibility: public
Access: read/write

Description:

This is a conversion property; on write access it will convert the assigned number to a string representation using FloatToStr and store a pointer to that string in the *index-th* item of the array. On read access it will return the number stored in the string at *index*, or 0, if the string does not contain a valid real number.

The property is implemented via the GetAsReal and PutAsReal methods.

CBaseArray = Class of TBaseArray;

This is a class reference type defined in Unit Arrays.

Hierarchy of the Array Classes

Function **Cloneitem**(item: Pointer): Pointer; **(Method)**

See Also

Classes: TPointerArray, TPStringArray, TPCharArray
Visibility: public
Directives: virtual in TPointerArray, override in derived classes

Parameters:

Item Pointer to a data item to clone.

Return: Pointer to the cloned data item or *item*, depending on the state of the AF_OwnsData flag and the methods implementation.

Description:

This method is provided in TPointerArray and all derived classes to implement deep copy of data items. The method of TPointerArray just returns the *Item* pointer, as will the overriden methods in TPStringArray and TPCharArray, if the AF_OwnsData flag is not set. If this flag is set, however, the methods of the string array classes will make a copy of the passed string or Pchar on the heap, using StrNew or NewStr, and return the pointer to this copy.

Error Conditions:

May cause an EOutOfMemory exception, if a dynamic copy of the passed item is made and the heap manager runs out of memory.

Function **CmpCardinals**(Var item1, item2): Integer;

See also

Unit: Arrays

Parameters:

item1, item2 the two numbers to compare

Returns: < 0, if item1 < item2, > 0 if item1 > item2, 0 if item1 = item2

Description:

This is one of the comparison functions supplied by the Arrays Unit. It is used by the TCardinalArray class to implement the Sort and Find methods. The return is *not* limited to -1, 0, and +1!

Error Conditions:

none

Function **CmpDoubles**(Var item1, item2): Integer;

See also

Unit: Arrays

Parameters:

item1, item2 the two numbers to compare

Returns: < 0, if item1 < item2, > 0 if item1 > item2, 0 if item1 = item2

Description:

This is one of the comparison functions supplied by the Arrays Unit. It is used by the TDoubleArray class to implement the Sort and Find methods. The return is *not* limited to -1, 0, and +1!

Error Conditions:

none

Function **CmpExtendeds**(Var item1, item2): Integer;

See also

Unit: Arrays

Parameters:

item1, item2 the two numbers to compare

Returns: < 0, if item1 < item2, > 0 if item1 > item2, 0 if item1 = item2

Description:

This is one of the comparison functions supplied by the Arrays Unit. It is used by the TExtendedArray class to implement the Sort and Find methods. The return is *not* limited to -1, 0, and +1!

Error Conditions:

none

Function **CmpIntegers**(Var item1, item2): Integer;

See also

Unit: Arrays

Parameters:

item1, item2 the two numbers to compare

Returns: < 0, if item1 < item2, > 0 if item1 > item2, 0 if item1 = item2

Description:

This is one of the comparison functions supplied by the Arrays Unit. It is used by the TIntegerArray class to implement the Sort and Find methods. The return is *not* limited to -1, 0, and +1!

Error Conditions:

none

Function **CmpLongs**(Var item1, item2): Integer;

See also

Unit: Arrays

Parameters:

item1, item2 the two numbers to compare

Returns: < 0, if item1 < item2, > 0 if item1 > item2, 0 if item1 = item2

Description:

This is one of the comparison functions supplied by the Arrays Unit. It is used by the TLongIntIArray class to implement the Sort and Find methods. The return is *not* limited to -1, 0, and +1!

Error Conditions:

none

Function **CmpPChars**(Var item1, item2): Integer;

See also

Unit: Arrays

Parameters:

item1, item2 pointers (PChars) to the two zero-terminated strings to compare

Returns: < 0, if item1 < item2, > 0 if item1 > item2, 0 if item1 = item2

Description:

This is one of the comparison functions supplied by the Arrays Unit. It is used by the TPCharArray class to implement the Sort and Find methods. The function uses the lstrcmp function from the Windows API, is case sensitive and honors the character collating sequence defined by the active language driver. The return is *not* limited to -1, 0, and +1!

Error Conditions:

none

Function **CmpPStrings**(Var item1, item2): Integer;

See also

Unit: Arrays

Parameters:

item1, item2 pointers to the two Pascal strings to compare

Returns: < 0, if item1 < item2, > 0 if item1 > item2, 0 if item1 = item2

Description:

This is one of the comparison functions supplied by the Arrays Unit. It is used by the TPStringArray class to implement the Sort and Find methods. The function uses the AnsiCompareStr function from the **SysUtils** Unit, is case sensitive and honors the character collating sequence defined by the active language driver. The return is *not* limited to -1, 0, and +1!

Error Conditions:

none

Function **CmpReals**(Var item1, item2): Integer;

See also

Unit: Arrays

Parameters:

item1, item2 the two numbers to compare

Returns: < 0, if item1 < item2, > 0 if item1 > item2, 0 if item1 = item2

Description:

This is one of the comparison functions supplied by the Arrays Unit. It is used by the TRealArray class to implement the Sort and Find methods. The return is *not* limited to -1, 0, and +1!

Error Conditions:

none

Function **CmpSingles**(Var item1, item2): Integer;

See also

Unit: Arrays

Parameters:

item1, item2 the two numbers to compare

Returns: < 0, if item1 < item2, > 0 if item1 > item2, 0 if item1 = item2

Description:

This is one of the comparison functions supplied by the Arrays Unit. It is used by the TSingleArray class to implement the Sort and Find methods. The return is *not* limited to -1, 0, and +1!

Error Conditions:

none

Comparing, Sorting, and Searching Array Items

The TBaseArray class has a generic mechanism to sort itself and to search for a specific item. To be able to do this you, the programmer, needs to supply a **comparison function** for the array to use. This function must adhere to the prototype TCompareProc.

You set and retrieve the comparison function using CompareProc property, the function pointer is stored in the FCompareProc field, which is private.

The classes derived for specific numeric types from TBaseArray already have a predefined comparison function but you need to explicitely set one for any class you derive from TBaseArray (which does not have such a function). This is best be done in the classes constructor. Failure to set a comparison function will result in an ECompUndefined exception when you try to call the Sort or Find methods!

The Unit Arrays exports a number of comparison functions for numeric types:

You sort an array by calling its Sort method, specifying a sort order. The sort order is stored in the FSortOrder field and can be accessed via the SortOrder property. Inserting data into the array will set the sort order to unsorted (TS_NONE). If you derive your own classes from TBaseArray better adhere to this convention or Sort and Find may act in unforseen ways!

You search for an item in an array by calling its Find method. The method wil use binary search, if the array is sorted, and sequential search if it is unsorted. Thus it is important that the SortOrder property correctly reflects the sorted status of the array!

Peter's Delphi Tools

Fast memory manipulation routines

Unit FastMem

Dynamic arrays

Unit Arrays

The units in this collection will compile with Delphi 1.0 or Delphi 2.0. They have not been exhaustively tested in Delphi 2.0, however!

Constructor **Create**(itemcount, itemsize: Cardinal);

Class	all classes derived from TBaseArray
Visibility:	public
Directives:	override

Parameters:

itemcount	number of items the array should hold, 0 is mapped to 1 since the array size cannot be 0!
itemsize	ignored

Description:

Allocates the memory for the array and sets the fields according to the passed data. In the Win16 version the product of itemcount and base type size has to be < 64Kbyte. All items in the array are set to 0. We reduce the itemcount to an allowed value, if necessary, without raising any error if it is to large.

Create calls the inherited constructor with the proper item size and also installs a comparison function for the numeric array types. it also defines the initial set of array flags.

Error Conditions:

If GetMem fails we rely on the default exception handling to fail the constructor via an EOutOfMemory exception.

Deriving your own array classes

Lets assume you have a record type defined as

```
Type
  TNameString = string[30];
  TPersonnel= Record
               family_name: TNameString;
               first_name : TNameString;
               middle_name: TNameString;
               .... more fields here ....
              End;
  PPersonnel = ^TPersonnel;
```

Now you want to create a dynamic array class to hold an array of these records. There are only a few modifications we need to make for this descendant of TBaseArray. We need a new Create constructor and a property to access the data. This property needs two methods for its implementation. By convention the property should be named Data and the access methods GetData and PutData.

```
  TPersonnelArray = Class( TBaseArray )
    public
      Constructor Create( itemcount, dummy: Cardinal ); override;
      Procedure PutData( index: Cardinal; value: TPersonnel );
      Function GetData(index: Cardinal): TPersonnel;
      Function GetAsPtr(index: Cardinal): PPersonnel;

      property Data[ Index:Cardinal ]: TPersonnel
         read GetData write PutData; default;
      property AsPtr[ Index: Cardinal ]: PPersonnel
         read GetAsPtr;
  End; { TPersonnelArray }
```

The **Data** property allows us to access the array class with the usual array notation, but there is a subtle difference to be aware of: you cannot modify a single field of an array item via the **Data** property like you can for a normal array. The reason is that the property returns a *copy* of an array item and can also only *set* an item as a whole (using the property invokes method calls). Thats why i also defined the AsPtr property; it returns the actual address of an array item and can thus be used to modify the actual item data. It is also much faster for reading array item fields.

Ok, lets take a look at the implementation.

```
{+--------------------------
 | Methods of TPersonnelArray
 +--------------------------}
Type
  TPArray =Array[ 0..High( Cardinal ) div Sizeof( TPersonnel )-1 ] of
TPersonnel;
  PPArray = ^TPArray;
{ we use this dummy array type to fool the compiler in doing all the
   address calculations for item access for us }
Constructor TPersonnelArray.Create( itemcount, itemsize: Cardinal );
  Begin
```

```
    inherited Create( itemcount, Sizeof( TPersonnel ));
  End; { TIntegerArray.Create }

{ The constructor overrides the virtual Create constructor of the base
  class. The constructor needs to be virtual for the Clone method of
  TBaseArray to work for all descendants. All derived classes do not use
  the itemsize parameter but it has to be there for compatibility with the
  overridden constructor. }

Procedure TPersonnelArray.PutData( index: Cardinal ; value: TPersonnel );
  Begin
{$IFOPT R+}
    If ValidIndex( index ) Then
{$ENDIF}
      PPArray( Memory )^[ index ] := value;
  End; { TPersonnelArray.PutData }

{ Note how we use a typecast to trick the compiler into seeing our buffer
  pointer as a pointer to an array of TPersonnel. If range checking is
  enabled we test the index for validity using a method inherited from
  TBaseArray. ValidIndex will raise an ERangeError exception if the index
  is out of bounds. We could call ValidIndex unconditional and be on the
  safe side, since it returns False if the index is out of bounds, but that
  would significantly reduce the speed of access due to the method call
  overhead. }

Function TPersonnelArray.GetData(index: Cardinal): TPersonnel;
  Begin
{$IFOPT R+}
    If ValidIndex( index ) Then
{$ENDIF}
      Result := PPArray( Memory )^[ index ];
  End; { TPersonnelArray.GetData }

{ GetData uses the same principle as PutData }


Function TPersonnelArray.GetAsPtr(index: Cardinal): PPersonnel;
  Begin
{$IFOPT R+}
    If ValidIndex( index ) Then
{$ENDIF}
      Result := @PPArray( Memory )^[ index ];
  End; { TPersonnelArray.GetAsPtr }
```

