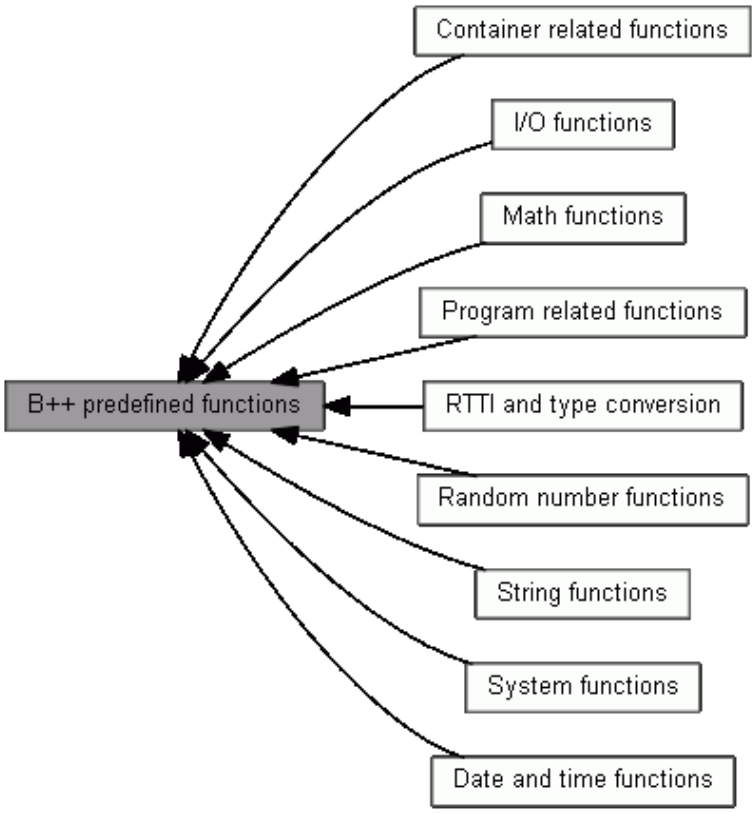
B++ predefined functions

This section describes predefined function available in each B++ program. More...

Collaboration diagram for B++ predefined functions:



Topics

Container related functions

The functions summarized here are used to handle the predefined container types `buffer`, `charbuffer`, `dictionary` and **`vector`**.

String functions

Math functions

Unless otherwise specified, the functions in this section return a result of type `double` and accept arguments of any type, if they are implicitely convertible to type `double`.

I/O functions

RTTI and type conversion

Date and time functions

Random number functions

Program related functions
System functions

Detailed Description

This section describes predefined function available in each B++ program.

The Functions are grouped according to their typical areas of use.

Note

- Predefined functions normally are not statically typed. Thus, when assigning values to the arguments, there are generally no implicit type conversions. With the exception of integral types, which are always compatible with each other, application code is responsible for correct conversion.
  If individual functions or function groups carry out such conversions, this is mentioned in their descriptions.
- The implementations of the functions check the validity of arguments passed at runtime and can trigger exceptions of type `string` in case of an error
- The types of parameters and return values are specified in the signatures.

B++ Reference © 2005-2026 by R.-Erik Ebert

All parts of B++ are provided under terms and conditions of MIT license.

Container related functions

B++ predefined functions

The functions summarized here are used to handle the predefined container types `buffer`, `charbuffer`, `dictionary` and **`vector`**. More...

Collaboration diagram for Container related functions:



Functions

`vector newvector (uint capacity=0, bool init=true)`
Creates a new vector.
`buffer newbuffer (const var assignment)`

Creates buffer as a copy of another buffer or string.

```
buffer newbuffer (uint size, byte fillbyte=0)
```

Creates buffer filled with a specified byte.

```
charbuffer newcbuffer (const var assignment)
```

Creates character buffer as a copy of another buffer or string.

```
charbuffer newcbuffer (uint size, int fillchar=0)
```

Creates character buffer filled with a specified character.

```
dictionary newdict (bool sorted=false)
```

Creates new empty dictionary.

```
bool dictcontains (const dictionary dict, const string key)
```

Checks for dictionary entry.

```
bool dicterase (dictionary dict, const string key)
```

Removes entry from dictionary.

```
void dictclear (dictionary dict)
```

Removes all entries from specified dictionary.

```
vector dictgetkeys (const dictionary dict)
```

Gets vector of all keys contained in specified dictionary.

```
const vector dictgetvalues (const dictionary dict)
```

Gets const vector of all values contained in specified dictionary.

```
uint size (const var container)
```

Gets number of elements in a container.

```
uint shrink (var container)
```

Shrinks capacity of container to current size.

```
uint vecsize (const vector vec)
```

Gets number of elements in a vector.

```
void qsort (var container, var callback=null)
```

Sorts content of a container using QuickSort algorithm.

```
uint foreach (var container, var callback,...)
```

Enumerates a container and calls callback for each of its items.

```
uint find (const var cont, const var searchVal, uint pos=0)
```

Finds position of value in a container.

```
uint rfind (const var cont, const var searchVal, uint pos=NPOS)
```

Finds last position of value in a container.

```
uint lbound (const var cont, uint startpos, uint endpos, const const var
val, const var comparer=null)
```

Finds lower bound for specified value in specified range of a sorted container.

```
uint lbound (const var cont, const var val, const var comparer=null)
```

Finds lower bound for specified value in a sorted container.

```
uint ubound (const var cont, uint startpos, uint endpos, const var val,
     const var comparer=null)
```

Finds upper bound for specified value in specified range of a sorted container.

```
uint ubound (const var cont, const var val, const var comparer=null)
```

Finds upper bound for specified value in a sorted container.

```
uint bsearch (const var cont, uint startpos, uint endpos, const var val,
     const var comparer=null)
```

Does a binary search for specified value in specified range of a sorted container.

```
uint bsearch (const var cont, const var val, const var comparer=null)
```

Does a binary search for specified value in a sorted container.

```
vector T (...)
```

Create a vector from n-tupel of any values.

```
vector Vec (...)
```

Create a vector from n-tupel of any values.

```
charbuffer CBuf (...)
```

Creates a character buffer from n-tupel of values.

```
var insert (const var dst, uint pos, const var insertion, uint cnt=1)
```

Inserts copy of value one or more times into copy of container at specified position.

```
var insrange (const var dst, uint pos, const var src, uint startpos=0,
     uint endpos=NPOS)
```

Inserts range of elements from one container into copy of another (or same) container of same type at specified position.

```
uint reserve (var cont, uint capacity)
```

Modifies capacity of container.

```
bool resize (var cont, uint size, var fill=null)
```

Modifies capacity of container.

```
vector vinsert (vector dst, int pos, var insertion, uint cnt=1)
```

Inserts value one or more times into vector at specified position.

```
var erase (const var cont, uint pos=0, uint cnt=NPOS)
```

Erases one or more element(s) from copy of container.

```
dictionary erase (const dictionary dict, const string key=null)
```

Erases specified element or all elements from copy of dictionary.

```
vector verase (const vector vec, uint pos=0, uint cnt=NPOS)
```

Erases one or more element(s) from copy of vector.

```
var swap (const var cont, uint pos1, uint pos2)
```

Exchanges specified elements in copy of a container.

```
var reverse (const var cont, uint from=0, uint to=NPOS)
```

Reverses order of elements in copy of a container.

```
vector vswap (const vector vec, uint pos1, uint pos2)
```

Exchanges specified elements in copy of vector.
`var move (const var cont, uint srcpos, uint dstpos)`
Moves an element inside copy of container.
`vector vmove (const vector vec, uint srcpos, uint dstpos)`
Moves an element inside copy of vector.
`var slice (const var cont, uint from=0, uint cnt=NPOS)`
Extracts a subsequence of elements from container.
`vector vslice (const vector vec, uint from=0, uint cnt=NPOS)`
Extracts a subsequenze of elements from vector.
`void clear (var container)`
Removes all elements from specified container.
`void vclear (vector vec)`
Removes all elements from specified vector.

Detailed Description

The functions summarized here are used to handle the predefined container types `buffer`, `charbuffer`, `dictionary` and `vector`.

Function Documentation

◆ bsearch() [1/2]

```
uint bsearch ( const var cont,
               const var val,
               const var comparer = null )
```

Does a binary search for specified value in a sorted container.

Parameters

container Container to search in. Possible types are **vector**, buffer or charbuffer.

val Value to search for. In case of buffer or charbuffer as type of *cont*, *val* should be of integral type.

comparer Compare callback. The callback may be either a function or a functor instance (an object defining **operator()**). The callback is invoked with two arguments (elements of cont to compare. It should return a negative number if left argument is lower than right argument, a positive number if left argument is greater than right argument or zero in case of equality.
If *comparer* is not specified, comparison is done using default rules.

Returns

Returns index of first element in *cont* equal to *val*.
If there is no such element, result is NPOS.

Note

Function assumes *cont* to be sorted in ascending order with respect to result of behaviour of *comparer*, otherwise result is undefined.

References **uint()**, **val**, and **var**.

◆ bsearch() [2/2]

```
uint bsearch ( const var cont,
               uint      startpos,
               uint      endpos,
               const var val,
               const var comparer = null )
```

Does a binary search for specified value in specified range of a sorted container.

Parameters

container Container to search in. Possible types are `vector`, `buffer` or `charbuffer`.

startpos Start index of search interval in *cont*.

endpos End index of search interval in *cont*. If *endpos* is greater or equal to size of *cont* search is done to end.

val Value to search for. In case of `buffer` or `charbuffer` as type of *cont*, *val* should be of integral type.

comparer Compare callback. The callback may be either a function or a functor instance (an object defining **operator()**). The callback is invoked with two arguments (elements of `cont` to compare. It should return a negative number if left argument is lower than right argument, a positive number if left argument is greater than right argument or zero in case of equality.
If *comparer* is not specified, comparison is done using default rules.

Returns

Returns index of first element in range [startpos,endpos] equal to *val*.
If there is no such element, result is NPOS.

> **Note**
> Function assumes range [startpos,endpos] to be sorted in ascending order with respect to result of behaviour of *comparer*, otherwise result is undefined.

References **uint()**, **val**, and **var**.

Referenced by **c16_to_char()**, **char_to_c16()**, **getCpIdFromName()**, and **BppScanner::getKeywordTypeFromString()**.

◆ `CBuf()`

```
charbuffer CBuf ( ... )
```

Creates a character buffer from n-tupel of values.

Function builds a character buffer from argument list.

Parameters

... List of values. Elements of argument list may be of integral types (interpreted as character in internal character set; values of character types are converted), `buffer`, `charbuffer` or `string`. Arguments of other types are ignored.

Returns

character buffer created from values

Usage example;

```
string str = "Hello World!";
charbuffer myCBuf = CBuf(str,0x20,"Let's ",'b','u','i','l',"d a buffer!",'\n');
print(string(myCBuf));
```

References **charbuffer()**.

◆ clear()

```
void clear ( var container )
```

Removes all elements from specified container.

Parameters

container container of type vector, charbuffer, buffer, string or dictionary to clear,

References **var**.

◆ dictclear()

```
void dictclear ( dictionary dict )
```

Removes all entries from specified dictionary.

Parameters

dict dictionary to remove entries from

> **Note**
>
> Simply an alias to **clear(var container)**.

References **dictionary()**.

◆ dictcontains()

```
bool dictcontains ( const dictionary dict,
                    const string     key )
```

Checks for dictionary entry.

Function checks if an entry for *key* is contained in *dict*.

Parameters

dict dictionary to search

key key to search for

Returns

`true` if found, otherwise `false`

References **dictionary()**.

Referenced by **LOGPALETTE::LOGPALETTE()**, **RGNDATA::RGNDATA()**, **TTPOLYCURVE::TTPOLYCURVE()**, **XMLElement::getAttribute()**, and **XMLElement::hasAttribute()**.

◆ dicterase()

```
bool dicterase ( dictionary  dict,
                 const string key )
```

Removes entry from dictionary.

Function removes entry of key *key* from *dict* if contained.

Parameters

dict dictionary to remove entry from

key key of entry to remove

Returns

`true` on success, otherwise `false`

References **dictionary()**.

Referenced by **XMLElement::removeAttribute()**.

◆ dictgetkeys()

```
vector dictgetkeys ( const dictionary dict )
```

Gets vector of all keys contained in specified dictionary.

Parameters

dict dictionary to get keys from

Returns

vector containing all keys of *dict*

Note

Returned vector contains copies of keys in dictionary. Therefore client code can modify strings in returned vector without sideeffects for *dict*.
The vector may be used instead of an iterator to walk through the dictionary.
The order of keys in returned vector depends on sorted state of dictionary. In a sorted dictionary keys are in alphabetical order, otherwise in order of insertion.

Usage example;

```
dictionary myDict = newdict();  // create unsorted dictionary
// add any entries here
// ...
vector keyvec = dictgetkeys(myDict);
uint n = size(keyvec);
for (uint i=0; i<n;++i)
   print(keyvec[i],":",myDict[keyvec[i]],"\n");
```

See also

newdict, **dictgetvalues**

References **dictionary()**, and **vector()**.

Referenced by **JSON::_writeJson()**, and **XMLNode::isEqualNode()**.

◆ dictgetvalues()

```
vector dictgetvalues ( const dictionary dict )
```

Gets const vector of all values contained in specified dictionary.

Gets vector of all values contained in specified dictionary.

Parameters

dict dictionary to get values from

Returns

constant vector containing all values of *dict*

Note

Returned vector may be used instead of an iterator to walk through the dictionary.
The order of values in returned vector conforms to order of keys.

Usage example;

```
dictionary myDict = newdict();  // create unsorted dictionary
// add any entries here
// ...
// make constant alias of myDict
// cdict is declared as const to prevent from making a copy.
const auto cdict = mkconst(myDict);
vector keyvec = dictgetkeys(cdict);
const vector valvec = dictgetvalues(cdict);
uint n = size(keyvec); // same as size(valvec)
for (uint i=0; i<n;++i)
   print(keyvec[i],":",valvec[i],"\n");
```

See also

newdict, **dictgetkeys**

Parameters

dict dictionary to get values from

Returns

vector containing all values of *dict*

Note

Returned vector may be used instead of an iterator to walk through the dictionary.
The Order of values in returned vector conforms to order of keys.
Modification of elements of result vector having reference types by client code will modify *dict* values too.

Usage example;

```
dictionary myDict = newdict();  // create unsorted dictionary
// add any entries here
// ...
vector keyvec = dictgetkeys(myDict);
```

```
vector valvec = dictgetvalues(myDict);
uint n = size(keyvec); // same as size(valvec)
for (uint i=0; i<n;++i)
   print(keyvec[i],":",valvec[i],"\n");
```

See also

newdict, **dictgetkeys**

References **dictionary()**, and **vector()**.

Referenced by **JSON::_writeJson()**, and **XMLNode::isEqualNode()**.

◆ erase() [1/2]

```
dictionary erase ( const dictionary dict,
                   const string     key = null )
```

Erases specified element or all elements from copy of dictionary.

Erases specified element or all elements from dictionary.

Parameters

dict dictionary to erase element(s) from.

key Key of element to erase, if not specified all elements are removed from dictionary.

Returns

modified copy of dictionary

Note

If key not found, result is simply a copy of dict.

Parameters

dict dictionary to erase element(s) from.

key Key of element to erase, if not specified all elements are removed from dictionary.

Returns

modified dictionary

Note

If key not found, *dict* is not modified.

References **dictionary()**.

◆ erase() [2/2]

```
var erase ( const var cont,
            uint     pos = 0,
            uint     cnt = NPOS )
```

Erases one or more element(s) from copy of container.

Erases one or more element(s) from container.

Parameters

cont container to erase element(s) from, may be of type **vector**, buffer, charbuffer or string.

pos Position index of first element to erase.

cnt Maximum number of elements to erase.

Returns

modified copy of container

Parameters

cont container to erase element(s) from, may be of type **vector**, buffer, charbuffer or string.

pos Position index of first element to erase.

cnt Maximum number of elements to erase.

Returns

modified container

References **uint()**, and **var**.

Referenced by **GDI::Region::combineRegions()**, and **GDI::EnhMetaFile::getDescription()**.

◆ find()

```
uint find ( const var cont,
            const var searchVal,
            uint      pos = 0 )
```

Finds position of value in a container.

Function searches in *cont* for value equal to *searchVal* and returns position of its first occurrence.

Parameters

cont	Container to search in, must be of type `vector`, `buffer`, `charbuffer` or `string`.
searchVal	Value to search for. If *cont* is of `string` type, *searchVal* may be of type `string` too or of an integral type (interpreted as character code).
pos	position to start search from.

Returns

position of first occurrence if found, otherwise NPOS

> **Note**
> Function does low level compare, i.e. it doesn't call equals- or compare-methods for objects.

References **uint()**, and **var**.

Referenced by **JSON::_writeJson()**, **XMLElement::getNextSiblingElement()**, **XMLElement::getPreviousSiblingElement()**, **XMLNode::getSibling()**, **XMLNode::insertBefore()**, **XMLNode::removeChild()**, and **XMLNode::replaceChild()**.

◆ foreach()

```
uint foreach ( var container,
               var callback,
                   ... )
```

Enumerates a container and calls callback for each of its items.

Parameters

container Container to enumarate. *container* may be of type `buffer`, `charbuffer`,`vector`, `dictionary` `string` or any class type defining its own **`foreach`** method of signature `uint foreach(var callback, ...)`.

callback Callback for enumeration. The callback may be either a function or a functor instance (an object defining **operator()**). The callback is invoked from **`foreach`** with at least two arguments (key and value of current container element). It should return an integer. If result of *callback* is zero (means `false`), enumeration is cancelled.

... If **`foreach`** called with more than two arguments, additional argumants are passed to `callback`.

Returns

Number of items enumerated.

Note

- If *container* is of `dictionary` type, first argument of *callback* is the key of item. For all other types, first argument is the index in *container*.
- Second argument of *callback* is the value of container item. If item value is of a reference type, *callback* can modify it if needed.
- *callback* shouldn't modify size of *container* of enumerated items. If it does so, it must return `false` to cancel enumeration.
- If *container* is an object, function calls **`foreach`** method of that object, passing *callback* and all additional arguments.
- Function allows (i.e. ignores) use of constant object for *container*. In this case *callback* should not modify content of container.

Usage example;

```
void main()
{
   // fill sorted dictionary from an unsorted one
   auto dict = { t:'t', e:'e',m:'m',p:'p',o:'o'}; // unsorted
   dictionary dict2 = newdict(true); // sorted
   // use dict2 for additional element
   foreach(dict,
           function(i,v,d) { d[i] = v; return true; },
           dict2);
   // callback for print container items
   auto iter = function(i, v) { print("I: ",i," V: ",v,"\n"); return true;};
   foreach(dict,iter); // list dict
   foreach(dict2,iter); // list dict2
```

```
    // enumerate a string
    foreach("Hello World",iter);
}
```

References **uint()**, and **var**.

Referenced by **BppXML::decodeXmlString()**, **BppXML::encodeXmlString()**, and **BppXML::initEntitymap()**.

◆ insert()

```
var insert ( const var dst,
             uint      pos,
             const var insertion,
             uint      cnt = 1 )
```

Inserts copy of value one or more times into copy of container at specified position.

Inserts value one or more times into container at specified position.

Parameters

dst container to copy and insert value into, may be of type **vector**, buffer, charbuffer or string.

pos Position in *dst* where insertion is done. If *pos* is greater than maximum valid index in *dst*, *insertion* is appended.

insertion Value to insert. If *dst* is of string type and *insertion* too and *cnt* is 1, string value of *insertion* is inserted at *pos*.
In all other cases either *dst* must be of type **vector** or *insertion* of integral type.
If *dst* is of type string or charbuffer and value of *insertion* is of any integral type, it is converted into internal character type.

cnt Number of elements to insert.

Returns

modified copy of container

Parameters

dst container to insert value into, may be of type **vector**, buffer, charbuffer or string.

pos Position in *dst* where insertion is done. If *pos* is greater than maximum valid index in *dst*, *insertion* is appended.

insertion Value to insert. If *dst* is of vector type and is itself is used for *insertion* argument, value is cloned before insertion.
If *dst* is of string type and *insertion* too and *cnt* is 1, string value of *insertion* is inserted at *pos*.
In all other cases *insertion* must be of integral type.
If *dst* is of type string or charbuffer and value of *insertion* is of any character type, it is converted into internal character type.

cnt Number of elements to insert.

Returns

modified container

References **uint()**, and **var**.

Referenced by **BppRefObject::BppRefObject()**.

- insrange()

```
var insrange ( const var dst,
               uint      pos,
               const var src,
               uint      startpos = 0,
               uint      endpos = NPOS )
```

Inserts range of elements from one container into copy of another (or same) container of same type at specified position.

Inserts range of elements from one container into another (or same) container of same type at specified position.

Parameters

dst container to insert values into copy of, may be of type `vector`, `buffer`, `charbuffer` or `string`.

pos Position in *dst* where insertion is done. If *pos* is greater than maximum valid index in *dst*, values are appended.

src Container to copy range from. Type of *src* must be same as of *dst*.

startpos First index of range to copy from *src*. If *startpos* is invalid, i.e. greater than last possible index in *src*, function does nothing.

endpos Last index of range to copy from *src*. If *endpos* is greater than last possible index in *src*, last possible index is used instead.
In *endpos* is less than *startpos*, insertion is done in reverse order.

Returns

new container as result of insertion

> **Note**
>
> This function should be used rather than `insert` to insert multiple elements from one container into onother one, because it needs only a single move operation in *dest*.

Parameters

dst container to insert values into, may be of type `vector`, `buffer`, `charbuffer` or `string`.

pos Position in *dst* where insertion is done. If *pos* is greater than maximum valid index in *dst*, values are appended.

src Container to copy range from. Type of *src* must be same as of *dst*.
If *src* and *dest* refer to same container and a part of range to copy is behind *pos*, temporary shallow copy of *src* is made before insertion operation.

startpos First index of range to copy from *src*. If *startpos* is invalid, i.e. greater than last possible index in *src*, function does nothing.

endpos Last index of range to copy from *src*. If *endpos* is greater than last possible index in *src*, last possible index is used instead.
In *endpos* is less than *startpos*, insertion is done in reverse order.

Returns

modified container

> **Note**
>
> This function should be used rather than `insert` to insert multiple elements from one container into onother one, because it needs only a single move operation in *dest*.

References **uint()**, and **var**.

◆ lbound() [1/2]

```
uint lbound ( const var cont,
              const var val,
              const var comparer = null )
```

Finds lower bound for specified value in a sorted container.

Parameters

container Container to search in. Possible types are `vector`, `buffer` or `charbuffer`.

val Value to search for. In case of `buffer` or `charbuffer` as type of *cont*, *val* should be of integral type.

comparer Compare callback. The callback may be either a function or a functor instance (an object defining **operator()**). The callback is invoked with two arguments (elements of `cont` to compare. It should return a negative number if left argument is lower than right argument, a positive number if left argument is greater than right argument or zero in case of equality.
If *comparer* is not specified, comparison is done using default rules.

Returns

Returns index of first element in *cont* not less than *val*.
If there is no such element, result is same as #size(cont).

> **Note**
>
> Function assumes *cont* to be sorted in ascending order with respect to result of behaviour of *comparer*, otherwise result is undefined.

References **uint()**, **val**, and **var**.

◆ lbound() [2/2]

```
uint lbound ( const var       cont,
              uint            startpos,
              uint            endpos,
              const const var val,
              const var       comparer = null )
```

Finds lower bound for specified value in specified range of a sorted container.

Parameters

container Container to search in. Possible types are `vector`, `buffer` or `charbuffer`.

startpos Start index of search interval in *cont*.

endpos End index of search interval in *cont*. If *endpos* is greater or equal to size of *cont* search is done to end.

val Value to search for. In case of `buffer` or `charbuffer` as type of *cont*, *val* should be of integral type.

comparer Compare callback. The callback may be either a function or a functor instance (an object defining **operator()**). The callback is invoked with two arguments (elements of `cont` to compare. It should return a negative number if left argument is lower than right argument, a positive number if left argument is greater than right argument or zero in case of equality.
If *comparer* is not specified, comparison is done using default rules.

Returns

Returns index of first element in range [startpos,endpos] not less than *val*.
If there is no such element, result is *endpos*+1.

Note

Function assumes range [startpos,endpos] to be sorted in ascending order with respect to result of behaviour of *comparer*, otherwise result is undefined.

References **uint()**, **val**, and **var**.

◆ `move()`

```
var move ( const var cont,
           uint      srcpos,
           uint      dstpos )
```

Moves an element inside copy of container.

Moves an element inside a container.

Parameters

cont Container containing element to move. Possible types are **vector**, buffer or charbuffer.

srcpos Position index of element to move.

dstpos Destination position index for move operation.

Returns

modified copy of container.

Note

Function throws exception of string type in case of invalid index or invalid type of *cont*.

Example

```
const vector arr = [1,2,3,4,5,6];
auto arr2 = move(arr,4,1);
// arr2 is now [1,5,2,3,4,6]
```

Parameters

cont Container containing element to move. Possible types are **vector**, buffer or charbuffer.

srcpos Position index of element to move.

dstpos Destination position index for move operation.

Returns

modified container.

Note

Function throws exception of string type in case of invalid index or invalid type of *cont*.

Example

```
vector arr = [1,2,3,4,5,6];
move(arr,4,1);
// arr is now [1,5,2,3,4,6]
```

References **uint()**, and **var**.

◆ newbuffer() [1/2]

```
buffer newbuffer ( const var assignment )
```

Creates buffer as a copy of another buffer or string.

Function makes a complete copy of *assignment* value.

Parameters

assignment Value to create new buffer from. Value must be of type `string`, `buffer` or `charbuffer`.

Returns

new buffer

> **Note**
>
> In UNICODE builds wide characters will be converted to single bytes discarding high byte of each character.

References **buffer()**, and **var**.

Referenced by **GDI::Bitmap::create()**, and **GDI::EnhMetaFile::getEnhMetaFileBits()**.

◆ newbuffer() [2/2]

```
buffer newbuffer ( uint size,
                   byte fillbyte = 0 )
```

Creates buffer filled with a specified byte.

Parameters

size Number of bytes in buffer to be created.

fillbyte Byte value to initialize buffer with.

Returns

new buffer

References **buffer()**, **size()**, and **uint()**.

◆ newcbuffer() [1/2]

```
charbuffer newcbuffer ( const var assignment )
```

Creates character buffer as a copy of another buffer or string.

Function makes a complete copy of *assignment* value.

Parameters

assignment Value to create new character buffer from. Value must be of type `string`, `buffer` or `charbuffer`.

Returns

new character buffer

> **Note**
>
> In UNICODE builds if *assignment* is of type `buffer` each byte of *assignment* is converted into wide character setting zero for high byte.

References **charbuffer()**, and **var**.

Referenced by **GDI::MetaDC::MetaDC()**, **GDI::EnhMetaFile::getDescription()**, and **GDI::DC::getTextFace()**.

◆ newcbuffer() [2/2]

```
charbuffer newcbuffer ( uint size,
                        int  fillchar = 0 )
```

Creates character buffer filled with a specified character.

Parameters

size Number of bytes in buffer to be created.

fillchar Character value to initialize buffer with.

Returns

new character buffer

References **charbuffer()**, **size()**, and **uint()**.

◆ newdict()

```
dictionary newdict ( bool sorted = false )
```

Creates new empty dictionary.