You may ask why i choose to implement GetData and PutData the way i did and not fell back on methods of TBaseArray like CopyTo and CopyFrom or GetItem and PutItem? The answer is simple: speed! The dynamic array classes already suffer a performance penalty compared to genuine (static) arrays because item access via properties involves a function call here. Using the mentioned methods would have added further overhead due to more function calls and sanity checks. There is a price to pay for speed here, of course: safety. If range checking is disabled we do no validation checks on the passed index values; if they are beyond the upper bound of the array a protection fault will be the likely consequence! So take care your index stays in bounds. That should never be a problems if you use loops like

```
With aPersArr Do
  For i:= 0 To MaxIndex Do Begin
    ....
```

Ok, now for a simple example of a use for the TPersonnelArray class. The following code Uses WinCRT for output.

```
Var
    PersArr: TPersonnelArray;
    Pers   : TPersonnel;
Begin
    PersArr := TPersonnelArray.Create( 10, 0 );
{ create space for 10 TPersonnel records }
    try
{ assigning values to array items with code like
     PersArr[0].family_name := 'Below';
   is rejected by the compiler, for good reason. We have two alternatives to
   set values for array items, the first uses a temporary variable: }
    Pers.family_name := 'Below';
    Pers.first_name  := 'Peter';
    Pers.middle_name := 'E. A.';
    PersArr[0] := Pers;
    WriteLn('Personnel Info');
{ note that reading item fields is no problem. Always use a With construct
   to do it, or the property will be called for every field, which is rather
   unefficient! }
    With PersArr[0] Do Begin
      WriteLn('Family Name: ', family_name);
      WriteLn('First Name : ', first_name);
      WriteLn('Middle Name: ', middle_name );
    End;
{ the second method uses a pointer to the actual array item data and is
   both faster and more memory efficient (no temp variable needed). }
    With PersArr.AsPtr[1]^ Do Begin
      family_name := 'Wolpertinger';
      first_name  := 'Gerrit';
      middle_name := 'E. T. H.';
    End;
    With PersArr[1] Do Begin
      WriteLn('Family Name: ', family_name);
      WriteLn('First Name : ', first_name);
      WriteLn('Middle Name: ', middle_name );
    End;
    finally
      PersArr.Free;
    end;
End.
```

Type

ECompUndefined = Class(Exception);

This error is raised when TBaseArray.Sort or TBaseArray.Find are called and a compare proc has not been assigned

Type

EFileTooLarge = Class(Exception);

This error is raised by one of the string array types (TPCharArray, TPStringArray) if their LoadFromTextfile method cannot load a file due to low memory or because it has more than 16K lines.

Type

ETypeMismatch = Class(Exception);

This error is raised when two instances of dynamic arrays used in an operation are not of the same class and component size. Methods that can raise this exception are TBaseArray.AssignTo (and the inherited **Assign** method calling AssignTo), TBaseArray.LoadFromStream, and TBaseArray.BlockCopy.

Unit Arrays exports the following comparison functions:

- CmpIntegers
- CmpCardinals
- CmpLongs
- CmpReals
- CmpSingles
- CmpDoubles
- CmpExtendeds
- CmpPChars
- CmpPStrings

MemFill
MemWordFill
MemDWordFill
MemMove
MemSwap

Procedure **Freeitem**(item: Pointer); **(Method)**

See Also

Classes: TPointerArray, TPStringArray, TPCharArray
Visibility: public
Directives: virtual in TPointerArray, override in derived classes

Parameters:

Item Pointer to a data item to free.

Description:

This method is provided in TPointerArray and all derived classes to implement proper release of memory allocated for data items. The method of TPointerArray does nothing, as will the overriden methods in TPStringArray and TPCharArray, if the AF_OwnsData flag is not set. If this flag is set, however, the methods of the string array classes will assume, that the item was allocated on the heap and use StrDispose or DisposeStr to free the memory.

FreeItem is called by InvalidateItems and should not be called directly.

Error Conditions:

May cause a fault if the *Item* was not allocated on the heap or has already been freed..

Function **GetAsInteger**(index: Cardinal): LongInt; **(Method)**

See Also

Class TPStringArray, TPCharArray
Visibility: public
Directives: none (static)

Parameters:

Index index of the array element to get, must be in the range 0..MaxIndex.

Returns: the integer number represented by the string stored at *index*, or 0, if the string does not contain a valid number.

Description:

This method is used to implement read accesss via the AsInteger property. You should not call it directly.

Errors:

none

Function **GetAsReal**(index: Cardinal): Extended; **(Method)**

See Also

Class TPStringArray, TPCharArray
Visibility: public
Directives: none (static)

Parameters:

Index index of the array element to get, must be in the range 0..MaxIndex.

Returns: the real number represented by the string stored at *index*, or 0.0, if the string does not contain a valid number.

Description:

This method is used to implement read accesss via the AsReal property. You should not call it directly.

Errors:

none

Function **GetData**(index: Cardinal): some type; **(Method)**

See Also

Class all classes derived from TBaseArray
Visibility: public
Directives: none (static)

Parameters:

Index index of the array element to get, must be in the range 0..MaxIndex.

Returns: the value of the array element. The actual return type matches the base type of the array class, e.g. Integer for TIntegerArray.

Description:

This method is used to implements read accesss to array elements via the Data property, which is the default property for all classes with the exception of TPointerArray..

Initial Array Flag Values

The array flags are initialized to default values by each array classes Create constructor. The following table lists the default values. See the topic "The Array Flags" for their meaning.

Class	AF_OwnsData	AF_AutoSize	AF_CanCompare
TBaseArray	X	X	
TIntegerArray	X	X	X
TCardinalArray	X	X	X
TLongIntArray	X	X	X
TRealArray	X	X	X
TSingleArray	X	X	X
TDoubleArray	X	X	X
TExtendedArray	X	X	X
TPointerArray		X	
TPCharArray	X	X	X
TPStringArray	X	X	X

ForEach	ForEachProc
FirstThat	FirstThatProc
LastThat	LastThatProc

Procedure **LoadFromTextFile**(Const Filename: String; appendData: Boolean; reporter: TProgressReporter); **(Method)**

See Also

Class: TPCharArray, TPStringArray
Visibility: public
Directives: none (static)

Parameters:

Filename name of the file to read

appendData true if the lines from file are to be appended to the array, false if the previous contents of the array are to be discarded.

reporter optional pointer to a function to call after each line read to report progress to the user, can be Nil.

Description:

Loads the contents of the requested file line by line into the array (storage for each line is allocated from the heap, the lines must be terminated by a carriage return/line feed pair), which is redimensioned to fit the data. If *appendData* is false the array is first cleared, otherwise the new lines are appended to the array. After each line has been read the *reporter* function (if <> Nil) is called, passing the current position in the file and its total size. The reporter function can use this information to display a progress report to the user. If the function returns false, further reading of the file is aborted. The value returned in the *retain* parameter of the reporter function determines, whether the lines already read are kept or discarded.

Error Conditions:

May raise a EInOutError exception if a file-related error occurs. Will raise a EFileToLarge exception if the file has more lines than can fit into the array (which is limited to 16 K lines in the Win16 version). The already read lines are always kept if this exception occurs.

Procedure **LoadItemFromStream**(S: TStream; Var Item: Pointer); **(Method)**

See Also

Classes: TPointerArray, TPStringArray, TPCharArray
Visibility: public
Directives: virtual in TPointerArray, override in derived classes

Parameters:

S	Stream to load data from, must be open.
Item	Returns a pointer to the loaded item.

Description:

This method is provided in TPointerArray and all derived classes to implement proper loading of data items from a stream. The method of TPointerArray does nothing, the overriden methods in TPStringArray and TPCharArray will read a length value from the stream, followed by as many characters as the length value dictates. Note that streams written by a TPStringArray object are not compatible with those written by a TPCharArray, since the first uses a length byte while the second uses a length Cardinal! If you need to transfer data between those two object types via streams/files, use the SaveToTextfile and LoadFromTextfile methods, which produce standard ANSI files.

LoadItemFromStream is called by LoadFromStream and should not be called directly.

Error Conditions:

May cause a read fault in the stream or an EOutOfMemory exception while making a copy of the read string on the heap.

Procedure **MemDWordFill**(pTarget: Pointer; numDWords: Cardinal; value: LongInt);

See Also:

Unit:

FastMem

Parameters:

pTarget	pointer to memory to fill
numDWords	number of dwords to fill
value	dword value to fill with

Description:

Fills the memory pointed to by *pTarget* with *numDWord* copies of *value*. This overwrites 4**numDWords* bytes!

Error Conditions:

May generate a GPF if the memory area addressed by *pTarget* cannot take *numDWords* dwords!

Procedure **MemFill**(pTarget: Pointer; numBytes: Cardinal; value: Byte);

See Also:

Unit:

FastMem

Parameters:

pTarget	pointer to memory to fill
numBytes	number of bytes to fill
value	byte value to fill with

Description:

Like System.FillChar, only faster, since it fills in the largest possible unit (word or dword) as far as possible.

Error Conditions:

May generate a GPF if the memory area addressed by *pTarget* cannot take *numBytes* bytes!

Procedure **MemMove**(pSource, pTarget: Pointer; numBytes: Cardinal);

See Also:

Unit:

FastMem

Parameters:

pSource	pointer to memory to copy from
pTarget	pointer to memory to copy to
numBytes	number of bytes to copy

Description:

Like System.Move, only faster for larger numbers of bytes, since it does the copy word or dword-wise, as far as possible.The procedure checks for overlap of source and target regions and performs the copy from highest address backwards, if the regions overlap in a problematic way. The logic is optimized for the source address beeing even (data word or dword-aligned).

Error Conditions:

May cause a GPF if the memory addressed by the pointers has a size of less than *numBytes* bytes.

Procedure **MemSwap**(pSource, pTarget: Pointer; numBytes: Cardinal);

See Also:

Unit:

FastMem

Parameters:

pSource	pointer to first memory area
pTarget	pointer to second memory area
numBytes	number of bytes to swap

Description:

exchanges the contents of the memory addressed by the two pointers. These areas should never overlap or the results will invariably be somewhat strange!

Error Conditions:

May cause a GPF if the memory addressed by the pointers has a size of less than *numBytes* bytes.

Procedure **MemWordFill**(pTarget: Pointer; numWords: Cardinal; value: Word);

See Also:

Unit:

FastMem

Parameters:

pTarget	pointer to memory to fill
numWords	number of words to fill
value	word value to fill with

Description:

Fills the memory pointed to by *pTarget* with *numWord* copies of *value*. This overwrites 2**numWords* bytes!

Error Conditions:

May generate a GPF if the memory area addressed by *pTarget* cannot take *numWords* words!

Methods inherited from TPointerArray
GetData
PutData
CloneItem
FreeItem
SaveToTextFile
LoadFromTextFile
SaveitemToStream
LoadItemFromStream
PutAsString
GetAsString
PutAsInteger
GetAsInteger
PutAsReal
GetAsReal

Methods inherited from TPointerArray
GetData
PutData
CloneItem
FreeItem
SaveToTextFile
LoadFromTextFile
SaveitemToStream
LoadItemFromStream
PutAsPChar
GetAsPChar
GetAsPString
PutAsInteger
GetAsInteger
PutAsReal
GetAsReal

Methods inherited from TBaseArray
GetData
PutData
CopyFrom
CopyTo
InvalidateItems
CloneItem
FreeItem
SaveToFile
LoadFromFile
SaveToStream
LoadFromStream
SaveitemToStream
LoadItemFromStream

inherited methods
GetData
PutData

Const
    NOT_FOUND = High(Cardinal);

This value is returned by the TBaseArray.Find method if the passed value could not be found in the array.

inherited properties
Data
AsString
AsInteger
AsReal
AsPtr

inherited properties
Data
AsPChar
AsPString
AsInteger
AsReal
AsPtr

inherited properties
Data
AsPtr

inherited properties
Data

Data[Index: Cardinal]: some type **(Property)**

See Also

Class all classes derived from TBaseArray
Visibility: public, default
Access: read/write

Description:

This property provides the array-style access to data elements of all the specific array classes derived from TBaseArray. It is the default property (exception: TPointerArray), so you can just add an index value in square brackets to an array object to access an element. The return type matches the base type of each array class, e.g. Integer for TIntegerArray. See the topic Deriving your own array classes for some of the pitfalls to be aware of in using a property to access array elements.