Parameters

sorted Flag for sorted dictionary. If `true` entries in dictionary will be ordered by keys, otherwise new elements are appended to end.

Returns

new dictionary

Note

- Value of *sorted* flag affects results of dictgetkeys and dictgetvalues functions. If set these functions return entries key ordered, otherwise in order of insertion.
- In DOS and WindowsCE builds element access will be faster in sorted than non-sorted dictionaries.

References **dictionary()**.

Referenced by **Bp2Dictionary_ctor()**, **BppXML::fillAttributes()**, **XMLElement::getAttributes()**, and **XMLElement::setAttribute()**.

◆ `newvector()`

```
vector newvector ( uint capacity = 0,
                   bool init = true )
```

Creates a new vector.

Function creates a new vector and optionally initializes its elements.

Parameters

capacity initial capacity of vector

init Flag for vector initialization. If `true`, vector is initialized, i.e. vector size is set to be same as *capacity* and all elements are set to `null`.

Returns

vector object

Note

Setting capacity and initialization on creation may increase speed because there is no need to resize vector at runtime.

Usage example:

```
vec = newvector(10);
for (i=1;i<=10;++i) vec[i-1] = i*i;
```

See also

insert,**verase**, **reserve**

References **uint()**, and **vector()**.

Referenced by **StatusBar::applyPaneSizes()**, and **XMLNode::foreach()**.

◆ qsort()

```
void qsort ( var container,
             var callback = null )
```

Sorts content of a container using QuickSort algorithm.

Function behaves like qsort function of C library.

Parameters

container Container to sort. *container* may be of type `buffer`, `charbuffer` or **`vector`**.

callback Compare callback. The callback may be either a function or a functor instance (an object defining **operator()**). The callback is invoked from qsort algorithm with two arguments (elements of `container` to compare. It should return a negative number if left argument is lower than right argument, a positive number if left argument is greater than right argument or zero in case of equality.
If *callback* is not specified, comparison is done using default rules.

Usage example;

```
class comparer
{
   comparer() {}
   operator()(a,b) { return b-a; } // makes descending order
};

void main()
{
   var vec = [3,1,5,7,2,0,9];
   var sz = size(vec);
   // sort ascending using anonymous function for callback
   qsort(vec,function(a,b) { return a-b; });
   // prints 0123579
   for (var i=0;i<sz;++i) print(vec[i]);
   print("\n");
   // sort descending using comparer object
   qsort(vec,new comparer());
   // prints 9753210
   for (i=0;i<sz;++i) print(vec[i]);
}
```

References **var**.

◆ reserve()

```
uint reserve ( var  cont,
               uint capacity )
```

Modifies capacity of container.

Function tries to reserve memory for *capacity* elements in *cont*. It doesn't remove any data, i.e. if current size (number of elements) is greater than *capacity* it does nothing.

Parameters

cont Container to modify. Possible container types are `vector`, `buffer` or `charbuffer`.

capacity New capacity of container (number of elements).

Returns

Capacity of container.

References **uint()**, and **var**.

◆ resize()

```
bool resize ( var  cont,
              uint size,
              var  fill = null )
```

Modifies capacity of container.

Function resizes *cont*. to *size* elements. If new size is less than current size, elements are removed from end of container. If new size is greater than current, new empty elements will be appended to *cont*

Parameters

cont Container to resize. Possible container types are `vector`, `buffer` or `charbuffer`.

size New size of container (number of elements). @parem fill Value to use for initialization of new (appended) elements. If *cont* is of type `buffer` or `charbuffer`, *fill* must be of an integral type.
If *fill* is omitted, `null` is used for default value of `vector` elements and `0` for elements of `buffer` or `charbuffer`.

Returns

true on success.

References **size()**, **uint()**, and **var**.

◆ reverse()

```
var reverse ( const var cont,
              uint      from = 0,
              uint      to = NPOS )
```

Reverses order of elements in copy of a container.

Reverses order of elements of a container.

Parameters

cont Container containing elements to reverse. Possible types are `vector`, `buffer` or `charbuffer`.

from Position index of first element in range to reverse.

to Position index of second element in range to reverse. If *to* is greater as last valid indec in container, last indec is used instead.

Returns

modified copy of container.

Note

Function throws exception of string type in case invalid argument types or if *from* > *to*.

Example;

```
const vector arr = [1,2,3,4,5,6];
vector arr2 = reverse(arr,1,3);
// arr2 is now [1,4,3,2,5,6]
```

Parameters

cont Container containing elements to reverse. Possible types are `vector`, `buffer` or `charbuffer`.

from Position index of first element in range to reverse.

to Position index of second element in range to reverse. If *to* is greater as last valid indec in container, last indec is used instead.

Returns

modified container.

Note

Function throws exception of string type in case invalid argument types or if *from* > *to*.

Example;

```
vector arr = [1,2,3,4,5,6];
reverse(arr,1,3);
// arr is now [1,4,3,2,5,6]
```

References **uint()**, and **var**.

Referenced by **_createDictIterator()**.

◆ rfind()

```
uint rfind ( const var cont,
             const var searchVal,
             uint      pos = NPOS )
```

Finds last position of value in a container.

Function searches backwards in *cont* for value equal to *searchVal* and returns position of its last occurrence.

Parameters

cont	Container to search in, must be of type **vector**, buffer or charbuffer.
searchVal	Value to search for.
pos	Position to start backwards search from.

Returns

Position of last occurrence if found, otherwise NPOS

> **Note**
> Function does low level compare, i.e. it doesn't call equals- or compare-methods for objects.

References **uint()**, and **var**.

◆ shrink()

```
uint shrink ( var container )
```

Shrinks capacity of container to current size.

Parameters

container	Value of type buffer, charbuffer or **vector**.

Returns

Capacity (number of elements) of *container*

References **uint()**, and **var**.

◆ size()

```
uint size ( const var container )
```

Gets number of elements in a container.

Parameters

container Any container, i.e. value of type `buffer`, `charbuffer`, `vector` or `dictionary`, or `string`.

Returns

number of elements in *container*

References **uint()**, and **var**.

Referenced by **_GetMem()**, **XMLDocumentFragment::_serialize()**, **Bp2Vector_erase()**, **Bp2Vector_insert()**, **Bp2Vector_slice()**, **fread()**, **GDI::EnhMetaFile::fromBits()**, **fwrite()**, **newbuffer()**, **newcbuffer()**, **newstring()**, **XMLNode::normalize()**, **BppValue::operator==()**, **BppDictionaryIO::readValue()**, **resize()**, **RECT::setSize()**, **WinBase::setSize()**, **WinBase::setVirtualSize()**, **JSON::stringify()**, and **BppDictionaryIO::writeValue()**.

◆ slice()

```
var slice ( const var cont,
            uint     from = 0,
            uint     cnt = NPOS )
```

Extracts a subsequence of elements from container.

Parameters

cont container to build subsequence from. Possible types are `vector`, `buffer` or `charbuffer`.

from Index of first element to extract in *vec*.

cnt Number of elements to extract. If *cnt* greater than number of elements behind *from*, all elements to end of *cont* are included into result.

Returns

New container of same type as *cont* containing selected Elements of *vec*.

References **uint()**, and **var**.

◆ swap()

```
var swap ( const var cont,
           uint      pos1,
           uint      pos2 )
```

Exchanges specified elements in copy of a container.

Exchanges specified elements of a container.

Parameters

cont Container containing elements to exchange. Possible types are **vector**, buffer or charbuffer.

pos1 Position index of first element.

pos2 Position index of second element.

Returns

modified copy of container.

Note

Function throws exception of string type in case of invalid index or invalid type of *container*.

Example;

```
const vector arr = [1,2,3,4,5,6];
vector arr2 = swap(arr,4,1);
// arr2 is now [1,5,3,4,2,6]
```

Parameters

cont Container containing elements to exchange. Possible types are **vector**, buffer or charbuffer.

pos1 Position index of first element.

pos2 Position index of second element.

Returns

modified container.

Note

Function throws exception of string type in case of invalid index or invalid type of *container*.

Example;

```
vector arr = [1,2,3,4,5,6];
swap(arr,4,1);
// arr is now [1,5,3,4,2,6]
```

References **uint()**, and **var**.

◆ T()

```
vector T ( ... )
```

Create a vector from n-tupel of any values.

Function creates a vector from argument list.

Parameters

... list of any values

Returns

vector created from values

> **Note**
>
> Other than in vector constructor ([...]) arguments of this function are not restricted to literals.

Usage example;

```
string str = "Hello World!\n";
vector vec = T(str,"This is B++\n",strlen(str);
uint n = size(vec);
for (uint i=0;i<n;++i) print(vec[i]);
```

References **vector()**.

Referenced by **Rational::Rational()**, and **EditCtrl::getInsertionPosition()**.

◆ ubound() [1/2]

```
uint ubound ( const var cont,
              const var val,
              const var comparer = null )
```

Finds upper bound for specified value in a sorted container.

Parameters

container Container to search in. Possible types are **vector**, buffer or charbuffer.

val Value to search for. In case of buffer or charbuffer as type of *cont*, *val* should be of integral type.

comparer Compare callback. The callback may be either a function or a functor instance (an object defining **operator()**). The callback is invoked with two arguments (elements of cont to compare. It should return a negative number if left argument is lower than right argument, a positive number if left argument is greater than right argument or zero in case of equality.
If *comparer* is not specified, comparison is done using default rules.

Returns

Returns index of first element in *cont* greater than *val*.
If there is no such element, result is same as #size(cont).

Note

Function assumes *cont* to be sorted in ascending order with respect to result of behaviour of *comparer*, otherwise result is undefined.

References **uint()**, **val**, and **var**.

◆ ubound() [2/2]

```
uint ubound ( const var cont,
              uint      startpos,
              uint      endpos,
              const var val,
              const var comparer = null )
```

Finds upper bound for specified value in specified range of a sorted container.

Parameters

container Container to search in. Possible types are `vector`, `buffer` or `charbuffer`.

startpos Start index of search interval in *cont*.

endpos End index of search interval in *cont*. If *endpos* is greater or equal to size of *cont* search is done to end.

val Value to search for. In case of `buffer` or `charbuffer` as type of *cont*, *val* should be of integral type.

comparer Compare callback. The callback may be either a function or a functor instance (an object defining **operator()**). The callback is invoked with two arguments (elements of `cont` to compare. It should return a negative number if left argument is lower than right argument, a positive number if left argument is greater than right argument or zero in case of equality.
If *comparer* is not specified, comparison is done using default rules.

Returns

Returns index of first element in range [startpos,endpos] greater than *val*.
If there is no such element, result is *endpos*+1.

Note

Function assumes range [startpos,endpos] to be sorted in ascending order with respect to result of behaviour of *comparer*, otherwise result is undefined.

References **uint()**, **val**, and **var**.

◆ vclear()

```
void vclear ( vector vec )
```

Removes all elements from specified vector.

Parameters

vec vector to clear,

> **Note**
>
> Simply an alias to **clear(var container)**.

References **vector()**.

Referenced by **XMLNode::insertNode()**, and **BppXML::parse()**.

◆ Vec()

```
vector Vec ( ... )
```

Create a vector from n-tupel of any values.

Function is a synonym to **T(...)** function.

Parameters

... list of any values

Returns

vector created from values

See also

T(...)

References **vector()**.

◆ vecsize()

```
uint vecsize ( const vector vec )
```

Gets number of elements in a vector.

Parameters

vec vector to get size from

Returns

number of elements in *vec*

See also

size

References **uint()**, and **vector()**.

◆ verase()

```
vector verase ( const vector vec,
                uint         pos = 0,
                uint         cnt = NPOS )
```

Erases one or more element(s) from copy of vector.

Erases one or more element(s) from vector.

Parameters

vec vector to erase element(s) from.

pos Position index of first element to erase.

cnt Maximum number of elements to erase.

Returns

modified copy of vector

Note

Simply an alias of erase(const var container,pos,cnt).

Parameters

vec vector to erase element(s) from.

pos Position index of first element to erase.

cnt Maximum number of elements to erase.

Returns

modified vector

Note

Simply an alias of erase(var container,pos,cnt).

References **uint()**, and **vector()**.

Referenced by **XMLNode::normalize()**, and **XMLNode::removeChildAt()**.

◆ vinsert()

```
vector vinsert ( vector dst,
                 int    pos,
                 var    insertion,
                 uint   cnt = 1 )
```

Inserts value one or more times into vector at specified position.

Parameters

dst	vector to insert value into.
pos	Position in *cont* where insertion is done. If *pos* is negative or greater than maximum valid index in *cont*, *insertion* is appended.
insertion	Value (of any type) to insert. If *vec* itself is used for *insertion* argument, value is cloned before insertion.
cnt	Number of elements to insert.

Returns

modified vector

Note

Simply an alias of insert(var dst,pos,insertion,cnt).

References **uint()**, **var**, and **vector()**.

Referenced by **XMLElement::_appendElements()**, and **XMLNode::insertNode()**.

◆ vmove()

```
vector vmove ( const vector vec,
               uint         srcpos,
               uint         dstpos )
```

Moves an element inside copy of vector.

Moves an element inside a vector.

Parameters

vec vector containing element to move.

srcpos Position index of element to move.

dstpos Destination position index for move operation.

Returns

modified copy of vector.

Note

Simply an alias of move(const var cont, srcpos, dstpos).

Parameters

vec vector containing element to move.

srcpos Position index of element to move.

dstpos Destination position index for move operation.

Returns

modified vector.

Note

Simply an alias of move(var cont, srcpos, dstpos).

References **uint()**, and **vector()**.

◆ vslice()

```
vector vslice ( const vector vec,
                uint         from = 0,
                uint         cnt = NPOS )
```

Extracts a subsequenze of elements from vector.

Parameters

vec vector to build subsequence from.

from Index of first element to extract in *vec*.

cnt Number of elements to extract. If *cnt* greater than number of elements behind *from*, all elements to end of *vec* are icluded into result.

Returns

New vector containing selected Elements of *vec*.

Note

Simply an alias of slice(var cont, from, cnt).

References **uint()**, and **vector()**.

◆ vswap()

```
vector vswap ( const vector vec,
               uint        pos1,
               uint        pos2 )
```

Exchanges specified elements in copy of vector.

Exchanges specified elements of vector.

Parameters

vec vector containing elements to exchange.

pos1 Position index of first element.

pos2 Position index of second element.

Returns

modified copy of vector vector.

Note

Simply an alias of swap(const var cont ,pos1, pos2).

Parameters

vec vector containing elements to exchange.

pos1 Position index of first element.

pos2 Position index of second element.

Returns

modified vector.

Note

Simply an alias of swap(var cont ,pos1, pos2).

References **uint()**, and **vector()**.

B++ Reference © 2005-2026 by R.-Erik Ebert

All parts of B++ are provided under terms and conditions of MIT license.

String functions

B++ predefined functions

Collaboration diagram for String functions:



Functions

`string newstring (const var assignment)`
Create string as a copy of another string or buffer.
`string newstring (const string formatstr,...)`
Create string from format specification.
`string newstring (uint size, int fillchar=' ')`
Creates string filled with a specified character.
`uint strlen (const string str)`
Get length of a string.
`int strcmp (const string str1, const string str2)`
Compare function (case sensitive).
`int stricmp (const string str1, const string str2)`
Compare function (case insensitive).
`vector strsplit (const string str, string delimiter, bool storeEmptyTokens=false, bool storeDelimitChar=false)`
String tokenize function.
`uint strfind (const string str, const var search, uint start=0)`
Find position of substring or character.
`string strerase (const string str, uint pos, uint cnt=NPOS)`
Erase part from copy of string.
`string strinsert (const string str, uint pos, const string insstr)`
Insert string into copy of another one.
`string strinsert (string str, uint pos, int insc, uint cnt=1)`
Insert character(s) into string.
`string strreplace (const string str, const var search, const var replacement)`
Replace part of copy of string.
`string substr (const string str, uint start, uint len=NPOS)`
Get substring from string.
`string strupr (const string str)`
Make string upper case.
`string strlwr (const string str)`
Make string lower case.

Detailed Description

Function Documentation

◆ newstring() [1/3]

```
string newstring ( const string formatstr,
                                ... )
```

Create string from format specification.

Function builds new string from format specification and any number of additional arguments.

Parameters

formatstr Format specification. Value must be of `string` type. The format string has similary syntax as in C - see **BppString** description.

... More arguments to replace variables in *formatstring*.

Returns

new string

> **Note**
>
> Elements of variable argument list normally should be of simple types, i.e. characters, numbers or strings. For all other types return value of **BppValue::addr()** method is used for variable value.

◆ newstring() [2/3]

```
string newstring ( const var assignment )
```

Create string as a copy of another string or buffer.

Function makes a complete copy of *assignment* value.

Parameters

assignment Value to create new string from. Value must be of type `string`, `buffer` or `charbuffer`. In case of `buffer` each byte contained in *assignment* is assumed to be a single byte character.

Returns

new string

References **var**.

Referenced by **XMLCDATASection::_serialize()**, **XMLComment::_serialize()**, **XMLDocumentType::_serialize()**, **XMLElement::_serialize()**, **XMLNode::_serialize()**, **XMLProcessingInstruction::_serialize()**, **BppXML::initEntitymap()**, **XMLDOMException::operator string()**, and **JSON::stringify()**.

◆ newstring() [3/3]

```
string newstring ( uint size,
                   int  fillchar = ' ' )
```

Creates string filled with a specified character.

Parameters

size Number of characters in string to be created (without trailing '\0' character.

fillchar Character to initialize string with.

Returns

new string

References **size()**, and **uint()**.

◆ strcmp()

```
int strcmp ( const string str1,
             const string str2 )
```

Compare function (case sensitive).

Parameters

str1 left `string` for compare

str2 right `string` for compare

Returns

- < 0 if *str1* is lower than *str2*
- 0 if *str1* equals to *str2*
- > 0 if *str1* is greater than *str2*

Referenced by **GetProcAddress()**.

◆ strerase()

```
string strerase ( const string str,
                  uint         pos,
                  uint         cnt = NPOS )
```

Erase part from copy of string.

Erase part of string.

Function deletes one or more characters from a copy of string.

Parameters

str source string

pos index of first character to delete

cnt Number of characters to delete. If value is greater than number of characters behind *pos*, all characters to end of string will be deleted.

Returns

modified copy of *str*

Note

Function modifies argument *str*. The return value is the modified string itself, not a copy of it.

Function deletes one or more characters from string.

Parameters

str string to erase character(s) from

pos index of first character to delete

cnt Number of characters to delete. If value is greater than number of characters behind *pos*, all characters to end of string will be deleted.

Returns

modified copy of *str*.

References **uint()**.

◆ strfind()

```
uint strfind ( const string str,
               const var    search,
               uint         start = 0 )
```

Find position of substring or character.

Function searches *str* for substring or character *search* and returns position of its first occurrence.

Parameters

str string to search

search string or character to search for

start character position to start search from.

Returns

position of first occurrence if found, otherwise NPOS

References **uint()**, and **var**.

Referenced by **XMLElement::_appendElements()**, and **BppXML::parse()**.

◆ stricmp()

```
int stricmp ( const string str1,
              const string str2 )
```

Compare function (case insensitive).

Parameters

str1 left `string` for compare

str2 right `string` for compare

Returns

- < 0 if *str1* is lower than *str2*
- 0 if *str1* equals to *str2*
- > 0 if *str1* is greater than *str2*

Referenced by **_findLibrary()**.

◆ strinsert() [1/2]

```
string strinsert ( const string str,
                   uint         pos,
                   const string insstr )
```

Insert string into copy of another one.

Insert string into another one.

Function inserts value of *insstr* into *str* at position *pos*.

Parameters

str const string to copy and modify

pos Index position where *insstr* should be inserted. Use length of *str* for *pos* to append *insstr* value.

insstr string to insert

Returns

modified copy of *str*

Function inserts value of *insstr* into *str* at position *pos*.

Parameters

str string to modify

pos Index position where *insstr* should be inserted. Use length of *str* for *pos* to append *insstr* value.

insstr string to insert

Returns

modified string

Note

Function modifies argument *str*. The return value is the modified string itself, not a copy of it.

References **uint()**.

Referenced by **XMLDocumentType::_serialize()**, and **XMLElement::_serialize()**.

◆ strinsert() [2/2]

```
string strinsert ( string str,
                   uint   pos,
                   int    insc,
                   uint   cnt = 1 )
```

Insert character(s) into string.

Function inserts *cnt* characters *insc* into *str* at position *pos*.

Parameters

str string to modify

pos Index position for insertion. Use length of *str* for *pos* to append characters.

insc character value to insert

cnt number of characters to insert

Returns

modified string

Note

Function modifies argument *str*. The return value is the modified string itself, not a copy of it.

Function inserts *cnt* characters *insc* into a copy of *str* at position *pos*.

Parameters

str const string to modify

pos Index position for insertion. Use length of *str* for *pos* to append characters.

insc character value to insert

cnt number of characters to insert

Returns

modified copy string

References **uint()**.

◆ strlen()

```
uint strlen ( const string str )
```

Get length of a string.

Parameters

str `string` value to get length of

Returns

number of characters in *str* (without trailing zero)

References **uint()**.

Referenced by **_TmpLocale::_TmpLocale()**, **BppString::BppString()**, **XMLElement::_appendElements()**, **XMLElement::_getMatchMode()**, **XMLDocumentType::_serialize()**, **XMLElement::_serialize()**, **ButtonCtrl::create()**, **EditCtrl::create()**, **XMLElement::getChildElements()**, **XMLElement::getElementsByTagName()**, **GDI::DC::getTextExtent()**, **BppXML::parse()**, **searchpath()**, **str2wcsBuf()**, **GDI::DC::textOut()**, and **GDI::DC::textOutExt()**.

◆ strlwr()

```
string strlwr ( const string str )
```

Make string lower case.

Function converts letters contained in copy of *str* to lower case.

Parameters

str const string to convert

Returns

converted string

Note

Function behaves like C-function of same name, i.e. result depends on locale settings.

See also

setlocale, **strupr**

Function converts letters contained in *str* to lower case.

Parameters

str string to convert

Returns

converted string

Note

Function modifies content of *str* and returns reference to modified string.
It behaves like C-function of same name, i.e. result depends on locale settings.

See also

setlocale, **strupr**

◆ strreplace()

```
string strreplace ( const string str,
                    const var    search,
                    const var    replacement )
```

Replace part of copy of string.

Replace part of string.

Function replaces all occurrences of *search* by value of *replacement*.

Parameters

str	const string to copy and modify
search	character or substring to be replaced
replacement	character or string to be used for replacement

Returns

result of replacement

Note

Both, *search* and *replacement*, must be either single characters or strings.

Function replaces all occurrences of *search* by value of *replacement*.

Parameters

str	string to modify
search	character or substring to be replaced
replacement	character or string to be used for replacement

Returns

current object

Note

Both, *search* and *replacement*, must be either single characters or strings.

References **var**.

Referenced by **JSON::_writeJson()**, **JSON::_writeval()**, **BppXML::decodeXmlString()**, **BppXML::encodeXmlString()**, and **BppXML::parse()**.

◆ strsplit()

```
vector strsplit ( const string str,
                  string       delimiter,
                  bool         storeEmptyTokens = false,
                  bool         storeDelimitChar = false )
```

String tokenize function.

Function splits *str* into tokens using delimiters defined in *delimiter*.

Parameters

str	`string` to tokenize
delimiter	Set of characters (type `string`) to be used for delimiters
storeEmptyTokens	If `true`, empty tokens are stored in result too.
storeDelimitChar	If `true`, delimiter character between tokens are stored in result too.

Returns

vector containing tokens

Note

- If *str* is empty, result vector is empty too.
- Delimiters (if *storeDelimitChar* set) are atored in result as character codes (i.e. of type **int**).
- If *storeEmptyTokens* is set and last character of *str* is a delimiter, empty string is appended to result.

Usage example;

```
void listvec(vector v)
{
   int n = size(v);
   print("size: ",n,"\n");
   for (int i=0;i<n;++i) print(""+v[i],"\n");
   print("-----\n");
}

main() {
   string str = "This is a  sample string.";
   listvec(strsplit(str," ."));
   listvec(strsplit(str," .",false,true));
   listvec(strsplit(str," .",true));
   listvec(strsplit(str," .",false,true));
   listvec(strsplit(str," .",true,true));
}
```

References **vector()**.

Referenced by **StatusBar::create()**.

◆ strupr()

```
string strupr ( const string str )
```

Make string upper case.

Function converts letters contained in copy of *str* to upper case.

Parameters

str const string to convert

Returns

converted string

Note

Function behaves like C-function of same name, i.e. result depends on locale settings.

See also

setlocale, **strlwr**

Function converts letters contained in *str* to upper case.

Parameters

str string to convert

Returns

converted string

Note

Function modifies content of *str* and returns reference to modified string.
It behaves like C-function of same name, i.e. result depends on locale settings.

See also

setlocale, **strlwr**

◆ substr()

```
string substr ( const string str,
                uint         start,
                uint         len = NPOS )
```

Get substring from string.

Function returns substring of *str*, beginning at position *start* and of maximum *len* characters length.

Parameters

str string to get substrung from

start Start position of substring.

len Maximum size of substring. If len is omitted, all characters of current string (beginning at *start*) are included in result.

Returns

new string containing specified substring

References **uint()**.

Referenced by **XMLElement::getChildElements()**, **XMLElement::getElementsByTagName()**, and **BppXML::parse()**.

B++ Reference © 2005-2026 by R.-Erik Ebert

All parts of B++ are provided under terms and conditions of MIT license.

Math functions

B++ predefined functions

Unless otherwise specified, the functions in this section return a result of type `double` and accept arguments of any type, if they are implicitely convertible to type `double`. More...

Collaboration diagram for Math functions:



Functions

double sin (double x)
Calculates sin(*x*).
double cos (double x)
Calculates cos(*x*).
double tan (double x)
Calculates tan(*x*).

`double atan (double x)`
Calculates arctan(*x*).
`double sqrt (double x)`
Calculates square root.
`double log (double x)`
Calculates natural logarithm ln(*x*).
`double exp (double x)`
Calculates exponential function $\mathbf{e}^x$.
`double pow (double x, double y)`
Calculates exponential function x^y.
`var abs (var x)`
Calculates absolute value of *x*.

Detailed Description

Unless otherwise specified, the functions in this section return a result of type `double` and accept arguments of any type, if they are implicitely convertible to type `double`.

Function Documentation

◆ `abs()`

```
var abs ( var x )
```

Calculates absolute value of *x*.

Parameters

x argument

Returns

absolute value of *x*

Note

- Return type is same as type of *x*.
- The argument can be of any numeric type (not of a character type). An implicit conversion is not done.

References **var**.

Referenced by **Rational::fromDouble()**, **Rational::gcd()**, **Bp2WinBase::handleMessage()**, **Rational::lcm()**, and **Rational::simplify()**.

◆ atan()

```
double atan ( double x )
```

Calculates arctan(*x*).

Parameters

x argument

Returns

arcus tangens of *x* (radiant)

◆ cos()

```
double cos ( double x )
```

Calculates cos(*x*).

Parameters

x argument (radiant)

Returns

cosinus of *x*

◆ exp()

```
double exp ( double x )
```

Calculates exponential function $\mathbf{e}^{x}$.

Parameters

x argument

Returns

e to the power *x*

◆ log()

```
double log ( double x )
```

Calculates natural logarithm ln(*x*).

Parameters

x argument, must be positive

Returns

natural logarithm of *x*

◆ pow()

```
double pow ( double x,
             double y  )
```

Calculates exponential function x^y.