The property is implemented via the GetData and PutData methods also present in all descendants of the array base class.

Procedure **PutAsInteger**(index: Cardinal; value: LongInt); **(Method)**

See Also

Class TPStringArray, TPCharArray
Visibility: public
Directives: none (static)

Parameters:

Index index of the array element to set, must be in the range 0..MaxIndex.

value the integer number to store as string.

Description:

The method converts the passed number to a string representation, using IntToStr, and stores a pointer to a copy of the string made on the heap at *index*. If the array flag **AF_OwnsData** is set (the default) the previous string stored at that location will be freed automatically.

This method is used to implement write accesss via the AsInteger property. You should not call it directly.

Errors:

none

Procedure **PutAsReal**(index: Cardinal; value: Extended); **(Method)**

See Also

Class TPStringArray, TPCharArray
Visibility: public
Directives: none (static)

Parameters:

Index index of the array element to set, must be in the range 0..MaxIndex.

value the real number to store as string.

Description:

The method converts the passed number to a string representation, using FloatToStr, and stores a pointer to a copy of the string made on the heap at *index*. If the array flag **AF_OwnsData** is set (the default) the previous string stored at that location will be freed automatically.

This method is used to implement write accesss via the AsReal property. You should not call it directly.

Errors:

none

Function **PutData**(index: Cardinal; value: some type); **(Method)**

See Also

Class all classes derived from TBaseArray
Visibility: public
Directives: none (static)

Parameters:

Index index of the array element to set, must be in the range 0..MaxIndex.

value the value to copy into that array elements. The actual type matches the base type of the array class, e.g. Integer for TIntegerArray.

Description:

This method is used to implements write accesss to array elements via the Data property, which is the default property for all derived classes with the exception of .TPointerArray.

Procedure **SaveItemToStream**(S: TStream; Item: Pointer); **(Method)**

See Also

Classes: TPointerArray, TPStringArray, TPCharArray
Visibility: public
Directives: virtual in TPointerArray, override in derived classes

Parameters:

S Stream to write data to, must be open.

Item Pointer to the item to write.

Description:

This method is provided in TPointerArray and all derived classes to implement proper writing of data items to a stream. The method of TPointerArray does nothing, the overriden methods in TPStringArray and TPCharArray will write a length value to the stream, followed by as many characters as the length value dictates. Note that streams written by a TPStringArray object are not compatible with those written by a TPCharArray, since the first uses a length byte while the second uses a length Cardinal! If you need to transfer data between those two object types via streams/files, use the SaveToTextfile and LoadFromTextfile methods, which produce standard ANSI files.

SaveItemToStream is called by SaveToStream and should not be called directly.

Error Conditions:

May cause a write fault in the stream.

Procedure **SaveToTextFile**(Const Filename: String; appendData: Boolean; reporter: TProgressReporter); **(Method)**

See Also

Class: TPCharArray, TPStringArray
Visibility: public
Directives: none (static)

Parameters:

Filename	name of the file to write to
appendData	true if the lines from the array are to be appended to the file, false if the previous contents of the file are to be discarded.
reporter	optional pointer to a function to call after each line written to report progress to the user, can be Nil.

Description:

Writes the contents of the array line by line to the requested file, terminating each with a carriage return/line feed pair. If *appendData* is false and the file already exists it will be overwritten, otherwise the new lines are appended to the file. After each line has been written the *reporter* function (if <> Nil) is called, passing the current position in the file and its expected final size. The reporter function can use this information to display a progress report to the user. If the function returns false, further writing of the file is aborted. The value returned in the *retain* parameter of the reporter function determines, whether the partial file is kept or deleted.

Error Conditions:

May raise a EInOutError exception if a file-related error occurs.

ECompUndefined
ETypeMismatch
EFileTooLarge

Type

TArrayFlags = (AF_OwnsData, AF_AutoSize, AF_CanCompare, AF_User1, AF_User2, AF_User3, AF_User4, AF_User5, AF_User6, AF_User7, AF_User8, AF_User9, AF_User10, AF_User11, AF_User12);

This enumerated type is used to define some basic features of a dynamic arrays behaviour. All array classes inherit the FFlags field from TBaseArray, which is a set of TArrayFlags. Currently only the first three flag values are used. See "The Array Flags" for a discussion of the effects of the various flags.

Type

TArrayFlagSet = Set of TArrayFlags;

This is the type used by the FFlags field of TBaseArray to hold the array flags.

TBaseArray (Class)

See Also Fields Methods Properties Exceptions

Unit: Arrays
Ancestor: TPersistent

Description:

TBaseArray is the base class of all dynamic, resizeable array classes in the Arrays Unit. Instances of this class can be used as such but if you want to use an instance with the normal array notation you have to derive a class for your specific type and give it a default property with the right type declaration.

When you create an instance of this class you pass the number of elements you need and the size of an element (size of the base type of the array) to the Create constructor. You can change the number of elements later with the Redim method but the element size cannot be changed. The object stores the array elements in a single block of memory obtained with GetMem and resized with ReallocMem, under Win16 (Delphi 1.0) the total size of the array is thus limited to 64KByte. This limitation does not apply under Win32 (Delphi 2.0). You can obtain a pointer to the array memory via the Memory property, get the allocated size via MemSize, the number of elements with Capacity and the highest valid index with MaxIndex (the first element has the index 0). MaxCapacity gives you the maximal number of elements that will fit into 64 KBytes (for Delphi 1.0, in Delphi 2.0 MaxCapacity returns the number of elements that will fit into the largest arrays the can theoretically be allocated).

Individual elements can be accessed directly via the ItemPtr property but you should use the GetItem and PutItem methods to access individual elements and CopyTo, CopyFrom and BlockCopy to manipulate ranges of elements. To make a copy of the whole object, including data, use the Clone method, to overwrite the data use the Assign method inherited from TPersistent or call AssignTo directly. Elements can also be removed (Delete), inserted (Insert), and appended (Append), which can optionally change the size of the array (depends on AF_AutoSize, one of the array flags)..

The array object has methods to be written to a file or stream (SaveToFile, SaveToStream) or be read from one (LoadFromFile, LoadFromStream).

The array objects have a number of functionalities i found very useful in Borland Pascals collections, too. They can be sorted (Sort) and searched for an element (Find), they even have iterator methods like ForEach, FirstThat and LastThat. These methods are designed to work with object methods instead of local procedures or functions like in BP. Alternate versions are provided for the use with non-method procedures or functions (ForEachProc, FirstThatProc, LastThatProc), but these cannot be passed local routines.

Array objects will report errors by raising exceptions. The most common exception you may expect is **EOutOfMemory**, which may crop up when the array object is created or resized. **ERangeError** will be generated only, when range checking is enabled and signifies an index > MaxIndex error for any method or property using an index parameter. Custom exceptions like ECompUndefined and ETypeMismatch may be raised by certain methods, too.

FMemory

FMemSize

FItemSize

FMaxIndex

FSortOrder

FCompareProc

FFlags

Methods of TBaseArray

Alphabetic list of methods

Constructors and Destructors

Create	Destroy

Accessing Data

GetItemPtr	PutItem	GetItem
Append	Insert	Delete
BlockCopy	CopyTo	CopyFrom

Iterators, Searching & Sorting

Sort	Find
FirstThat	FirstThatProc
LastThat	LastThatProc
ForEach	ForEachProc

Saving and Loading from Files and Streams

SaveToFile	LoadFromFile	
SaveToStream	LoadFromStream	
Load	Store	DefineProperties

Manipulating the whole Object

Clone	AssignTo	ReDim
Zap		

Auxillary & Sundry Methods

GetCount	GetCapacity	GetMaxCapacity
InvalidateItems	ValidIndex	ValidateBounds
SetFlag	ClearFlag	HasFlag
SetCompareProc		

Capacity	CompareProc
Count	ItemPtr
ItemSize	SortOrder
MaxCapacity	MaxIndex
Memory	MemSize
Flags	

TBaseArray
TIntegerArray
TCardinalArray
TLongIntArray
TRealArray
TSingleArray
TDoubleArray
TExtendedArray
TPointerArray
TPCharArray
TPStringArray
Comparing, sorting and searching array elements

Procedure **Append**(Var Source; numItems: Cardinal); **(Method)**

See Also

Class: TBaseArray
Visibility: public
Directives: virtual

Parameters:

Source data area to copy the new array elements from, has to be at least ItemSize **numItems* bytes large.

numItems number of array elements to copy from *Source*.

Description:

This method appends the passed elements to the array. The array grows by *numItem* elements, independent of the setting of the AF_AutoSize flag. If it cannot grow enough, not all elements may be copied from *Source!* The method does a blind-faith mem-to-mem copy of *numItems**ItemSize bytes from *Source* to the new elements added to the end of the array by ReDim. It also sets the SortOrder of the array to TS_NONE.

Error Conditions:

If the method is asked to append more elements than can fit without growing the array to more than 64 Kbytes size, the *numItems* parameter is adjusted to the maximal number of elements possible, without an exception beeing raised. Thsi applies only to the 16-bit version (Delphi 1.0). Redim is used to grow the array and it may raise an EOutOfMemory exception. If *Source* is smaller than promised a GPF may result due to read beyond end of segment.

Procedure **AssignTo**(Dest: TPersistent); **(Method)**

See Also

Class: TBaseArray
Visibility: private
Directives: override

Parameters:

Dest an object of the same type as this one

Description:

This method copies the contents of this array to the destination array, provided the destination is a descendant of TBaseArray and has the same component size. The destination array is redim'ed to the same size as this array. The actual copy is performed by the CopyFrom method, which a descendant class can override to realize a deep copy, for instance, if the items stored in the array are pointers. Existing elements in the target array will be invalidated.

This method overrides an abstract method of TPersitent, that is called by the Assign method.

Error Conditions:

This method will raise a ETypeMismatch exception, if the type of the destination does not match that of Self. It may also cause a protection fault, if *Dest* ist Nil (really stupid!) or an out of memory exception in ReDim.

Procedure **BlockCopy**(Source: TBaseArray; fromIndex, toIndex, numitems: Cardinal); **(Method)**

See Also

Class: TBaseArray
Visibility: public
Directives: virtual

Parameters:

Source	array to copy elements from. This has to be an array object of the same type as Self.
fromIndex	index in Source of the first element to copy. Has to be in the range 0..Source.MaxIndex.
toIndex	index in Self to copy the first element to. Has to be in the range 0..MaxIndex.
numItems	number of array elements to copy from *Source*.

Description:

Uses CopyFrom to do the actual copy process after doing a few sanity checks on the source. CopyFrom does the checks on the target. The *numitems* count may be reduced if either the source does not have that many elements after *fromIndex* or Self cannot take them. The target array will **not** grow automatically, if needed!

The overwritten elements will be invalidated with a call to InvalidateItems. The method also sets the SortOrder of the array to TS_NONE.

Error Conditions:

Will raise a ETypeMismatch exception if the *Source* object is not of the same or a derived type as Self and also if it has a different item size. ERangeError exceptions may be raised by called methods, if range checking is enabled.

Capacity: Cardinal **(Property)**

See Also

Class TBaseArray
Visibility: public
Access: read-only
Description:

This property returns the number of elements (items) in the array. This value will change any time the array is resized. The property is implemented via the GetCapacity method.