Parameters

x base

y exponent

Returns

x to the power of *y*

Note

If exponent *y* isn't a whole number value, *x* must not be negative.

◆ sin()

```
double sin ( double x )
```

Calculates sin(*x*).

Parameters

x argument (radiant)

Returns

sinus of *x*

◆ sqrt()

```
double sqrt ( double x )
```

Calculates square root.

Parameters

x argument, must not be negative

Returns

square root of *x*

◆ tan()

```
double tan ( double x )
```

Calculates tan(*x*).

Parameters

x argument (radiant)

Returns

tangens of *x*

B++ Reference © 2005-2026 by R.-Erik Ebert

All parts of B++ are provided under terms and conditions of MIT license.

I/O functions

B++ predefined functions

Collaboration diagram for I/O functions:



Common I/O functions

Note

Functions of this section are available in all B++ builds.

```
void print (...)
```

Print list of values to default output device.

```
FILE fopen (const string filename, const string mode)
```

Open a file.

```
FILE freopen (const string filename, const string mode, FILE fp)
```

Reopen a file stream with different file and/or mode.

```
int fclose (FILE fp)
```

Close a file.

```
int fflush (const FILE fp)
```

Flushes buffer of file stream.

```
int feof (const FILE fp)
```

Check for end of file.

```
int fseek (FILE fp, int offset, int origin)
```

Moves the file pointer to a specified position.

```
int ftell (const FILE fp)
```

Gets current position in specified file.
`int getc (FILE fp)`
Read single character from opened file.
`int getc ()`
Read single character input object installed for virtual machine.
`bool kbhit ()`
Checks if any key was pressed.
`int putc (int c, FILE fp)`
Write single character to opened file.
`string gets ()`
Read string from standard input.
`string fgets (FILE fp)`
Read string from file.
`int fputs (const string text, FILE fp)`
Write string to file.
`var freadval (FILE fp)`
Read value from file.
`bool fwriteval (const var value, FILE fp)`
Write value to file.
`uint fread (buffer readbuf, FILE fp)`
Read file.
`uint fread (buffer readbuf, uint cnt, FILE fp)`
Read specified number of bytes from file.
`uint fread (buffer readbuf, uint size, uint cnt, FILE fp)`
Read specified number of elements of fixed size from file.
`uint fwrite (const buffer buf, FILE fp)`
Write buffer to file.
`uint fwrite (const buffer buf, uint cnt, FILE fp)`
Write specified number of bytes to file.
`uint fwrite (const buffer buf, uint size, uint cnt, FILE fp)`
Write specified number of fixed sized elements to file.

Wide character I/O functions

Note

Functions of this section are available in Windows builds only.

`int getwc (FILE fp)`

	Read single wide character from opened file.
`int putwc (int c, FILE fp)`	
	Write single wide character to opened file.
`string fgetws (FILE fp)`	
	Read wide character string from file.
`int fputws (const string text, FILE fp)`	
	Write wide character string to file.

Detailed Description

Function Documentation

◆ fclose()

```
int fclose ( FILE fp )
```

Close a file.

Function closes a file previously opened by fopen.

Parameters

fp descriptor of file to be closed

Returns

0 on success, otherwise non-zero value

> **Note**
> Function must not be used with default streams stdin, stdout or stderr.

Referenced by **BppFileCharReader::~BppFileCharReader()**, **BppFileStreamInput::~BppFileStreamInput()**, **BppFileStreamPrinter::~BppFileStreamPrinter()**, **build_exe()**, **cpy_file()**, **DllEntry::create()**, **fakePclose()**, **BppDictionaryIO::readModule()**, **searchDirList()**, and **BppDictionaryIO::writeModule()**.

◆ feof()

```
int feof ( const FILE fp )
```

Check for end of file.

Function checks if end of specified file is reached.

Parameters

fp descriptor of file

Returns

non-zero value if end of file is reached, otherwise 0

Referenced by **BppFileCharReader::_get()**, **cpy_file()**, **BppFileStreamInput::getString()**, and **BppDictionaryIO::readString()**.

◆ fflush()

```
int fflush ( const FILE fp )
```

Flushes buffer of file stream.

Function behaves like same named C-function.

Parameters

fp descriptor of file to be flushed

Returns

0 on success, otherwise non-zero value

Note

If *fp* is null, all files opened in write mode are flushed.

◆ fgets()

```
string fgets ( FILE fp )
```

Read string from file.

Function reads a string from an opened file, beginning at current position. If returns on new line or if end of file is reached.

Parameters

fp descriptor of file to read from

Returns

string read

Note

Trailing line feed character is not part of result.
Function should be used only for reading from files opened in text mode.

See also

gets, **fread**

◆ fgetws()

```
string fgetws ( FILE fp )
```

Read wide character string from file.

Function reads a wide character string from an opened file, beginning at current position. If returns on new line or if end of file is reached.

Parameters

fp descriptor of file to read from

Returns

string read

Note

Trailing line feed character is not part of result.
Function should be used only for reading from files opened in text mode.

See also

fread

◆ fopen()

```
FILE fopen ( const string filename,
             const string mode )
```

Open a file.

Function tries to open file *filename* in specified open mode. It behaves exactly as same-named C-function.

Parameters

filename Name of file to open. The name may contain relative or absolute peth specification.

mode Specification of open mode, may contain following elements:

- r : Open an existing file for read only.
- w : Create file and open it for write. If specified file already exists it is overwritten.
- a : Open a file for append data. If file doesn't exist it will be created.
- a+ : Open a file for read and append data. If file doesn't exist it will be created.
- r+ : Open existinng file for read and write.
- w+ : Create file and open it for read and write. If specified file already exists it is overwritten.
- t : Open file in text mode (may be combined with a, r, r+, w and w+).
- b : Open file in binary mode (may be combined with a, r, r+, w and w+).

Returns

File descriptor on success, otherwise `null`.

Note

Files opened by this function should be closed using fclose if no longer needed.

See also

fclose

Referenced by **DllEntry::create()**.

◆ `fputs()`

```
int fputs ( const string text,
            FILE         fp )
```

Write string to file.

Function writes *text* into opened file *fp*, beginning at current position.

Parameters

text string to write

fp descriptor of file to write into

Returns

on success code of last character written, otherwise EOF (-1).

Note

Function should be used for writing to files opened in text mode only.

See also

fwrite

◆ fputws()

```
int fputws ( const string text,
             FILE         fp )
```

Write wide character string to file.

Function writes *text* into opened file *fp*, beginning at current position.

Parameters

text string to write

fp descriptor of file to write into

Returns

on success code of last character written, otherwise EOF (-1).

Note

Function should be used only for writing to files opened in text mode.

See also

fwrite

◆ fread() [1/3]

```
uint fread ( buffer readbuf,
             FILE   fp )
```

Read file.

Function appends contents of file specified by *fp* to *readbuf*.

Parameters

readbuf Destination buffer.

fp Descriptor of opened file to read from.

Returns

number of bytes read

Note

- Contents of *fp* is read from current file position to end of file.
- Capacity of *readbuf* is enlarged if neccessary.

References **buffer()**, and **uint()**.

Referenced by **cpy_file()**, **DllEntry::create()**, **detectFileCp()**, **BppDictionaryIO::readBytes()**, **BppDictionaryIO::readPtr()**, and **BppDictionaryIO::readString()**.

◆ fread() [2/3]

```
uint fread ( buffer readbuf,
             uint   cnt,
             FILE   fp )
```

Read specified number of bytes from file.

Function appends up to *cnt* bytes from contents of file specified by *fp* to *readbuf*.

Parameters

readbuf Destination buffer.

cnt Maximum number of bytes to read.

fp Descriptor of opened file to read from.

Returns

number of bytes read

Note

- Capacity of *readbuf* is enlarged if neccessary.
- Reading starts at current file position

References **buffer()**, and **uint()**.

◆ fread() [3/3]

```
uint fread ( buffer readbuf,
             uint   size,
             uint   cnt,
             FILE   fp )
```

Read specified number of elements of fixed size from file.

Function appends up to *cnt* elements of *size* bytes length from contents of file specified by *fp* to *readbuf*.

Parameters

readbuf Destination buffer.

size size of single element to read

cnt Maximum number of elements to read.

fp Descriptor of opened file to read from.

Returns

number of elements read

Note

- Capacity of *readbuf* is enlarged if neccessary.
- Reading starts at current file position

References **buffer()**, **size()**, and **uint()**.

◆ freadval()

```
var freadval ( FILE fp )
```

Read value from file.

Function reads any B++ value from opened binary file, beginning at current position.

Parameters

fp Descriptor of file to read from. The file must be in format as created by fwriteval function.

Returns

on success value read, otherwise `null`

Note

Client code should assign returned value to a variable for further use. Please refer to **fwriteval** documentation for a usage example.

See also

fwriteval

References **var**.

◆ `freopen()`

```
FILE freopen ( const string filename,
               const string mode,
               FILE         fp )
```

Reopen a file stream with different file and/or mode.

Function closes *fp* tries to reopen it for file *filename* in specified open mode. It behaves exactly as same-named C-function.

Parameters

filename Name of file to open. The name maycontain relative or absolute peth specification.

mode Specification of open mode, may contain following elements:

- r : Open an existing file for read only.
- w : Create file and open it for write. If specified file already exists it is overwritten.
- a : Open a file for append data. If file doesn't exist it will be created.
- a+ : Open a file for read and append data. If file doesn't exist it will be created.
- r+ : Open existinng file for read and write.
- w+ : Create file and open it for read and write. If specified file already exists it is overwritten.
- t : Open file in text mode (may be combined with a, r, r+, w and w+).
- b : Open file in binary mode (may be combined with a, r, r+, w and w+).

fp Descriptor of file to reopen.

Returns

File descriptor on success, otherwise `null`.

Note

Files opened by this function should be closed using fclose if no longer needed.

See also

fclose

◆ `fseek()`

```
int fseek ( FILE fp,
            int  offset,
            int  origin )
```

Moves the file pointer to a specified position.

Function behaves like C-function of same name.

Parameters

fp descriptor of file

offset Number of bytes from *origin*

origin Initial position, must be one of SEEK_SET (0), SEEK_CUR (1) or SEEK_END (2).

Returns

0 on success, non zero value otherwise

Referenced by **build_exe()**, and **detectFileCp()**.

◆ ftell()

```
int ftell ( const FILE fp )
```

Gets current position in specified file.

Function behaves like C-function of same name.

Parameters

fp descriptor of file

Returns

Current position of file pointer in *fp*.

◆ fwrite() [1/3]

```
uint fwrite ( const buffer buf,
              FILE         fp )
```

Write buffer to file.

Function writes complete contents of *buf* into file, starting at current file position.

Parameters

buf Buffer containing contents to write.

fp Descriptor of opened file to write to.

Returns

number of bytes written

References **buffer()**, and **uint()**.

Referenced by **build_exe()**, **cpy_file()**, **BppDictionaryIO::writeBytes()**, **BppDictionaryIO::writePtr()**, and **BppDictionaryIO::writeString()**.

◆ fwrite() [2/3]

```
uint fwrite ( const buffer buf,
              uint         cnt,
              FILE         fp )
```

Write specified number of bytes to file.

Function writes up to *cnt* bytes from *buf* to file specified by *fp*.

Parameters

buf Buffer containing contents to write.

cnt Maximum number of bytes to write.

fp Descriptor of opened file to write to.

Returns

number of bytes written

Note

Function writes into file starting at current file position.

References **buffer()**, and **uint()**.

◆ fwrite() [3/3]

```
uint fwrite ( const buffer buf,
              uint         size,
              uint         cnt,
              FILE         fp )
```

Write specified number of fixed sized elements to file.

Function writes up to *cnt* elements of *size* bytes length from *buf* to file specified by *fp*.

Parameters

buf Buffer containing contents to write.

size size of e single element

cnt Maximum number of elements to write.

fp Descriptor of opened file to write to.

Returns

number of elements written

Note

Function writes into file starting at current file position.

References **buffer()**, **size()**, and **uint()**.

◆ `fwriteval()`

```
bool fwriteval ( const var value,
                 FILE      fp )
```

Write value to file.

Function writes any B++ value into opened binary file, beginning at current position.

Parameters

value value to write

fp descriptor of file to write into

Returns

`true` on success, otherwise `false`

Note

Using **fwriteval** and **freadval** functions allows to serialize and deserialize any B++ data structure in a simple way. Espacially that includes complete object hierarchies too. Any cross references between objects inside a hierarchy are saved on serialization and resolved on deserialization.

Example:

```
// Serialization example using fwriteval and freadval.
// first we define a list entry class

class ListEntry
{
   static _maxID = 0;
   _a;
   _b;
   _ID;
   _next;
}

// ctor
ListEntry::ListEntry(a,b)
{
   _ID = ++_maxID;
   _a=a;
   _b=b;
   _next = null;
}

ListEntry::getA() { return _a; }

// print contents of entry
ListEntry::pr()
{
   print("ID: ",_ID," ",getA(), " ",_b,"\n");
}

ListEntry::getNext() { return _next; }

ListEntry::setNext(next) { _next = next; }

ListEntry::append(v)
```

```
{
   if (_next)
      _next->append();
   else
   {
      _next = v;
   }
}

// dtor
ListEntry::~ListEntry()
{
   print("destroyed ",_ID,"\n");
}

main()
{
   var lst = new ListEntry(-1,0);
   for(var i=0;i<9;i++)               // append more objects to lst
   {
      lst->append(new ListEntry(i,i+1));
   }
   for(var p = l;p;p=p->getNext())    // print contents
      p->pr();
   var fp=fopen("obj.dat","wb");      // open file for write
   fwriteval(lst,fp);              // write entire list to file
   fclose(fp);
   lst = null;                     // free list
   fp=fopen("obj.dat","rb");       // open file for read
   lst=freadval(fp);               // read contents and assign to lst
   fclose(fp);
   for(p = lst;p;p=p->getNext())   // print contents
      p->pr();
}
```

See also

freadval

References **var**.

◆ getc() [1/2]

```
int getc ( )
```

Read single character input object installed for virtual machine.

Function reads a single character from input object installed for virtual machine.

Returns

character code (-1 if nothing read)

> **Note**
>
> By default virtual machine uses `stdin` device to get the character. However, another input object (**IBppInput** implementation) may be installed by client code.

Referenced by **BppFileCharReader::_get()**.

◆ getc() [2/2]

```
int getc ( FILE fp )
```

Read single character from opened file.

Function reads a single-byte character from specified file.

Parameters

fp file descriptor

Returns

character code (-1 if end of file reached)

◆ gets()

```
string gets ( )
```

Read string from standard input.

Function by default reads string from stdin device (usually the console). It returns on new line (i.i.erase. Enter key). The trailing line feed is not contained in return value.

Returns

string read

> **Note**
>
> Function normally works on systems having a console only. However, an input object (**IBppInput** implemantation) may be installed for virtual machine to get input from another device than `stdin`.

See also

fgets

◆ getwc()

```
int getwc ( FILE fp )
```

Read single wide character from opened file.

Function reads a wide character from specified file.

Parameters

fp file descriptor

Returns

character code (-1 if end of file reached)

◆ kbhit()

```
bool kbhit ( )
```

Checks if any key was pressed.

Returns

true if a key with a character code was pressed.

Note

Behavior of function differs depending on OS where program is running on:

- For MS-DOS and Windows console applications, it behaves like `kbhit` C-function.
- For Windows applications without console, function is emulated using `GetAsyncKeyState` function.
  Please note that there is no character available on `stdin` when function returns *true* in that case.
- On UNIX platforms it's emulated too. It behaves as for Windows console application if current process is connected to a terminal window. Otherwise behavior is undefined.

Referenced by **keyPressed()**.

◆ `print()`

```
void print ( ... )
```

Print list of values to default output device.

Parameters

... Comma separated list of values. Output depends on value type as follows:

- `string` : Value is written to output without any changes.
- all integer types (`sbyte ... ulong`): Value is written as unformatted integer or unsigned integer (depending on type).
- `double` or `float` : Value is written as unformatted double of automatic notation (like g-format pf printf).
- `intptr` : Value is written as address in hexadecimal representation.
- all other types : Type name and address of value are written to output.

Note

- Output is done using **IBppPrinter** implementation set for default printer of virtual machine.
- All arguments are assumed to be `const`.

Referenced by **JSON::_writeJson()**.

◆ putc()

```
int putc ( int  c,
           FILE fp )
```

Write single character to opened file.

Function writes a single-byte character to specified file.

Parameters

c character to write

fp file descriptor

Returns

character code written (-1 on error)

◆ putwc()

```
int putwc ( int  c,
          FILE fp )
```

Write single wide character to opened file.

Function writes a wide character to specified file.

Parameters

c character to write

fp file descriptor

Returns

character code written (-1 on error)

B++ Reference © 2005-2026 by R.-Erik Ebert

All parts of B++ are provided under terms and conditions of MIT license.

RTTI and type conversion

B++ predefined functions

Collaboration diagram for RTTI and type conversion:



Functions

bool	**bool** (const **var** value)
	Convert value to boolean value.
sbyte	**sbyte** (const **var** value)
	Converts value to signed byte.
char	**char** (const **var** value)
	Converts value to signed char.
byte	**byte** (const **var** value)
	Converts value to unsigned byte (unsigned char).
uchar	**uchar** (const **var** value)
	Converts value to unsigned char.
char8_t	**char8_t** (const **var** value)
	Converts value to unsigned char.
short	**short** (const **var** value)
	Converts value to short integer.
ushort	**ushort** (const **var** value)

	Converts value to unsigned short integer.
char16_t	char16_t (const var value)
	Converts value to UTF-16le character.
int	int (const var value)
	Converts value to integer.
uint	uint (const var value)
	Converts value to unsigned integer.
char32_t	char32_t (const var value)
	Converts value to UTF-32le character.
wchar_t	wchar_t (const var value)
	Converts value to wide character.
u8char	u8char (const var value)
	Converts value to UTF-8 unicode character.
long	long (const var value)
	Converts value to long integer.
ulong	ulong (const var value)
	Converts value to unsigned long integer.
intptr	intptr (const var value)
	Converts value to integer of size same as address pointers of system.
uintptr	uintptr (const var value)
	Converts value to unsigned integer of size same as address pointers of system.
float	float (const var value)
	Converts value to **float**.
double	double (const var value)
	Converts value to **double**.
string	string (const var value)
	Converts value to string.
buffer	buffer (const var value)
	Converts value to buffer.
charbuffer	charbuffer (const var value)
	Converts value to charbuffer.
vector	vector (const var value)
	Converts value to **vector**.
dictionary	dictionary (const var value)
	Converts value to dictionary.
buffer	str2utf8buf (const string src, bool raw=false)
	Converts string to UTF-8 buffer.

string utf8buf2str (const buffer src, bool raw=false)	
	Converts UTF-8 string to string.
int gettype (const var value)	
	Get type id of argument value.
string gettypename (int typeId, bool prefixed=false)	
	Get human readable name of a data type.
string getclassname (const var value, bool qualified=false)	
	Get class name of object or class.
Class getclass (const var value)	
	Gets class of object or finds class for specified name.
object newobject (Class classtype,...)	
	Creates new object instance of specified class.
var getsymbol (const string name, bool isclass=false)	
	Gets a global symbol (variable, function or class) by name at runtime.
object dynamic_cast (const Class desttype, var expr)	
	Do dynamic cast to specified destination class.
object dynamic_cast (const string classname, var expr)	
	Do dynamic cast to destination class specified by name.
function findmethod (const var objectOrClass, const string name, bool isstatic=false)	
	Searches for a method of object or class, available in current call context.
bool isconst (var expr)	
	Checks if specified value is const.
var mkconst (var expr, bool setconst=true)	
	Modifies constant state of exprerssion result.

Detailed Description

Function Documentation

◆ bool()

```
bool bool ( const var value )
```

Convert value to boolean value.

Function converts argument of any type to **bool** value.

Parameters

value Value to convert.

Returns

boolean representation for *value*.
Result is `false` for numeric types of value 0, the `null` value and all reference type having data pointer NULL (i.e. without any data). In all other cases result value is `true`, if argument is not an object. For argument of object type its conversion operators **bool** or **int** are called if defined.

Note

For conversion of objects, function searches for **bool** operator first, then for **int** operator.

See also

size, **gettype**, **int**

References **var**.

◆ buffer()

```
buffer buffer ( const var value )
```

Converts value to `buffer`.

Function tries to convert argument into value of `buffer` type.

Parameters

value Value to convert. The value must be of type **`int`**, **`float`**, **`double`**, `buffer`, `charbuffer` or `object`.

Returns

value converted to `buffer`

Note

- For numeric types (integral numbers, characters, **`float`** and **`double`**) returned buffer contains a bytewise copy of argument and has same langth as native size of argument type.
- In case of a `charbuffer` or `string` all characters of argument, including trailing zero, are copied to result buffer. Size of result is number of characters copied multipled by number of bytes used for a single character.
- In case of a `buffer` function simply returns a reference (not a copy!) to argument. If argument is `const`, result is `const` too.
- Values of `object` type must have a `buffer` conversion operator.
- In case of invalid argument type exception of string type is thrown. Use **gettype** to check type of value.

See also

size, **gettype**

References **buffer()**, and **var**.

Referenced by **String::a_buf()**, **Struct::addBuffer()**, **buffer()**, **GDI::Bitmap::create()**, **fread()**, **fread()**, **fread()**, **GDI::EnhMetaFile::fromBits()**, **fwrite()**, **fwrite()**, **fwrite()**, **GDI::Bitmap::getBitmapBits()**, **Struct::getData()**, **CStruct::getDataBuffer()**, **GDI::EnhMetaFile::getEnhMetaFileBits()**, **EditCtrl::getInsertionPosition()**, **GDI::DC::getMiterLimit()**, **newbuffer()**, **newbuffer()**, **BppWinMsgPrinter::operator()()**, **GDI::Bitmap::setBitmapBits()**, **GDI::DC::setMiterLimit()**, **str2utf8buf()**, **GDI::DC::stretchDIBits()**, **utf8buf2str()**, and **String::w_buf()**.

◆ byte()

```
byte byte ( const var value )
```

Converts value to unsigned byte (unsigned char).

Function tries to convert argument into value of `byte` type (8 bits unsigned integer).

Parameters

value Value to convert. The value must be of any number type, `string`, `buffer` or `object`.

Returns

value converted to `int`

Note

- A `string` value should contain string representation of a decimal, octal, hexadecimal or dual integer number. Hexadecimal numbers are prefixed by '0x'or '0X', dual numbers by '0b' or '0B'. An octal number starts with 0, followed by a digit between 1 and 7. In all other cases a decimal number is assumed.
- If *value* is of `buffer` type, the buffer must contain a 8, 16, 32 or 64 bits integer value (i.e. size of buffer must be 1, 2, 4 or 8).
- In case of `object` for *value*, the specified object must define an `byte` conversion operator.
- In case of invalid argument type exception of string type is thrown. Use **gettype** to check type of value.

See also

size, **gettype**

References **var**.

◆ `char()`

```
char char ( const var value )
```

Converts value to signed char.

Function tries to convert argument into value of **char** type (single byte character).

Parameters

value Value to convert. The value must be of any number or character type, `string`, `buffer` or `object`.

Returns

value converted to **char**

Note

- A `string` value should contain string representation of a decimal, octal, hexadecimal or dual integer number. Hexadecimal numbers are prefixed by '0x'or '0X', dual numbers by '0b' or '0B'. An octal number starts with 0, followed by a digit between 1 and 7. In all other cases a decimal number is assumed.
- If *value* is of `buffer` type, the buffer must contain a 8, 16, 32 or 64 bits integer value (i.e. size of buffer must be 1, 2, 4 or 8).
- In case of `object` for *value*, the specified object must define a **char** conversion operator.
- In case of invalid argument type exception of string type is thrown. Use **gettype** to check type of value.

See also

size, **gettype**

References **var**.

Referenced by **Bp2String_a_buf()**, and **Bp2Struct_addInt8()**.

◆ `char16_t()`

`char16_t char16_t ( const var value )`

Converts value to UTF-16le character.

Function tries to convert argument into value of **char16_t** type (16 bits character).

Parameters

value Value to convert. The value must be of any number or character type, `string`, `buffer` or `object`.

Returns

value converted to **int**

Note

- A `string` value should contain string representation of a decimal, octal, hexadecimal or dual integer number. Hexadecimal numbers are prefixed by '0x'or '0X', dual numbers by '0b' or '0B'. An octal number starts with 0, followed by a digit between 1 and 7. In all other cases a decimal number is assumed.
- If *value* is of `buffer` type, the buffer must contain a 8, 16, 32 or 64 bits integer value (i.e. size of buffer must be 1, 2, 4 or 8).
- In case of `object` for *value*, the specified object must define a **char16_t** conversion operator.
- In case of invalid argument type exception of string type is thrown. Use **gettype** to check type of value.

See also

size, **gettype**

References **var**.

◆ `char32_t()`

`char32_t char32_t ( const var value )`

Converts value to UTF-32le character.

Function tries to convert argument into value of `char32_t` type (32 bits character).

Parameters

value Value to convert. The value must be of any number or character type, `string`, `buffer` or `object`.

Returns

value converted to `int`

Note

- A `string` value should contain string representation of a decimal, octal, hexadecimal or dual integer number. Hexadecimal numbers are prefixed by '0x'or '0X', dual numbers by '0b' or '0B'. An octal number starts with 0, followed by a digit between 1 and 7. In all other cases a decimal number is assumed.
- If *value* is of `buffer` type, the buffer must contain a 8, 16, 32 or 64 bits integer value (i.e. size of buffer must be 1, 2, 4 or 8).
- In case of `object` for *value*, the specified object must define a `char32_t` conversion operator.
- In case of invalid argument type exception of string type is thrown. Use **gettype** to check type of value.

See also

size, **gettype**

References **var**.

◆ `char8_t()`

```
char8_t char8_t ( const var value )
```

Converts value to unsigned char.

Function tries to convert argument into value of **char8_t** type (unsigned single byte character).

Parameters

value Value to convert. The value must be of any number or character type, `string`, `buffer` or `object`.

Returns

value converted to **char**

Note

- **char8_t** is simply an alias of `uchar` type.
- A `string` value should contain string representation of a decimal, octal, hexadecimal or dual integer number. Hexadecimal numbers are prefixed by '0x'or '0X', dual numbers by '0b' or '0B'. An octal number starts with 0, followed by a digit between 1 and 7. In all other cases a decimal number is assumed.
- If *value* is of `buffer` type, the buffer must contain a 8, 16, 32 or 64 bits integer value (i.e. size of buffer must be 1, 2, 4 or 8).
- In case of `object` for *value*, the specified object must define a **char8_t** conversion operator.
- In case of invalid argument type exception of string type is thrown. Use **gettype** to check type of value.

See also

size, **gettype**

References **var**.

◆ `charbuffer()`

```
charbuffer charbuffer ( const var value )
```

Converts value to charbuffer.

Function tries to convert argument into value of charbuffer type.

Parameters

value Value to convert. The value must be of a numeric integral type, a character type, **float**, **double**, buffer, charbuffer, string or object.