Procedure **ClearFlag**(aFlag: TArrayFlags); **(Method)**

See Also

Class: TBaseArray
Visibility: public
Directives: none (static)

Parameters:

aFlag array flag to clear

Description:

This method clears the passed flag bit in the arrays FFlag field. See "The Array Flags" for details about the way array flags work.

Error Conditions:

none

Function **Clone:** TBaseArray; **(Method)**

See Also

Class: TBaseArray
Visibility: public
Directives: virtual

Parameters: none

Return value: Pointer to a freshly minted exact copy of this object.

Description:

Creates a new object of the same type as this one is and copies the arrays contents to the new object via AssignTo. If the actual class type stores pointers to other stuff it is the responsibility of that class to override the CopyFrom method eventually used to implement a deep copy.

The Clone method relies on all descendants overriding the Create Constructor they inherit from TBaseArray!

Error Conditions:

Construction of the new object may fail due to out of memory. The assign process may conceivably also fail, if it involves a deep copy. If that happens, the raised EOutOfMemory is trapped, the new object destroyed and the exception is reraised for handling at an upper level.

CompareProc: TCompareProc **(Property)**

See Also

Class TBaseArray
Visibility: public
Access: read/write
Description:

This property allows access to the private FCompareProc field that stores the function pointer for the comparison function the array uses to sort itself and to search for items. You can change this function anytime to sort by a different criterion. Assigning Nil to this property disables Sort and Find and will result in an ECompUndefined exception if these two methods are used. Setting this property will also automatically set (or clear, if it is set to Nil) the AF_CanCompare flag. Writing to the property invokes the SetCompareProc method.

Procedure **CopyFrom**(Var Source; toIndex, numitems: Cardinal); **(Method)**

See Also

Class: TBaseArray
Visibility: public
Directives: virtual

Parameters:

Source	memory location to copy elements from. This has to be an area at least *numitems**ItemSize bytes large..
toIndex	index in Self to copy the elements to. Has to be in the range 0..MaxIndex.
numItems	number of array elements to copy from *Source*.

Description:

This methods overwrites the next *numItems* elements in this array starting at position *toIndex* with elements from the Source. The target array will **not** grow automatically, if needed! Instead the *numItems* value may be reduced if the specified number of elements will not fit into the target array.

A derived class storing pointers or objects may need to override this method to provide a deep copy mechanism. Using the base classes method will result in a shallow copy, with duplicate references to the same memory or object in both source and target array.

The overwritten elements will be invalidated with a call to InvalidateItems. The method also sets the SortOrder of the array to TS_NONE.

Error Conditions:

If toIndex is > MaxIndex the method will raise a ERangeError exception, if range checking is on, or do nothing if range checking is off. If the *Source* memory contains less than the specified number of elements to copy a protection fault may result.

Procedure **CopyTo**(Var Dest; fromIndex, numitems: Cardinal); **(Method)**

See Also

Class: TBaseArray
Visibility: public
Directives: virtual

Parameters:

Dest	memory location to copy elements to. This has to be an area at least *numitems**ItemSize bytes large..
fromIndex	index in Self to copy the elements from. Has to be in the range 0..MaxIndex.
numItems	number of array elements to copy to *Dest*.

Description:

This method copies *numItems* elements from this array to a memory target. If the method is asked to copy more elements than there are, the *numItems* parameter is adjusted to the maximal number of elements possible without an exception beeing raised.

The method may have a problem if the copied elements are pointers or objects, since this is a shallow copy and the result will be several references to the same memory locations! A derived class may have to override this method to deal with this problem.

Error Conditions:

If toIndex is > MaxIndex the method will raise a ERangeError exception, if range checking is on, or do nothing if range checking is off. If the *Dest* memory can hold less than the specified number of elements to copy a protection fault may result.

Count: Cardinal **(Property)**

See Also

Class TBaseArray
Visibility: public
Access: read-only
Description:

This property returns the number of *used* elements (items) in the array. The property is implemented via the GetCount method.

For the base class TBaseArray, Count is equivalent to Capacity, that is, all elements of the array are considered in use. However, a descendant class may override GetCount and implement a behaviour more like Borland Pascals collections, where the array serves as container with slots that are filled one after the other. The Count property is used by all the iterator methods as well as by Sort and Find, which will thus only work on the filled slots in such a descendant class.

Constructor **Create**(itemcount, itemsize: Cardinal);

See Also: Create for derived classes

Class	TBaseArray
Visibility:	public
Directives:	virtual

Parameters:

itemcount	number of items the array should hold, 0 is mapped to 1 since the array size cannot be 0!
itemsize	size in bytes of an individual item

Description:

Allocates the memory for the array and sets the fields according to the passed data. In the Win16 version the product of *itemcount* and *itemsize* has to be < 64Kbyte. This limitation does not apply to the Win32 version. All items in the array are set to 0. Create also defines the initial set of array flags.

We reduce the *itemcount* to an allowed value, if necessary, without raising any error if it is to large.

All descendants of TBaseArray have to provide an overridden Create constructor to allow the Clone method to work properly.

Error Conditions:

If GetMem fails we rely on the default exception handling to fail the constructor via an EOutOfMemory exception.

Procedure **DefineProperties**(Filer: TFiler); **(Method)**

See Also

Class: TBaseArray
Visibility: private
Directives: override

Parameters:

Filer a Delphi storage object

Description:

This methods prepares the object for streaming by telling the Filer which methods to call for loading and storing the array data. It should not be called directly. The methods assigned are Load and Store.

Error Conditions:

none

Procedure **Delete**(atIndex, numItems: Cardinal); **(Method)**

See Also

Class: TBaseArray
Visibility: public
Directives: virtual

Parameters:

atIndex index to first element to delete, has to be in the range 0..MaxIndex.

numItems number of array elements to delete.

Description:

This method deletes elements by moving all elements above the requested position down *numItems* slots and filling the last *numItems* elements with 0. If the AF_AutoSize flag is set the method also redims the array to the smaller size. The deleted elements are invalidated first, so a descendant class storing pointers or objects can provide a way to free the storage for the deleted elements or do other cleanup tasks, as appropriate, by overriding InvalidateItems..

Error Conditions:

If *atIndex* is > MaxIndex the method will raise a ERangeError exception, if range checking is on, or do nothing if range checking is off. If the method is asked to delete more items than there are, the *numItems* parameter is adjusted to the maximal number of items possible, without an exception beeing raised. ReDim may cause an EOutOfMemory exception even if the memory block shrinks. This is due to the implementation of ReAllocMem, the Delphi function beneath ReDim, which uses an allocate-new-block-and-copy strategy instead of shrinking the block in place. If such an error occurs, the requested elements will have already been deleted and the extra elements at the arrays end are set to 0.

Destructor **Destroy**;

Class TBaseArray
Visibility: public
Directives: override

Parameters: none

Description:

Standard destructor, frees the memory allocated for the array and then calls the inherited destructor. Descendants of TBaseArray do not need to override Destroy unless they have extra cleanup duty to perform.

Error Conditions:

none

FCompareProc: TCompareProc **(Field)**

See Also

Class TBaseArray
Visibility: private
Description:

This field holds the function pointer to the current comparison function of the array. You can obtain its value via the CompareProc property (read/write). The value is set by the Sort method and also by any method changing data in the array (which will make the array unsorted). .

FFlags: TArrayFlagSet **(Field)**

See Also

Class TBaseArray
Visibility: private
Description:

This field holds set of the current array flags. These flags determine some important aspects of an arrays behaviour. See "The Array Flags" for a discussion. You can access the flags via Flags property as a set or manipulate individual flags with the HasFlag, SetFlag. and ClearFlag methods.

FItemSize: Cardinal **(Field)**

See Also

Class TBaseArray
Visibility: private
Description:

This field holds the size of an array element (item) in bytes. You can obtain its value via the ItemSize property (read only). The value is set at construction time and cannot be changed later.

FMaxIndex: Cardinal **(Field)**

See Also

Class TBaseArray
Visibility: private
Description:

This field holds highest index value allowed for the array, the first item has index 0. You can obtain this value via the MaxIndex property (read only). The value will change each time the array is resized.

FMemSize: Cardinal **(Field)**

See Also

Class TBaseArray
Visibility: private
Description:

This field holds the allocated size of the array memory in bytes. You can obtain its value via the MemSize property (read only). The value will change each time the array is resized.

FMemory: Pointer **(Field)**

See Also

Class TBaseArray
Visibility: private
Description:

This field holds the pointer to the array data. You can obtain its value via the Memory property (read only). It is strongly recommended that you do not access the array memory directly and do not pass this pointer around or store it in variables. *The pointer will change anytime the array is resized!*

FSortOrder: TSortOrder **(Field)**

See Also

Class TBaseArray
Visibility: private
Description:

This field holds the current sort order of the array. You can obtain its value via the SortOrder property (read/write). The value is set by the Sort method and also by any method changing data in the array (which will make the array unsorted). .

Function **Find**(Var value): Cardinal; **(Method)**

See Also

Class: TBaseArray
Visibility: public
Directives: virtual

Parameters:

value a data element to search for

Return value: the index of the found element in the array (possible values are in the range 0..MaxIndex) or the constant NOT_FOUND, if the passed data was not found in the array.

Description:

Depending on the sort state of the array this Function will do a binary or sequential search thru the array, using the comparison function supplied in FCompareProc to compare *value* to the current array element. Find is also dependent on the state of the AF_CanCompare flag. If this flag is not set, the methods immediately returns NOT_FOUND!

Warning! If the list is sorted and contains multiple instances of the same value, the search will not necessarily find the *first* instance of this value! This is a general shortcome of binary search; set SortOrder to TS_NONE before the search to force sequential search if the array contains multiple copies of the same value.

Like for the Sort method descendants may gain a considerable improvement in performance if they reimplement this method with optimized data access and comparison.

Error Conditions:

Will raise a ECompUndefined exception if no comparison function has been defined but the AF_CanCompare flag is set.

Function **FirstThat**(locator: TLocatorMethod; processMsg: Boolean; intervall: Cardinal): Pointer; **(Method)**

See Also

Class: TBaseArray
Visibility: public
Directives: none (static)

Parameters:

locator	an object method to call for each element
processMsg	True, if Application.ProcessMessages should be called during iteration, False otherwise
intervall	determines how often ProcessMessages is called, a higher number means messages will be processed less often since the method uses (index mod intervall)=0 as trigger to call ProcessMessages. If *processMsg* is False this parameter will be ignored.
Return value:	The address of the element for which the *locator* returned True, or **Nil** if it returned False for all elements.

Description:

The method loops over all entries of the array and passes the address of each with its index to the *locator* method. The loop terminates immediately when the *locator* method returns True. If *processMsg* = True, the method will call Application.ProcessMessages on each *intervall'th* round of the loop. Note that this only happens when this Unit has been compiled with the symbol DOEVENTS defined!

Error Conditions:

The method has no error conditions per se but horrible things will happen if you call it with a **Nil** *locator* since we do not check for this condition!

Function **FirstThatProc**(locator: TLocator; processMsg: Boolean; intervall: Cardinal): Pointer; **(Method)**

See Also

Class: TBaseArray
Visibility: public
Directives: none (static)

Parameters:

locator	a function to call for each element
processMsg	True, if Application.ProcessMessages should be called during iteration, False otherwise
intervall	determines how often ProcessMessages is called, a higher number means messages will be processed less often since the method uses (index mod intervall)=0 as trigger to call ProcessMessages. If *processMsg* is False this parameter will be ignored.
Return value:	The address of the element for which the *locator* returned True, or **Nil** if it returned False for all elements.

Description:

The method loops over all entries of the array and passes the address of each with its index to the *locator* function. The loop terminates immediately when the *locator* function returns True. If *processMsg* = True, the method will call Application.ProcessMessages on each *intervall'th* round of the loop. Note that this only happens when this Unit has been compiled with the symbol DOEVENTS defined!