Returns

value converted to charbuffer

Note

- For numeric types (except **intptr** and uintptr) returned charbuffer contains a string representation of argument. Floating point values are converted using g format specification.
- Values of types **intptr** and uintptr are interpreted as addresses and result in a hexadecimal represantation like printf C-function using format specification f.
- Values of character types are converted into internal character type and result in a character buffer containing single character.
- In case of a buffer argument all bytes of argument are interpreted as single character values, independent of current character size.
- In case of a charbuffer function simply returns a reference (not a copy!) to argument. If argument is const, result is const too.
- Values of object type must have a charbuffer conversion operator.
- In case of invalid argument type exception of string type is thrown. Use **gettype** to check type of value.

See also

size, **gettype**

References **charbuffer()**, and **var**.

Referenced by **GDI::MetaDC::MetaDC()**, **Struct::addCBuffer()**, **String::c_buf()**, **CBuf()**, **charbuffer()**, **newcbuffer()**, and **newcbuffer()**.

◆ dictionary()

```
dictionary dictionary ( const var value )
```

Converts value to `dictionary`.

Function tries to convert argument into value of `dictionary` type.

Parameters

value Value to convert. The value must be of type `vector`, `dictionary` or `object`.

Returns

value converted to `vector`

Note

- In case of a `vector` result contains vector elements as values where keys are string representations of element indexes in vector.
- In case of a `dictionary` function simply returns a reference (not a copy!) of argument. If argument is `const`, result is `const` too.
- Values of `object` type must have a `dictionary` conversion operator.
- In case of invalid argument type exception of string type is thrown. Use **gettype** to check type of value.

See also

size, **gettype**

References **dictionary()**, and **var**.

Referenced by **FIXED::FIXED()**, **LOGPALETTE::LOGPALETTE()**, **RGNDATA::RGNDATA()**, **TTPOLYCURVE::TTPOLYCURVE()**, **XFORM::XFORM()**, **CStruct::applyData()**, **CStruct::create()**, **Dictionary::dict()**, **dictclear()**, **dictcontains()**, **dicterase()**, **dictgetkeys()**, **dictgetvalues()**, **dictionary()**, **erase()**, **BppXML::fillAttributes()**, **XMLElement::getAttributes()**, **XMLNode::getAttributes()**, **__SubstructProxy::getData()**, **CStruct::getData()**, **GDI::EnhMetaFile::getDescription()**, **getenv()**, **GDI::DC::getPathData()**, **CStruct::getStructDesc()**, **FIXED::getStructDesc()**, **LOGPALETTE::getStructDesc()**, **RGNDATA::getStructDesc()**, **TTPOLYCURVE::getStructDesc()**, **XFORM::getStructDesc()**, **XMLNode::isEqualNode()**, and **newdict()**.

◆ `double()`

```
double double ( const var value )
```

Converts value to **double**.

Function tries to convert argument into value of **double** type.

Parameters

value Value to convert. The value must be of type **int**, **float**, **double**, string or buffer.

Returns

value converted to **double**

Note

- A string value must be string representation of a **double** number in format [*whitespace*] [*sign*] [digits] [.digits] [{d | D | e | E }[sign]digits] where a whitespace is a space or tab character and sign is either plus (+) or minus (-).
- If *value* is of buffer type, the buffer must contain a 32 bits (float) or 64 bits (double) value (i.e. size of buffer must be 4 or 8).
- In case of object for *value*, the specified object must define a **double** or **float** conversion operator.
- In case of invalid argument type exception of string type is thrown. Use **gettype** to check type of value.

See also

size, **gettype**

References **var**.

◆ dynamic_cast() [1/2]

```
object dynamic_cast ( const Class desttype,
                      var         expr )
```

Do dynamic cast to specified destination class.

Function behaves very similarly to **dynamic_cast** operator in C++.
First it evaluates expression *expr*. If evaluation results in an object of class type *desttype* or derived class, it returns the object, otherwise `null`.

Parameters

desttype class to cast *expr* to

expr expression of any type

Returns

object of class *desttype* (or derived) on success, otherwise `null`

Note

- Other than **dynamic_cast** operator in C++ there is no exeption thrown if *expr* is not an object. In this case function simply returns `null`. Use **gettype** to check a value for its basic type.
- Function preserves constant state of *expr*, i.e., if result of *expr* is a constant object, result of function is `const` too. In that case object is cloned if result is assigned to a non-const variable.

Example:

```
class Base
{
   Base();
}
Base::Base() { }

class Derived : Base
{
   Derived();
}
Derived::Derived() { }

main(;obj1, obj1, o, i)
{
   obj1 = new Base();
   obj2 = new Derived();
   o = dynamic_cast(Base, obj1);      // returns obj1
   o = dynamic_cast(Base, obj2);      // returns obj2
   o = dynamic_cast(Derived, obj1);   // returns null
   o = dynamic_cast(Derived, obj2);   // returns obj2
   i = dynamic_cast(Base, 17);        // returns null (not 0)
}
```

See also

gettype, **getclass**, **dynamic_cast(string,var)**

References **var**.

Referenced by **GDI::MemoryDC::MemoryDC()**, **Rational::Rational()**, **XMLNode::XMLNode()**, **XMLElement::_appendElements()**, **StatusBar::create()**, **XMLElement::getChildElementByAttribute()**, **XMLDocument::getDoctype()**, **XMLDocument::getDocumentElement()**, **XMLElement::getFirstChildElement()**, **XMLElement::getLastChildElement()**, **GDI::Bitmap::isDIB()**, **MessageBox()**, **SetParent()**, and **Window::setStatusBar()**.

◆ dynamic_cast() [2/2]

```
object dynamic_cast ( const string classname,
                      var          expr )
```

Do dynamic cast to destination class specified by name.

Function behaves very similarly to **dynamic_cast** operator in C++.
First it evaluates expression *expr*. If evaluation results in an object of class specified by *classname* or derived class, it returns the object, otherwise `null`.

Parameters

classname name of class to cast *expr* to

expr expression of any type

Returns

object of class *desttype* (or derived) on success, otherwise `null`

Note

- Other than **dynamic_cast** operator in C++ there is no exeption thrown if *expr* is not an object. In this case function simply returns `null`. Use **gettype** to check a value for its basic type.
- Function preserves constant state of *expr*, i.e., if result of *expr* is a constant object, result of function is `const` too. In that case object is cloned if result is assigned to a non-const variable.

Example:

```
class Base
{
   Base();
}
Base::Base() { }

class Derived : Base
{
   Derived();
}
Derived::Derived() { }

main(;obj1, obj1, o, i)
{
   obj1 = new Base();
   obj2 = new Derived();
   o = dynamic_cast("Base", obj1);               // returns obj1
   o = dynamic_cast("Base", obj2);               // returns obj2
   o = dynamic_cast("Derived", obj1);            // returns null
   o = dynamic_cast("Derived", obj2);            // returns obj2
   o = dynamic_cast(getclassname(obj2), obj1);   // returns null
   o = dynamic_cast(getclassname(obj1), obj2);   // returns obj2
   i = dynamic_cast("Base", 17);                 // returns null (not 0)
}
```

See also

gettype, **getclassname**, **dynamic_cast(Class, var)**

References **var**.

◆ findmethod()

```
function findmethod ( const var    objectOrClass,
                      const string name,
                      bool         isstatic = false )
```

Searches for a method of object or class, available in current call context.

Parameters

objectOrClass	Class or object to search for method in. Argument also can be a string specifying a class name.
name	Name of method to search for. This argument is strong typed, i.e. an implicit cast is done if needed.
isstatic	Flag determines, if search is done for static method (class function) or element function.

Returns

Function if found, otherwise `null`.

Note

- Since B++ does not support multiple inheritance nor an interface concept, this function can be used to check whether a particular object implements an expected function.
- The behavior of the function depends on the calling context. If the function is called within a member function of an object and the object's class references a base class, then protected methods will also be found. If *`objectOrClass`* references the same class as the calling object, private methods will be found as well. Otherwise, *`name`* must reference a public method.

See also

BppClass::findMember

References **var**.

◆ float()

```
float float ( const var value )
```

Converts value to **float**.

Function tries to convert argument into value of **float** type.

Parameters

value Value to convert. The value must be of type **int**, **float**, **double**, string, buffer or object.

Returns

value converted to **float**

Note

- A string value must be string representation of a **float** number in format [*whitespace*] [*sign*] [digits] [.digits] [{d | D | e | E }[sign]digits] where a whitespace is a space or tab character and sign is either plus (+) or minus (-).
- If *value* is of buffer type, the buffer must contain a 32 bits (float) or 64 bits (double) value (i.e. size of buffer must be 4 or 8).
- In case of object for *value*, the specified object must define a **float** or **double** conversion operator.
- In case of invalid argument type exception of string type is thrown. Use **gettype** to check type of value.

See also

size, **gettype**

References **var**.

◆ getclass()

```
Class getclass ( const var value )
```

Gets class of object or finds class for specified name.

Parameters

value `object`, `Class` or `string` to get class for.

Returns

class for *value*

Note

- In case of invalid argument type, exception of string type is thrown.
- If *value* is of `string` type and no class found for (qualified) name specified by *value*, result is `null`.
- If *value* is already a class, function does nothing but return the argument.

See also

dynamic_cast, **gettype**, **getclassname**, **newobject**

References **var**.

◆ getclassname()

```
string getclassname ( const var value,
                      bool      qualified = false )
```

Get class name of object or class.

Parameters

value `object` or `Class` to get class name from.

qualified If flag is set, fully qualified name of class will be returned.

Returns

class name

Note

In case of invalid argument type, exception of string type is thrown.

See also

dynamic_cast, **gettype**, **getclass**

References **var**.

◆ getsymbol()

```
var getsymbol ( const string name,
                bool         isclass = false )
```

Gets a global symbol (variable, function or class) by name at runtime.

Parameters

name Name of symbol to search for.

isclass If set to `true` function searches for a class, otherwise for global symbol of any other type.

Returns

If a symbol for *name* was found, function returns its value, e.g. function pointer for a function, otherwise `null`.

Note

This function is mainly intended for using different system functions, depending on platform used, in same program without need of platform specific compilation.

Example

```
// Creates a new directory
bool mkDir(const string path)
{
   intptr adr = addr(path);
   if (__OSTYPE__ == "DOS")
   {
      function segread = getsymbol("segread");
      function getreg = getsymbol("getreg");
      function setreg = getsymbol("setreg");
      function int86x = getsymbol("int86x");
      segread();
      setreg("ah",0x39);
      setreg("ds",adr >> 16);
      setreg("dx",adr &6 0xffff);
      int86x(0x21);
      return !getreg("cflag");
   }
   else if (__OSTYPE__ == "WIN32")
   {
      function GetFunc = getsymbol("GetKernel32Func");
      intptr pfunc = GetFunc("CreateDirectory?");
      return CallLibFunc(pfunc,adr,0);
   }
   // Fallback for other systems
   return system("mkdir "+path) ==0;
}
```

See also

dynamic_cast, **gettype**

References **var**.

◆ gettype()

```
int gettype ( const var value )
```

Get type id of argument value.

Parameters

value value to get type id from

Returns

type id of *value*

> **Note**
>
> Use **gettypename** function to convert returned id into readable string representation.

See also

gettypename, **BppValueType**

References **var**.

Referenced by **Rational::Rational()**, **JSON::_isDict()**, **JSON::_isNumeric()**, **JSON::_isString()**, **JSON::_isVector()**, **JSON::_writeJson()**, **JSON::_writeval()**, and **StatusBar::create()**.

◆ gettypename()

```
string gettypename ( int  typeId,
                     bool prefixed = false )
```

Get human readable name of a data type.

Parameters

typeId id of data type to get name of

prefixed Specifies if result is prefixed by 'BVT_' in order to get internal symbol name of type. Prefix is used if *prefixed* can be evaluated to `true`.

Returns

Name of data type. Following table shows possible names:

Type id (HEX)	Type name	Type
0 (00)	NULL	null
1 (01)	VOID	void
2 (02)	AUTO	auto
64 (40)	SBYTE	sbyte
65 (41)	BOOL	bool
66 (42)	CHAR	char
68 (44)	BYTE	bool
70 (46)	UCHAR	uchar
71 (47)	U8CHAR	u8char
72 (48)	SHORT	short
76 (4C)	USHORT	ushort
78 (4E)	CHAR16	char16_t
79 (4F)	WCHAR	wchar_t
80 (50)	INT	int
81 (51)	INTPTR	intptr
84 (54)	UINT	uint
85 (55)	UINTPTR	uintptr
86 (56)	CHAR32	char32_t
88 (58)	LONG	long
92 (5C)	ULONG	ulong
112 (70)	FLOAT	float
120 (78)	DOUBLE	double
128 (80)	FILE	FILE
129 (81)	CODE	native function
130 (82)	VAR	variable, dictionary entry

160 (A0)	OBJECT	`object`
161 (A1)	STRONGOBJECT	`strong typed object`
168 (A8)	CLASS	`Class`
176 (B0)	STRING	`string`
177 (B1)	BUFFER	`buffer`
178 (B2)	CHARBUFFER	`charbuffer`
180 (B4)	VECTOR	**`vector`**
181 (B5)	BYTECODE	B++ function
182 (B6)	LOCALVERINFO	local variable info (internal used)
184 (B8)	DICTIONARY	`dictionary`
224 (E0)	OBJECTREF	free reference to `object`
232 (E8)	CLASSREF	free reference to `Class`
240 (F0)	STRINGREF	free reference to `string`
241 (F1)	BUFFERREF	free reference to `buffer`
242 (F2)	CHARBUFFERREF	free reference to `charbuffer`
244 (F4)	VECTORREF	free reference to **`vector`**
245 (F5)	VARINFOREF	free reference to B++ function
246 (F6)	CODEREF	free reference to native function
248 (F8)	DICTIONARYREF	free reference to `dictionary`

Note

Predefined literals of symbolc type names should be used for *typeid* value rather than numeric constants, e.g. BVT_INT instead of 80.

See also

gettype

Referenced by **StatusBar::create()**.

◆ `int()`

```
int int ( const var value )
```

Converts value to integer.

Function tries to convert argument into value of **int** type (32 bits signed integer).