Error Conditions:

The method has no error conditions per se but horrible things will happen if you call it with a **Nil** *locator* since we do not check for this condition!

Flags: TArrayFlagSet **(Property)**

See Also

Class TBaseArray
Visibility: public
Access: read/write
Description:

This property allows access to the FFlags field holding the array flags as a set. These flags determine some important aspects of an arrays behaviour. See "The Array Flags" for a discussion. You manipulate individual flags with the HasFlag, SetFlag. and ClearFlag. methods. Using Include and Exclude on the set returned by this property is a futile exercise because it is only a copy of the real thing!

Procedure **ForEach**(iterator: TIteratorMethod; processMsg: Boolean; intervall: Cardinal);
(Method)

See Also

Class: TBaseArray
Visibility: public
Directives: none (static)

Parameters:

iterator	an object method to call for each element
processMsg	True, if Application.ProcessMessages should be called during iteration, False otherwise
intervall	determines how often ProcessMessages is called, a higher number means messages will be processed less often since the method uses (index mod intervall)=0 as trigger to call ProcessMessages. If *processMsg* is False this parameter will be ignored.

Description:

The method loops over all entries of the array, and passes the address of each with its index to the *iterator* method. If *processMsg* = True, the method will call Application.ProcessMessages on each *intervall'th* round of the loop. Note that this only happens when this Unit has been compiled with the symbol DOEVENTS defined!

Error Conditions:

The method has no error conditions per se but horrible things will happen if you call it with a **Nil** *iterator* since we do not check for this condition!

Procedure **ForEachProc**(iterator: TIterator; processMsg: Boolean; intervall: Cardinal); **(Method)**

See Also

Class: TBaseArray
Visibility: public
Directives: none (static)

Parameters:

iterator	a procedure to call for each element
processMsg	True, if Application.ProcessMessages should be called during iteration, False otherwise
intervall	determines how often ProcessMessages is called, a higher number means messages will be processed less often since the method uses (index mod intervall)=0 as trigger to call ProcessMessages. If *processMsg* is False this parameter will be ignored.

Description:

The method loops over all entries of the array, and passes the address of each with its index to the *iterator* procedure. If *processMsg* = True, the method will call Application.ProcessMessages on each *intervall'th* round of the loop. Note that this only happens when this Unit has been compiled with the symbol DOEVENTS defined!

Error Conditions:

The method has no error conditions per se but horrible things will happen if you call it with a **Nil** *iterator* since we do not check for this condition!

Function **GetCapacity:** Cardinal; **(Method)**

See Also

Class: TBaseArray
Visibility: private
Directives: none (static)

Returns: the number of elements in the array

Description:

This method is used to implement the Capacity property. It returns the number of elements the array can hold (MaxIndex+1).

Error Conditions:

none

Function **GetCount:** Cardinal; **(Method)**

See Also

Class: TBaseArray
Visibility: public
Directives: virtual

Returns: the number of used items in the array

Description:

This method is used to implement the Count property. For this class it acts like Capacity, because all items of the array are considered in use. But for a descendant class that works more like a BP collection, only part of the items may be actually used. These classes can override GetCount to return the actually used number. The Count property is used by Sort, Find and the iterator methods to get the upper bound of the range to operate on; these methods will thus work without changes in collection-like descendants.

Error Conditions:

none

Procedure **GetItem**(Var data; index: Cardinal); **(Method)**

See Also

Class: TBaseArray
Visibility: public
Directives: none (static)

Parameters:

data data item to copy the array element to, has to be at least ItemSize large.

index index of the item to copy the data from. Valid values are 0..MaxIndex

Description:

Uses a direct mem-to-mem copy to put the data into the passed variable. No error checks on type of the passed data are possible here, the method just blindly copies ItemSize bytes to the destination!

Error Conditions:

If the passed index is out of range, the method will do nothing at all!.If *Data* is smaller than ItemSize you may also get a GPF if it is in a segment of its own or happens to be at the end of a segment.

Function **GetItemPtr**(index: Cardinal): Pointer; **(Method)**

See Also

Class: TBaseArray
Visibility: public
Directives: none (static)

Parameters:

index index of the item to access. Valid values are 0..MaxIndex.

Returns: pointer to the requested element of the array.

Description:

Does brute-force pointer arithmetic to calculate the items address from index and size.
Unless no array memory has been allocated (impossible under normal conditions) this method will never return Nil. Even in case of an invallid index (see below) it will return a valid pointer (to the first array element)!

This method is used to implemente the ItemPtr property, you should thus have no need to call it directly.

Error Conditions:

If the passed index is out of range, the method will raise an ERangeError exception, if range checking is enabled, otherwise it returns a pointer to the first item in the array.

Function **GetMaxCapacity:** Cardinal; **(Method)**

See Also

Class: TBaseArray
Visibility: private
Directives: none (static)

Returns: the maximal number of elements an array of this base type can hold without exceeding 64 KBytes in Win16 or High(Cardinal) bytes in Win32.

Description:

This method is used to implement the MaxCapacity property. It returns the number of elements the largest possible array can hold (High(Cardinal) div ItemSize).

Error Conditions:

none

Function **HasFlag**(aFlag: TArrayFlags): Boolean; **(Method)**

See Also

Class: TBaseArray
Visibility: public
Directives: none (static)

Parameters:

aFlag array flag to test

Returns: true is the flag is set, false otherwise

Description:

This method tests the passed flag bit in the arrays FFlag field. See "The Array Flags" for details about the way array flags work.

Error Conditions:

none

Procedure **Insert**(Var Source; atIndex, numItems: Cardinal); **(Method)**

See Also

Class: TBaseArray
Visibility: public
Directives: virtual

Parameters:

Source data area to copy the new array elements from, has to be at least ItemSize **numItems* bytes large.

atIndex index to insert the new elements at. If this value is > MaxIndex, the method will just call Append to append the elements to the array.

numItems number of array elements to copy from *Source*.

Description:

This method inserts the passed elements, moving all elements from position *atIndex* and up *numItems* positions upwards. The array grows as needed, if the AF_AutoSize flag is set. If this flag is not set or if the array cannot grow enough (the maximal size is 64 KBytes in Win16), elements will fall off the end! In this case the lost elements will be invalidated with a call to InvalidateItems. The method also sets the SortOrder of the array to TS_NONE.

Error Conditions:

If the method is asked to insert more elements than can be fitted from position *atIndex* without running across the 64 Kbytes barrier, the *numItems* parameter is adjusted to the maximal number of elements possible, without an exception beeing raised. This limitation does not apply to the Win32 version. Redim is used to grow the array and it may raise an EOutOfMemory exception. If *Source* is smaller than promised a GPF may result due to read beyond end of segment.

Procedure **InvalidateItems**(atIndex, numItems: Cardinal); **(Method)**

See Also TPointerArray.InvalidateItems

Class: TBaseArray
Visibility: public
Directives: virtual

Parameters:

atIndex index of the first element to invalidate (0..MaxIndex)

numItems number of elements to invalidate

Description:

This method does nothing for the base class. It is provided for the benefit of derived classes that need to do cleanup on deleted elements. InvalidateItems is called by most methods that overwrite or delete elements before the elements are nuked.

Error Conditions:

none

ItemPtr[Index:Cardinal]**:** Pointer **(Property)**

See Also

Class TBaseArray
Visibility: public
Access: read-only
Description:

This property returns a pointer to an element (item) in the array. The property is implemented via the GetItemPtr method. It will work for any basetype of the array. The *Index* parameter must be in the range 0..MaxIndex, an invalid index will cause an **ERangeError** exception, if range checking is enabled, or cause the property to return a pointer to the *first* element of the array. The property will never return Nil or an invalid pointer, unless some unexpected tampering with the arrays memory has invalidated the FMemory field.

ItemSize: Cardinal **(Property)**

See Also

Class TBaseArray
Visibility: public
Access: read-only
Description:

This property returns the size of an element (item) in the array. The property is implemented via the FItemSize field.

Function **LastThat**(locator: TLocatorMethod; processMsg: Boolean; intervall: Cardinal): Pointer; **(Method)**

See Also

Class: TBaseArray
Visibility: public
Directives: none (static)

Parameters:

locator	an object method to call for each element
processMsg	True, if Application.ProcessMessages should be called during iteration, False otherwise
intervall	determines how often ProcessMessages is called, a higher number means messages will be processed less often since the method uses (index mod intervall)=0 as trigger to call ProcessMessages. If *processMsg* is False this parameter will be ignored.
Return value:	The address of the element for which the *locator* returned True, or **Nil** if it returned False for all elements.

Description:

The method loops over all entries of the array, starting with the last and going backwards, and passes the address of each with its index to the *locator* method. The loop terminates immediately when the *locator* method returns True. If *processMsg* = True, the method will call Application.ProcessMessages on each *intervall'th* round of the loop. Note that this only happens when this Unit has been compiled with the symbol DOEVENTS defined!

Error Conditions:

The method has no error conditions per se but horrible things will happen if you call it with a **Nil** *locator* since we do not check for this condition!

Function **LastThatProc**(locator: TLocator; processMsg: Boolean; intervall: Cardinal): Pointer; **(Method)**

See Also

Class: TBaseArray
Visibility: public
Directives: none (static)

Parameters:

locator	a function to call for each element
processMsg	True, if Application.ProcessMessages should be called during iteration, False otherwise
intervall	determines how often ProcessMessages is called, a higher number means messages will be processed less often since the method uses (index mod intervall)=0 as trigger to call ProcessMessages. If *processMsg* is False this parameter will be ignored.
Return value:	The address of the element for which the *locator* returned True, or **Nil** if it returned False for all elements.

Description:

The method loops over all entries of the array, starting with the last and going backwards, and passes the address of each with its index to the *locator* function. The loop terminates immediately when the *locator* function returns True. If *processMsg* = True, the method will call Application.ProcessMessages on each *intervall'th* round of the loop. Note that this only happens when this Unit has been compiled with the symbol DOEVENTS defined!

Error Conditions:

The method has no error conditions per se but horrible things will happen if you call it with a **Nil** *locator* since we do not check for this condition!

Procedure **Load**(Reader: TReader); **(Method)**

See Also

Class: TBaseArray
Visibility: public
Directives: virtual

Parameters:

Reader a Delphi storage object to read the array object from

Description:

This method reads the array data from the Reader storage object. If the array already contains data, that is released first. The method should not be called directly, it will be called from the filer object handling the load.

Error Conditions:

This method does not raise exception by itself but out of memory and I/O error exceptions may be raised by the called functions.

Procedure **LoadFromFile**(Const Filename: String); **(Method)**

See Also

Class: TBaseArray
Visibility: public
Directives: virtual

Parameters:

Filename name of the file to read

Description:

Loads the contents of the requested file into the array, which is redimensioned to fit the data. For this to work smoothly the file should have been created by the SaveToFile method of an array object of the same type as this one and it must be < 64KBytes in size! If it is larger only part of it will be read. This limitation does not apply to the Win32 version. If the items in the file do have a different item size than this array assumes (a fact we cannot check), the loaded data will propably come out as garbage!

Error Conditions:

May raise a EInOutError exception if a file-related error occurs.

Procedure **LoadFromStream**(Stream: TStream); **(Method)**

See Also

Class: TBaseArray
Visibility: public
Directives: virtual

Parameters:

Stream stream to read the array data from. The stream must be open!

Description:

This method reads the stored arrays item size and max index and checks the item size vs. our own item size. If these two do match, the array is redimensioned according to the needed size and the array data are read from the passed stream.
Note that this is different from LoadFromFile, which only reads the array data and assumes they have the right item size! You can use this method to get the array data from an open stream that can already contain other data in front and additional data after. However, it is your responsibility to position the stream pointer correctly.

Warning! The method does not preserve or invalidate any elements already present in the array, with the exception of elements that fall off at the end if the array is resized to a smaller size (here Redim will invalidate the elements).

Error Conditions:

The stream may raise an exception if it runs into problems. Which one depends on the type of stream. We will raise an ETypeMismatch exception if the item size read from the stream does not match our own item size.

MaxCapacity: Cardinal **(Property)**

See Also

Class TBaseArray
Visibility: public
Access: read-only
Description:

This property returns the maximum number of elements (items) the array can hold without exceeding 64KBytes in Win16 or HIgh(Cardinal) bytes in Win32. The property is implemented via the GetMaxCapacity method.

MaxIndex: Cardinal **(Property)**

See Also

Class TBaseArray
Visibility: public
Access: read-only
Description:

This property returns the highest valid index for elements (items) in the array. The property is implemented via the FMaxIndex field.The value returned by the property will change every time the array is resized. The first element of the array has index **0**.

MemSize: Cardinal **(Property)**

See Also

Class TBaseArray
Visibility: public
Access: read-only
Description:

This property returns the size, in bytes, allocated for the array. The property is implemented via the FMemSize field.The value returned by the property will change every time the array is resized.

Memory: pointer **(Property)**

See Also

Class TBaseArray
Visibility: public
Access: read-only
Description:

This property returns the address of the memory holding the array data. The property is implemented via the FMemory field.The value returned by the property will change every time the array is resized.

It is not recommended to access the array memory directly but it can be done via the pointer returned by Memory, if necessary.

Procedure **PutItem**(Var data; index: Cardinal); **(Method)**

See Also

Class: TBaseArray
Visibility: public
Directives: none (static)

Parameters:

data data to copy into the array element, has to be at least ItemSize large.

index index of the item to copy the data to. Valid values are 0..MaxIndex

Description:

Uses a direct mem-to-mem copy to put the data into the array. No error checks on type of the passed data are possible here, the method just blindly copies ItemSize bytes from the source! It also sets the SortOrder of the array to TS_NONE.

Note: The method obviously overwrites the old contents of the index slot but it does *not* invalidate the old entry! Thus this method can be used by an InvalidateItems handler to set an element to Nil in an array of pointers or objects..

Error Conditions:

If the passed index is out of range, the method will do nothing at all. If *Data* is smaller than ItemSize you may also get a GPF if it is in a segment of its own or happens to be at the end of a segment.

Procedure **ReDim**(newcount: Cardinal); **(Method)**

See Also

Class: TBaseArray
Visibility: public
Directives: virtual

Parameters:

newcount number of items the new array should hold, cannot be 0! 0 is mapped to 1. The maximum number possible can be obtained via MaxCapacity.

Description:

Reallocates the array to a new size. The old items are copied over, as far as possible. New slots are nulled out. If the new array is smaller than the old one the extra items are invalidated so a derived class can do cleanup on them.

Error Conditions:

ReAllocMem, the RTL function used, may raise an out of memory exception. If compiled with debugging on ($D+) we will raise an ERangeError exception, if the requested size is > 64K and we are compiling for Win16. Otherwise *newcount* is just set to MaxCapacity.

Procedure **SaveToFile**(Const Filename: String); **(Method)**

See Also

Class: TBaseArray
Visibility: public
Directives: virtual

Parameters:

Filename name of the file to save to

Description:

Saves the data in this array to a file. Only the array data itself is written, neither the component size not the number of elements are stored! This makes it possible to access the file as a File Of Component (where Component is the type stored in this array, not a Delphi Component!).

Error Conditions:

May raise a EInOutError exception if a file-related error occurs.

Procedure **SaveToStream**(Stream: TStream); **(Method)**

See Also

Class: TBaseArray
Visibility: public
Directives: virtual

Parameters:

Stream stream to save the array to. The stream must be open!

Description:

This method stores the arrays item size and max index (NOT the number of items!) followed by the array data into the passed stream. Note that this is different from SaveToFile, which only writes the array data! You can use this method to append the array data to an open stream that can already contain other data in front and receive additional data after we are done here. We do not stream the array object itself, only its data!

Error Conditions:

The stream may raise an exception if it runs into problems. Which one depends on the type of stream.

Procedure SetCompareProc(proc: TCompareProc); **(Method)**

See Also

Class: TBaseArray
Visibility: public
Directives: none (static)

Parameters:

proc comparison function to use, may be Nil

Description:

This method is used by the CompareProc property to implement writing. It stores the passed function pointer in the FCompareProc field and also sets or clears (if *proc* = Nil) the AF_CanCompare flag.

Error Conditions:

none

Procedure **SetFlag**(aFlag: TArrayFlags); **(Method)**

See Also

Class: TBaseArray
Visibility: public
Directives: none (static)

Parameters:

aFlag array flag to set

Description:

This method sets the passed flag bit in the arrays FFlag field. See "The Array Flags" for details about the way array flags work.

Error Conditions:

none

Procedure **Sort**(ascending: Boolean); **(Method)**

See Also

Class: TBaseArray
Visibility: public
Directives: virtual

Parameters:

ascending determines whether the array is sorted in ascending (True) or descending (False) order

Description:

This method implements a recursive QuickSort. It can only do its work if a comparison function has been assigned to the FCompareProc field. Since this is a generic procedure to sort any kind of data, it is possible to get a much better performance for specific data types by reimplementing the Sort for this type. The recursive implementation also uses a lot of stack for larger arrays, which may be a problem under Win16.

If the array is already sorted in the way requested (at least the FSortOrder fields is set that way) the method will do nothing. You may set that field to TS_NONE via the SortOrder property to force a sort.

The behaviour of Sort is dependent on the flag AF_CanCompare. If this flag is not set, Sort will return immediately!

Error Conditions:

Will raise a ECompUndefined exception if no comparison function has been defined but the AF_CanCompare flag is set. The method may also run out of memory in GetMem while allocating the pivot data buffer or run into a stack overflow (stack checking is enabled unconditional for this method). You should always call Sort in a local try..except block and check the SortOrder property afterwards to avoid rude surprises.

```
try
  Sort;
except
  on e: ECompUndefined Do
{ this exception should always be reraised since it is caused by a
  programming error: failure to assign a comparison function. This should
  be fixed during testing and never appear in the production version. }
    raise
  else
    { swallow all other exceptions here }
end;
If SortMode = TS_NONE Then
{ issue message to user, we probably run into a stack overflow while
sorting }
```

SortOrder: TSortOrder **(Property)**

See Also

Class TBaseArray
Visibility: public
Access: read/write
Description:

This property returns the current sort order of the array; it can also be used to set the sort order, normally to TS_NONE. The property is implemented via the FSortOrder field. The value may change every time something is written to the array!

Procedure **Store**(Writer: TWriter); **(Method)**

See Also

Class: TBaseArray
Visibility: public
Directives: virtual

Parameters:

Writer a Delphi storage object to write the array object to

Description:

This method will be called by a filer object to write the array data to the stream or whatever storage the *Writer* stands for. The method should not be called directly.

.Error Conditions:

The Writer may raise an I/O error exception.

Function **ValidIndex**(index: Cardinal): Boolean; **(Method)**

See Also

Class: TBaseArray
Visibility: public
Directives: none (static)

Parameters:

Index index to check

Returns: True, if the index is in bounds, False otherwise.

Description:

This method is used by a lot of others to validate an index.

Error Conditions:

If *Index* is > MaxIndex the method will raise a ERangeError exception, if range checking is on, or just return false if range checking is off.

Function **ValidateBounds**(atIndex: Cardinal; Var numItems: Cardinal): Boolean; **(Method)**

See Also

Class: TBaseArray
Visibility: public
Directives: none (static)

Parameters:

atIndex index to check

numItems range starting at *atIndex* to verify

Returns: True, if the index is in bounds, False otherwise.

Description:

This method is used by a couple of others to validate an index and make sure that *numItems* is not higher than the number of items from position *atIndex* on to the end of array. .

Error Conditions:

If *Index* is > MaxIndex the method will raise a ERangeError exception, if range checking is on, or just return false if range checking is off.

Procedure **Zap**; **(Method)**

See Also

Class: TBaseArray
Visibility: public
Directives: virtual

Parameters:

none

Description:

This method invalidates all used elements of the array and then fills it with 0.

Error Conditions:

none but it really depends on what InvalidateItem does for the specific array class.

ECompUndefined
ETypeMismatch

TCardinalArray (Class)

See Also Fields Methods Properties Exceptions

Unit: Arrays
Ancestor: TBaseArray

Description:

This is a specialized array class to store cardinals. It inherits all fields, methods, and properties from TBaseArray and may cause the same exceptions. The class overrides the Create constructor of the base class.

To make the usage of instances of this class more like standard Pascal arrays, the class has a default property Data that provides the normal array syntax (aValue := array[index]; and array[index] := value). This property is implemented with two new public methods. PutData and GetData, which you will normally not need to call directly. The class also has a comparison function assigned, so you can use the Sort and Find methods of the parent class directly.

Type

TCompareProc = Function (VAR item1, item2): Integer;

Our virtual arrays need a function of this type to sort themselves and search items. As usual the return type should be < 0 if item1 < item2, > 0 if item1 > item2 and 0 if both are equal. Note that the result is not limited to -1, 0, +1! This allows faster comparison.

TDoubleArray (Class)

See Also Fields Methods Properties Exceptions

Unit: Arrays
Ancestor: TBaseArray

Description:

This is a specialized array class to store double-precision floating point numbers. It inherits all fields, methods, and properties from TBaseArray and may cause the same exceptions. The class overrides the Create constructor of the base class.

To make the usage of instances of this class more like standard Pascal arrays, the class has a default property Data that provides the normal array syntax (aValue := array[index]; and array[index] := value). This property is implemented with two new public methods. PutData and GetData, which you will normally not need to call directly. The class also has a comparison function assigned, so you can use the Sort and Find methods of the parent class directly.

TExtendedArray (Class)

See Also Fields Methods Properties Exceptions

Unit: Arrays
Ancestor: TBaseArray

Description:

This is a specialized array class to store extended floating-point numbers. It inherits all fields, methods, and properties from TBaseArray and may cause the same exceptions. The class overrides the Create constructor of the base class.

To make the usage of instances of this class more like standard Pascal arrays, the class has a default property Data that provides the normal array syntax (aValue := array[index]; and array[index] := value). This property is implemented with two new public methods. PutData and GetData, which you will normally not need to call directly. The class also has a comparison function assigned, so you can use the Sort and Find methods of the parent class directly.

TIntegerArray (Class)

See Also Fields Methods Properties Exceptions

Unit: Arrays
Ancestor: TBaseArray

Description:

This is a specialized array class to store Integers. It inherits all fields, methods, and properties from TBaseArray and may cause the same exceptions. The class overrides the Create constructor of the base class.

To make the usage of instances of this class more like standard Pascal arrays, the class has a default property Data that provides the normal array syntax (aValue := array[index]; and array[index] := value). This property is implemented with two new public methods. PutData and GetData, which you will normally not need to call directly. The class also has a comparison function assigned, so you can use the Sort and Find methods of the parent class directly.

Type

TIterator = Procedure(Var item; index: Cardinal);

This is the prototype for an iterator procedure that can be used with the ForEachProc method.

Type

TIteratorMethod =Procedure(Var item; index: Cardinal) of Object;

This is the prototype for an iterator procedure that can be used with the ForEach method.

Type

TLocator = Function(Var item; index: Cardinal): Boolean;

This is the prototype for an iterator procedure that can be used with the FirstThatProc and LastThatProc methods.

Type

TLocatorMethod = Function(Var item; index: Cardinal): Boolean of Object;

This is the prototype for an iterator procedure that can be used with the FirstThat and LastThat methods.

TLongIntArray (Class)

See Also Fields Methods Properties Exceptions

Unit: Arrays
Ancestor: TBaseArray

Description:

This is a specialized array class to store long Integers. It inherits all fields, methods, and properties from TBaseArray and may cause the same exceptions. The class overrides the Create constructor of the base class.

To make the usage of instances of this class more like standard Pascal arrays, the class has a default property Data that provides the normal array syntax (aValue := array[index]; and array[index] := value). This property is implemented with two new public methods. PutData and GetData, which you will normally not need to call directly. The class also has a comparison function assigned, so you can use the Sort and Find methods of the parent class directly.

TPCharArray (Class)

See Also | inherited Fields | Methods | Properties | Exceptions

Unit: Arrays
Ancestor: TPointerArray

Description:

This is a specialized array class to store PChars (pointers to zero-terminated strings). It inherits all fields, methods, and properties from TPointerArray and TBaseArray and may cause the same exceptions. The class overrides the Create constructor of the base class and the virtual methods FreeItem, CloneItem, SaveItemToStream and LoadItemFromStream of TPointerArray. It adds methods to read and write text files (LoadFromTextfile, SaveToTextfile).

To make the usage of instances of this class more like standard Pascal arrays, the class has a default property Data that provides the normal array syntax (aValue := array[index]; and array[index] := value). The type of this property is PChar. This property is implemented with two new public methods. PutData and GetData, which you will normally not need to call directly. There are also several properties that allow *in situ* data conversion: AsString, AsInteger, AsReal, which are all read/write. These properties are implemented with appropriate Get and Put methods.

The class also has a comparison function assigned, so you can use the Sort and Find methods of the parent class directly.

The default set of array flags for this class is **[AF_OwnsData, AF_CanCompare, AF_AutoSize]**. As a consequence, the array will make copies of any string/PChars put into it and automatically free the memory when an item is invalidated or the array is destroyed. The array will also resize automatically if you insert or delete items.

AsString[Index: Cardinal]: String **(Property)**

See Also

Class TPCharArray
Visibility: public
Access: read/write

Description:

This is a conversion property; on write access it will convert the assigned Pascal string to a zero-terminated string and store a pointer to that string in the *index-th* item of the array. On read access it will return the string at *index* as a Pascal string. If the stored string is longer than 255 characters only the first 255 will be returned.

The property is implemented via the GetAsString and PutAsString methods.

Function **GetAsString**(index: Cardinal): String; **(Method)**

See Also

Class TPCharArray,
Visibility: public
Directives: none (static)

Parameters:

Index index of the array element to get, must be in the range 0..MaxIndex.

Returns: the string store at *index* as Pascal string.

Description:

The method uses StrPas to convert the stored zero-terminated string to a Pascal string. Note that only the first 255 characters will be returned, if the stored string is longer than 255 characters.

This method is used to implement read accesss via the AsString property. You should not call it directly.

Errors:

May cause an EOutOfMemory exception, if the heap is full.

Procedure **PutAsString**(index: Cardinal; Const value: String); **(Method)**

See Also

Class TPCharArray,
Visibility: public
Directives: none (static)

Parameters:

Index	index of the array element to set, must be in the range 0..MaxIndex.
value	the Pascal string to store as zero-terminated string.

Description:

The method allocates just enough space to hold the passed string and converts it with StrPCopy to a zero-terminated string. The pointer to allocated memory is stored at *index*. If the array flag **AF_OwnsData** is set (the default) the previous PChar stored at that location will be freed automatically.

This method is used to implement write accesss via the AsString property. You should not call it directly.

Errors:

May cause an EOutOfMemory exception, if the heap is full.

TPStringArray (Class)

See Also inherited Fields Methods Properties Exceptions

Unit: Arrays
Ancestor: TPointerArray

Description:

This is a specialized array class to store PStrings (pointers to Pascal strings). It inherits all fields, methods, and properties from TPointerArray and TBaseArray and may cause the same exceptions. The class overrides the Create constructor of the base class and the virtual methods FreeItem, CloneItem, SaveItemToStream and LoadItemFromStream of TPointerArray. It adds methods to read and write text files (LoadFromTextfile, SaveToTextfile).

To make the usage of instances of this class more like standard Pascal arrays, the class has a default property Data that provides the normal array syntax (aValue := array[index]; and array[index] := value). The type of this property is String. This property is implemented with two new public methods. PutData and GetData, which you will normally not need to call directly. There are also several properties that allow *in situ* data conversion: AsPChar, AsInteger, AsReal, which are all read/write. These properties are implemented with appropriate Get and Put methods. There is also a AsPString property, which is read only.

The class also has a comparison function assigned, so you can use the Sort and Find methods of the parent class directly.

The default set of array flags for this class is **[AF_OwnsData, AF_CanCompare, AF_AutoSize]**. As a consequence, the array will make copies of any string/PChars put into it and automatically free the memory when an item is invalidated or the array is destroyed. The array will also resize automatically if you insert or delete items.

Delphi 2.0 Note

Due to the changed nature of the String type in Delphi 2.0 this array type will not be very efficient and should be abandonned in favour of a TStringList in Delphi 2.0. I don't know if there will be any problems (e.g. with proper release of string memory) if this arrays class is used in Delphi 2.0. It works ok in the context of the test program, however.

AsPChar[Index: Cardinal]: PChar **(Property)**

See Also

Class TPStringArray
Visibility: public
Access: read/write

Description:

This is a conversion property; on write access it will copy the assigned zero-terminated string to a Pascal string and store a pointer to that string in the *index-th* item of the array. The passed PChar remains the property of the caller! On read access it will return the Pascal string at *index* as a zero-terminated string. Storage for that string is allocated on the heap with StrAlloc. The returned PChar passes into the property of the caller, who has the responsibility to release the memory for it with StrDispose when it is no longer needed!

The property is implemented via the GetAsPChar and PutAsPChar methods.

Errors:

May raise an EOutOfMemory exception if no space can be allocated for the string copy.

AsPString[Index: Cardinal]: PString **(Property)**

See Also

Class TPStringArray
Visibility: public
Access: read

Description:

This property returns the pointer stored at *index* directly. It is basically an alias for AsPtr, using a typecast to convert the generic pointer returned by AsPtr to a PString. You should only use the pointer for read access to the stored string. **Do not dispose of the returned pointer!** It remains the property of the array!

Warning! If you write to the string **do not increase its length!** The strings stored in the array are all created on the heap with NewStr and thus will just have enough memory allocated to fit the number of characters and the length byte!

The property is implemented via the GetAsPString method.

Function **GetAsPChar**(index: Cardinal): PChar; **(Method)**

See Also

Class TPStringArray
Visibility: public
Directives: none (static)

Parameters:

Index index of the array element to get, must be in the range 0..MaxIndex.

Returns: will return a pointer to a copy of the Pascal string at *index* as a zero-terminated string.

Description:

Storage for the rturned zero-terminated string is allocated on the heap with StrAlloc. The returned PChar passes into the property of the caller, who has the responsibility to release the memory for it with StrDispose when it is no longer needed!

This method is used to implement read accesss via the AsPChar property. You should not call it directly.

Errors:

May raise an EOutOfMemory exception if no space can be allocated for the string copy.

Function **GetAsPString**(index: Cardinal): PString; **(Method)**

See Also

Class TPStringArray
Visibility: public
Directives: none (static)

Parameters:

Index index of the array element to get, must be in the range 0..MaxIndex.

Returns: will return a copy of the pointer to the Pascal string at *index.*

Description:

You should only use the returned pointer for read access to the stored string. **Do not dispose of the returned pointer!** It remains the property of the array!

Warning! If you write to the string **do not increase its length!** The strings stored in the array are all created on the heap with NewStr and thus will just have enough memory allocated to fit the number of characters and the length byte!

This method is used to implement read accesss via the AsPString property. You should not call it directly.

Errors:

none

Procedure **PutAsPChar**(index: Cardinal; value: PChar); **(Method)**

See Also

Class TPStringArray,
Visibility: public
Directives: none (static)

Parameters:

Index index of the array element to set, must be in the range 0..MaxIndex.

value pointer to the zero-terminated string to store as Pascal string.

Description:

The method converts the passed zero-terminated string with StrPas and stores a pointer to a copy of the Pascal string made on the heap at *index*. If the array flag **AF_OwnsData** is set (the default) the previous string stored at that location will be freed automatically. Note that only the first 255 characters of the passed string can be stored, if it is longer.

This method is used to implement write accesss via the AsPChar property. You should not call it directly.

Errors:

none

TPointerArray (Class)

See Also | inherited Fields | Methods | Properties | Exceptions

Unit: Arrays
Ancestor: TBaseArray

Description:

This is a specialized array class to store anonymous pointers. It inherits all fields, methods, and properties from TBaseArray and may cause the same exceptions. It defines no new fields but several new methods and properties. The class overrides the Create constructor of the base class, as well as the CopyFrom, CopyTo, InvalidateItem, LoadFromFile, SaveToFile, LoadFromStream, SaveToStream methods.

The class has a property Data that provides access to the stored pointers (the second property, AsPtr, is just an alias to Data for the benefit of derived classes). However, since this class serves as a base class for more specialized pointer classes, the property is not the default; you cannot use array syntax with an instance of this class. This property is implemented with two new public methods. PutData and GetData, which you will normally not need to call directly. The class also has no comparison function assigned, so you cannot use the Sort and Find methods of the parent class directly.

This is a prototype class that may not be very useful directly. It does not implement a deep copy mechanism, for example (because it has no idea what data the pointers are pointing to), and is unable to properly free deleted objects. Empty virtual methods are provided for descendant classes to implement these features (CloneItem, FreeItem, LoadItemFromStream, SaveitemToStream). The default array flags for this class are [AF_AutoSize], so neither item comparison nor deep copy and automatic disposal of deleted items are supported.

AsPtr[Index: Cardinal]: Pointer **(Property)**

See Also

Class TPointerArray
Visibility: public
Access: read/write

Description:

This property provides the array-style access to the pointers stored in a TPointerArray and all its descendants. It is equivalent to the Data property of TPointerArray.

The property is implemented via the GetData and PutData methods of TPointerArray.

Procedure **CopyFrom**(Var Source; toIndex, numitems: Cardinal); **(Method)**

See Also

Class: TPointerArray
Visibility: public
Directives: override

Parameters:

Source	memory location to copy elements from. This has to be an area at least *numitems**Sizeof(Pointer) bytes large..
toIndex	index in Self to copy the elements to. Has to be in the range 0..MaxIndex.
numItems	number of array elements to copy from *Source*.

Description:

This methods overwrites the next *numItems* elements in this array starting at position *toIndex* with elements from the Source. The target array will **not** grow automatically, if needed! Instead the *numItems* value may be reduced if the specified number of elements will not fit into the target array.

For each pointer taken from the *Source* the Cloneitem method is called and the pointer returned by it is stored in this objects array. The Cloneitem method of TPointerArray just returns the source pointer unchanged, but a derived class may override Cloneitem to provide a deep copy mechanism.

The overwritten elements will be invalidated with a call to InvalidateItems. The method also sets the SortOrder of the array to TS_NONE.

Error Conditions:

If toIndex is > MaxIndex the method will raise a ERangeError exception, if range checking is on, or do nothing if range checking is off. If the *Source* memory contains less than the specified number of elements to copy a protection fault may result. If deep copies are made, EOutOfMemory may be raised if the heap allocator runs out of space.

Procedure **CopyTo**(Var Dest; fromIndex, numitems: Cardinal); **(Method)**

See Also

Class: TPointerArray
Visibility: public
Directives: override

Parameters:

Dest	memory location to copy elements to. This has to be an area at least *numitems**Sizeof(Pointer) bytes large..
fromIndex	index in Self to copy the elements from. Has to be in the range 0..MaxIndex.
numItems	number of array elements to copy to *Dest*.

Description:

This method copies *numItems* elements from this array to a memory target. If the method is asked to copy more elements than there are, the *numItems* parameter is adjusted to the maximal number of elements possible without an exception beeing raised.

For each pointer in the source area CloneItem is called and the returned pointer is put into the *Dest* area. The Cloneitem method of TPointerArray just returns the source pointer unchanged, but a derived class may override Cloneitem to provide a deep copy mechanism.

Error Conditions:

If toIndex is > MaxIndex the method will raise a ERangeError exception, if range checking is on, or do nothing if range checking is off. If the *Dest* memory can hold less than the specified number of elements to copy a protection fault may result. If deep copies are made, EOutOfMemory may be raised if the heap allocator runs out of space.

Procedure **InvalidateItems**(atIndex, numItems: Cardinal); **(Method)**

See Also

Class: TPointerArray,
Visibility: public
Directives: override

Parameters:

atIndex index of the first element to invalidate (0..MaxIndex)

numItems number of elements to invalidate

Description:

This method calls the FreeItem method for every item in the passed range. FreeItem does nothing in the TPointerArray base class, but derived classes like TPStringArray and TPCharArray override it to release the memory allocated for an item. InvalidateItems is called by most methods that overwrite or delete elements before the elements are nuked. All items in the passed range will be set to Nil.

Error Conditions:

May cause a ERangeError exception, if range checking is on and the passed index is out of range. FreeItem may also cause an exception, if the passed item is not on the heap or was already disposed off somewhere else.

Procedure **LoadFromFile**(Const Filename: String); **(Method)**

See Also

Class: TPointerArray
Visibility: public
Directives: override

Parameters:

Filename name of the file to read

Description:

Loads the contents of the requested file into the array, which is redimensioned to fit the data. For this to work smoothly the file should have been created by the SaveToFile method of a pointer array object of the same type as this one. This method just creates a file stream and the calls LoadFromStream to do the work.

Error Conditions:

May raise a EInOutError exception if a file-related error occurs.

Procedure **LoadFromStream**(Stream: TStream); **(Method)**

See Also

Class: TPointerArray
Visibility: public
Directives: override

Parameters:

Stream stream to read the array data from. The stream must be open!

Description:

This method replaces the inherited method completely. It first reads the number of stored items from the stream (which must have been produced by the SaveToStream method of an object of the same type as this one), invalidates all pointers currently stored and resizes the array to the needed size. Then it calls LoadItemFromStream for all stored items and puts the returned pointers into the array. LoadItemFromStream does nothing for the base class TPointerArray and needs to be overridden by any derived class that wants to be able to store its items to a stream and read them back.

Caveat: The stream contains no identification of the stream type! Be careful to never mix streams generated by different types of objects or you may end up with a lot of garbage and unexpected exceptions!

Error Conditions:

The stream may raise an exception if it runs into problems. Which one depends on the type of stream.

Procedure **SaveToFile**(Const Filename: String); **(Method)**

See Also

Class: TPointerArray
Visibility: public
Directives: override

Parameters:

Filename name of the file to save to

Description:

This method overrides the method inherited from TBaseArray and replaces it completely. To make life simpler for the poor programmer (Self 8-)) this method creates a file stream and then calls SaveToStream. The file generated can be read back into an instance of the same class that created it via LoadFromFile.

Error Conditions:

May raise a EInOutError exception if a file-related error occurs.

Procedure **SaveToStream**(Stream: TStream); **(Method)**

See Also

Class: TPointerArray
Visibility: public
Directives: override

Parameters:

Stream stream to save the array to. The stream must be open!

Description:

Ths method completely replaces the inherited method. It first writes the number of items in the array (LongInt binary) to the stream and then calls the SaveItemToStream method for each item in the array. That virtual method does nothing for the TPointerArray base class but can be overridden by derived classes The produced stream can be read back into an array object of the same type with the LoadFromStream method.

Note that this method only writes the array data, not the object itself! There is also nothing written to the stream that would allow the identification of the array object type from the stream alone. It is your responsibility as programmer to make sure a stream written by one array object is not read into an object of a different type.

Error Conditions:

The stream may raise an exception if it runs into problems. Which one depends on the type of stream.

Type

TProgressReporter = Function(pos, max: LongInt; Var retain: Boolean): Boolean of Object;

This is a notification function that can be used with the LoadFromTextfile and SaveToTextfile methods of TPCharArray and TPStringArray. Its purpose is to be used to implement a progress indicator for the file worked on. It is passed the current position in the file and the total size of the file and can abort the processing by returning False. The value returned in *retain* determines, whether any partial data resulting from the processing up to the point of abort will be retained. The reporter also should call Application.ProcessMessages to keep the system responsive.

TRealArray (Class)

See Also Fields Methods Properties Exceptions

Unit: Arrays
Ancestor: TBaseArray

Description:

This is a specialized array class to store Borlands 6-byte floating-point numbers. It inherits all fields, methods, and properties from TBaseArray and may cause the same exceptions. The class overrides the Create constructor of the base class.

To make the usage of instances of this class more like standard Pascal arrays, the class has a default property Data that provides the normal array syntax (aValue := array[index]; and array[index] := value). This property is implemented with two new public methods. PutData and GetData, which you will normally not need to call directly. The class also has a comparison function assigned, so you can use the Sort and Find methods of the parent class directly.

TSingleArray (Class)

See Also Fields Methods Properties Exceptions

Unit: Arrays
Ancestor: TBaseArray

Description:

This is a specialized array class to store single precision floating-point numbers. It inherits all fields, methods, and properties from TBaseArray and may cause the same exceptions. The class overrides the Create constructor of the base class.

To make the usage of instances of this class more like standard Pascal arrays, the class has a default property Data that provides the normal array syntax (aValue := array[index]; and array[index] := value). This property is implemented with two new public methods. PutData and GetData, which you will normally not need to call directly. The class also has a comparison function assigned, so you can use the Sort and Find methods of the parent class directly.

Type

TSortOrder = (TS_NONE, TS_ASCENDING, TS_DESCENDING);

This enumerated type is used to specify the sort order of an array when invoking its Sort method. Also used by the SortOrder property.

The Array Flags

See also

TBaseArray, the base class of all the array classes, has a flag set of 16 flags (FFlags). Only three of these flags are used in the current implementation of the class library, the other 13 are free for your own uses. The set of flags is initialized in the Create constructor, dependend on the array class. See "Initial Array Flag Values" for a list of the default sets for each array class in the library. The following section describes, how the flags influence the behaviour of the array classes.

AF_OwnsData

This flag signifies two things:

1. The array object automatically makes copies of any data you put into it.
2. The array will dispose of an item as required when that item is overwritten or deleted or the whole object is destroyed.

This flag is set in the base type and all the numeric array types but has basically information value in these classes. In these arrays the data itself is stored in the array, putting it there automatically constitutes a copy operation and no special action is required to dispose of an item. You could clear the flag without any effects on the arrays behaviour.

The picture is different for arrays that store pointers or objects. The two string arrays TPStringArray and TPCharArray also have this flag set, for example. If you put a string or PChar into an array of these types, a duplicate of the string will be made and a pointer to the duplicate will be stored. The original you pass is left untouched and you can dispose of it anyway you like. If an item is deleted or replaced, the allocated memory for that item is released. The methods doing the actual low-level work here are CloneItem and FreeItem.

Caveat: If you clear the flag the behaviour of a string array would change drastically. It would now store a pointer to the original string you pass it so it becomes your responsibility to make any required copies. An item would also not get automatically freed if deleted or replaced (it may point to static data, for example). Using a string array without AF_OwnsData set has a **potential problem** you need to be aware of: some of the properties you can use to put data into the array need to make copies of the string to store to work porperly! If you assign a value to the AsInteger property, for example, the array method implementing the write access uses IntToStr to convert the number to a string an then stores the resulting string. If the flag is not set, the array will store the address of the temporary string created by IntToStr on the stack! The next time you access this item you will get an exception or garbage data because the string has long been gone to bit-heaven.

AF_AutoSize

If this flag is set, the array will automatically resize if you Insert or Delete items. If it is not set, items will fall of at the end of the array on insert; after a delete the now unused items at the end will be filled with 0. Append will grow the array independed of the flags setting, as will ReDim. Both are taken as explicit requests to grow the array.

You can change this flag anytime you like without any undue consequences. It is set by default for all array classes in this library.

AF_CanCompare

This flag determines the behaviour of the Sort and Find methods. If the flag is not set, these methods will return immediately. This is the only flag that is manipulated be a method other than Create. If you use the CompareProc property to assign a comparison function to an array, the flag will be set or cleared, depending whether the function pointer you pass is Nil or not. If you set this flag but have no comparison function assigned, any use of Sort or Find will result in an ECompUndefined exception!

AF_User1 .. AF_User13

These flags are not used by the class library and are free for your own extensions.

Unit Arrays

Constants Types Exceptions Classes Procedures Class Hierarchy

File: arrays.pas

Unit dependencies

The **Interface** uses SysUtils, and Classes.

The **Implementation** uses FastMem (another Unit in this toolbox). If the symbol **DOEVENTS** is declared, it will also use the Forms Unit (see below).

Delphi 2.0 Notes

The units in this collection are thread-safe in the sense that they do not use any global data that may suffer from concurrent access by different threads. The arrays objects themselves, however, have to be guarded in the usual way if you use the same array object in different threads. I see no possible problems with using different array objects in different threads. This whole field is completely untested, however!

Description

This Unit implements a base class for resizeable array types and a few specific derivatives for the common numeric types, as well as prototypes for array of pointers, PChars, and strings (see Class Hierarchy). The array classes in this unit are all limited to a maximum of 64Kbytes of data in Win16. This limitation does not apply to the Win32 version. The size of the stored items determines the maximal number of items. Errors will raise exceptions, *index overflow is only reported if range checking is on*! The **index range** of each class is **0..MaxIndex**, MaxIndex is a property of all class types. The arrays provide access to the stored elements via a Data property, which is the default for all specific classes (the base classes TBaseArray and TPointerArray have no default property). Methods are provided to insert, append or delete elements, to resize the array, sort it, search for an element, copy elements or the whole array etc.. See the description of TBaseArray for a more complete list of methods.

The classes have iterator methods similar to BP collections. These iterators can optionally call Application.ProcessMessages between rounds. This requires usage of the Forms Unit. Since this would involve a tremendous overhead for non-VCL projects the corresponding Uses clause and the iterator code calling Application.ProcessMessages is enclosed in $IFDEF **DOEVENTS** blocks. If DOEVENTS is defined, the Forms unit will be used. *DOEVENTS IS UNDEFINED BY DEFAULT!* You need to define this symbol in your project to make use of the ability to process messages inside iterator loops, and recompile this unit! The unit does not make any other use of VCL window objects.

Unit FastMem

Exported Procedures

File: fastmem.pas

Dependencies:

This Unit uses no other Units.

Description

This Unit contains a number of fast routines to fill, copy, and swap memory areas. You can use them as replacement for Delphis **FillChar** and **Move** routines; they are about 25-50% faster for Delphi 1.0. However, no noticeable speed difference exists in Delphi 2.0.

TBaseArray
TIntegerArray
TCardinalArray
TLongIntArray
TRealArray
TSingleArray
TDoubleArray
TExtendedArray
TPointerArray
TPCharArray
TPStringArray

NOT_FOUND

ECompUndefined
ETypeMismatch
EFileTooLarge

TArrayFlags
TArrayFlagsSet
TCompareProc
TIterator
TLocator
TIteratorMethod
TLocatorMethod
TProgressReporter
TSortOrder

MemFill
MemWordFill
MemDWordFill
MemMove
MemSwap