Parameters

value Value to convert. The value must be of any number type, `string`, `buffer` or `object`.

Returns

value converted to **int**

Note

- A `string` value should contain string representation of a decimal, octal, hexadecimal or dual integer number. Hexadecimal numbers are prefixed by '0x'or '0X', dual numbers by '0b' or '0B'. An octal number starts with 0, followed by a digit between 1 and 7. In all other cases a decimal number is assumed.
- If *value* is of `buffer` type, the buffer must contain a 8, 16, 32 or 64 bits integer value (i.e. size of buffer must be 1, 2, 4 or 8).
- In case of `object` for *value*, the specified object must define an **int** conversion operator.
- In case of invalid argument type exception of string type is thrown. Use **gettype** to check type of value.

See also

size, **gettype**

References **var**.

Referenced by **BppString::BppString()**, **BppInterpreter::_OP_AREF()**, **BppInterpreter::_OP_MKCONST()**, **BppInterpreter::_OP_TREF()**, **BppInterpreter::_OP_TSET()**, **Bp2WinBase::applyScrollBars()**, **BppInterpreter::call()**, **idCmpFunc()**, **BppInterpreter::invokeFunc()**, **BppInterpreter::invokeMember()**, **ValPtr::operator++()**, **ValPtr::operator+=()**, **BppWinMsgInput::setChar()**, **BppWinMsgInput::setChar()**, and **Bp2WinBase::show()**.

◆ `intptr()`

```
intptr intptr ( const var value )
```

Converts value to integer of size same as address pointers of system.

Function tries to convert argument into value of **intptr** type (32 or 64 bits signed integer).

Parameters

value Value to convert. The value must be of any number type, `string`, `buffer` or `object`.

Returns

value converted to **int**

Note

- A `string` value should contain string representation of a decimal, octal, hexadecimal or dual integer number. Hexadecimal numbers are prefixed by '0x'or '0X', dual numbers by '0b' or '0B'. An octal number starts with 0, followed by a digit between 1 and 7. In all other cases a decimal number is assumed.
- If *value* is of `buffer` type, the buffer must contain a 8, 16, 32 or 64 bits integer value (i.e. size of buffer must be 1, 2, 4 or 8).
- In case of `object` for *value*, the specified object must define an **intptr** conversion operator.
- In case of invalid argument type exception of string type is thrown. Use **gettype** to check type of value.

See also

size, **gettype**

References **intptr()**, and **var**.

Referenced by **GDI::Bitmap::Bitmap()**, **GDI::Brush::Brush()**, **GDI::ClientDC::ClientDC()**, **GDI::ColorSpace::ColorSpace()**, **GDI::DC::DC()**, **GDI::DriverDC::DriverDC()**, **GDI::EnhMetaFile::EnhMetaFile()**, **GDI::Font::Font()**, **GDI::GDIObject::GDIObject()**, **GDI::InfoDC::InfoDC()**, **GDI::MetaDC::MetaDC()**, **GDI::PaintDC::PaintDC()**, **GDI::Palette::Palette()**, **GDI::Pen::Pen()**, **GDI::Region::Region()**, **GDI::WindowDC::WindowDC()**, **addr()**, **GDI::Palette::animate()**, **Menu::AppendMenu()**, **WinBase::beginPaint()**, **String::c_str()**, **CallLibFunc()**, **cleartimer()**, **GDI::Bitmap::create()**, **WinBase::create()**, **Window::create()**, **GDI::Bitmap::CreateBitmap()**, **GDI::Pen::createExt()**, **WinBase::createFromHandle()**, **GDI::Bitmap::createFromHDC()**, **GDI::Brush::createIndirect()**, **GDI::DC::describePixelFormat()**, **GDI::DC::enumBrushes()**, **GDI::DC::enumFontFamiliesEx()**, **GDI::DC::enumFonts()**, **GDI::DC::enumPens()**, **GDI::EnhMetaFile::enumRecords()**, **WinBase::findWindow()**, **FreeLibrary()**, **GDI::Bitmap::getBITMAP()**, **GetCallbackFunc()**, **GetCoreFunc()**, **GDI::DC::getCurrentObject()**, **GetDialogBaseUnits()**, **GetGdi32Func()**, **GDI::GDIObject::getHandle()**, **Menu::getHandle()**, **WinBase::getHandle()**, **GDI::DC::getHDC()**, **GDI::EnhMetaFile::getHeader()**, **WinBase::getHInstance()**, **GDI::DC::getHWND()**, **MSG::getHwnd()**, **EditCtrl::getInsertionPosition()**, **GetKernel32Func()**, **GDI::Brush::getLOGBRUSH()**, **GDI::Font::getLOGFONT()**, **GDI::Pen::getLOGPEN()**, **MSG::getLParam**

(), **GDI::GDIObject::getObject()**, **GetOle32Func()**, **GDI::DC::getOutlineTextMetrics()**, **WinBase::getParentHandle()**, **GetProcAddress()**, **GetShell32Func()**, **GetStaticCallbackFunc()**, **GDI::GDIObject::getStockObject()**, **GetSysColor()**, **GetUser32Func()**, **GetWindowLong()**, **GetWindowLongPtr()**, **GDI::DC::gradientFill()**, **StatusBar::handleMessage()**, **WinBase::handleMessage()**, **Window::handleMessage()**, **WinMsgHandler::handleMessage()**, **Menu::InsertMenu()**, **intptr()**, **LoadImage()**, **LoadLibrary()**, **LoadSysBitmap()**, **LoadSysCursor()**, **LoadSysIcon()**, **memmove()**, **MessageBox()**, **GDI::BrushEnumeratorBase::operator()()**, **GDI::EnhMetaFile::EnumProc::operator()()**, **GDI::FontEnumeratorBase::operator()()**, **GDI::PenEnumeratorBase::operator()()**, **GDI::DC::pathToRegion()**, **GDI::DC::playEnhMetaFile()**, **PostMessage()**, **MSG::ptr()**, **GDI::DriverDC::reset()**, **GDI::DC::selectObject()**, **GDI::DC::selectPalette()**, **SendMessage()**, **WinBase::setBackgroundBrush()**, **GDI::Palette::setEntries()**, **MSG::setHwnd()**, **WinBase::setIdleMsg()**, **MSG::setLParam()**, **SetParent()**, **settimer()**, **SetWindowLong()**, **SetWindowLongPtr()**, and **WinBase::unregisterWindow()**.

◆ isconst()

```
bool isconst ( var expr )
```

Checks if specified value is const.

Parameters

expr any expression, typical variable or result of function call, to check for constant

Returns

true if *expr* is constant, false otherwise

See also

mkconst

References **var**.

◆ long()

```
long long ( const var value )
```

Converts value to long integer.

Function tries to convert argument into value of **long** type (64 bits signed integer).

Parameters

value Value to convert. The value must be of any number type, `string`, `buffer` or `object`.

Returns

value converted to **int**

Note

- A `string` value should contain string representation of a decimal, octal, hexadecimal or dual integer number. Hexadecimal numbers are prefixed by '0x'or '0X', dual numbers by '0b' or '0B'. An octal number starts with 0, followed by a digit between 1 and 7. In all other cases a decimal number is assumed.
- If *value* is of `buffer` type, the buffer must contain a 8, 16, 32 or 64 bits integer value (i.e. size of buffer must be 1, 2, 4 or 8).
- In case of `object` for *value*, the specified object must define a **long** conversion operator.
- In case of invalid argument type exception of string type is thrown. Use **gettype** to check type of value.

See also

size, **gettype**

References **var**.

◆ `mkconst()`

```
var mkconst ( var  expr,
              bool setconst = true )
```

Modifies constant state of exprerssion result.

Parameters

expr any expression, typical variable or result of function call, change constant state for

setconst new constant state

Returns

value of *expr* with modified constant state

Note

Function never modifies state of an l-value (variable) specified by *expr*. Result is either a copy of value in *expr* (in case of simple value) or a reference (in case of reference type value) with constant state as specified by *setconst*.

See also

isconst

References **var**.

◆ newobject()

```
object newobject ( Class classtype,
                         ... )
```

Creates new object instance of specified class.

Function behaves like new operator but allows dynamic specification of class to create instance of. It is intended for use as a helper for generic object creation at runtime.

Parameters

classtype Class to use for object creation.

... List of arguments for constructor call.

Returns

object created

Note

In case of invalid argument type or value `null` for *classtype*, exception of string type is thrown.

Example

```
object factoryFunc(string classname, ...) {
   vector args = argv();
   // replace class name by class reference in arguments
   args[0] = getclass(classname);
   return call(newobject,args);
}
```

See also

dynamic_cast, **gettype**, **getclassname**, **getclass**

◆ sbyte()

```
sbyte sbyte ( const var value )
```

Converts value to signed byte.

Function tries to convert argument into value of `sbyte` type (8 bits signed integer).

Parameters

value Value to convert. The value must be of any number or character type, `string`, `buffer` or `object`.

Returns

value converted to `sbyte`

Note

- A `string` value should contain string representation of a decimal, octal, hexadecimal or dual integer number. Hexadecimal numbers are prefixed by '0x'or '0X', dual numbers by '0b' or '0B'. An octal number starts with 0, followed by a digit between 1 and 7. In all other cases a decimal number is assumed.
- If *value* is of `buffer` type, the buffer must contain a 8, 16, 32 or 64 bits integer value (i.e. size of buffer must be 1, 2, 4 or 8).
- In case of `object` for *value*, the specified object must define a `sbyte` conversion operator.
- In case of invalid argument type exception of string type is thrown. Use **gettype** to check type of value.

See also

size, **gettype**

References **sbyte()**, and **var**.

Referenced by **sbyte()**.

◆ short()

```
short short ( const var value )
```

Converts value to short integer.

Function tries to convert argument into value of **short** type (16 bits signed integer).

Parameters

value Value to convert. The value must be of any number type, `string`, `buffer` or `object`.

Returns

value converted to **int**

Note

- A `string` value should contain string representation of a decimal, octal, hexadecimal or dual integer number. Hexadecimal numbers are prefixed by '0x'or '0X', dual numbers by '0b' or '0B'. An octal number starts with 0, followed by a digit between 1 and 7. In all other cases a decimal number is assumed.
- If *value* is of `buffer` type, the buffer must contain a 8, 16, 32 or 64 bits integer value (i.e. size of buffer must be 1, 2, 4 or 8).
- In case of `object` for *value*, the specified object must define a **short** conversion operator.
- In case of invalid argument type exception of string type is thrown. Use **gettype** to check type of value.

See also

size, **gettype**

References **var**.

Referenced by **BppValue::assign()**, and **Bp2Struct_addInt16()**.

◆ `str2utf8buf()`

```
buffer str2utf8buf ( const string src,
                     bool         raw = false )
```

Converts string to UTF-8 buffer.

Function converts argument to a buffer, containing UTF-8 representation of *src*.

Parameters

src string to convert.

raw If set to *true*, contents of *src* is copied to result without conversion from internal character set. Flag is ignored on Unicode builds.

Returns

buffer containing UTF-8 representation of *src*. The result contains a trailing Null byte.

Note

The *src* argument may be of any type, if it is implicitely convertible into string.

References **buffer()**.

◆ string()

```
string string ( const var value )
```

Converts value to `string`.

Function tries to convert argument into value of `string` type.

Parameters

value Value to convert. The value must be of any numeric type or character type, `charbuffer` or `object`.

Returns

value converted to `string`

Note

- Numeric values of integral types (except **`intptr`** and `uintptr`) are converted to decimal string representation.
- Values of type **`intptr`** or `uintptr` are represented as haxadecimal addresses (like `printf("%p",...)` in C).
- Values of types **`float`** and **`double`** will be converted to string without any special format (and without expotantial notation).
- Values of character types are converted into internal character type and result in a string containing single character.
- In case of a `charbuffer` value a new string containing all characters of buffer is returned.
- Values of `object` type must have a `string` conversion operator or a `toString` method for conversion.
- In case of invalid argument type exception of string type is thrown. Use **gettype** to check type of value.

See also

size, **gettype**

References **var**.

Referenced by **JSON::_writeval()**, **StatusBar::create()**, **BppXML::decodeXmlString()**, **BppXML::encodeXmlString()**, **GDI::EnhMetaFile::getDescription()**, and **Rational::operator string()**.

◆ u8char()

```
u8char u8char ( const var value )
```

Converts value to UTF-8 unicode character.

Function tries to convert argument into value of **u8char** type.

Parameters

value Value to convert. The value must be of any number or character type, `string`, `buffer` or `object`.

Returns

value converted to **int**

Note

- The internal representation of **u8char** is the representation of UTF-8 byte sequence and is stored as 32 bits value.
- A `string` value should contain string representation of a decimal, octal, hexadecimal or dual integer number. Hexadecimal numbers are prefixed by '0x'or '0X', dual numbers by '0b' or '0B'. An octal number starts with 0, followed by a digit between 1 and 7. In all other cases a decimal number is assumed.
- If *value* is of `buffer` type, and size of buffer is lower or equal to 4, its content is assumed to be a byte sequence representing the bytes of UTF-8 character.
- In case of `object` for *value*, the specified object must define a **u8char** conversion operator.
- In case of invalid argument type exception of string type is thrown. Use **gettype** to check type of value.

See also

size, **gettype**

References **u8char()**, and **var**.

Referenced by **u8char()**.

◆ uchar()

```
uchar uchar ( const var value )
```

Converts value to unsigned char.

Function tries to convert argument into value of `uchar` type (unsigned single byte character).

Parameters

value Value to convert. The value must be of any number or character type, `string`, `buffer` or `object`.

Returns

value converted to **char**

Note

- A `string` value should contain string representation of a decimal, octal, hexadecimal or dual integer number. Hexadecimal numbers are prefixed by '0x'or '0X', dual numbers by '0b' or '0B'. An octal number starts with 0, followed by a digit between 1 and 7. In all other cases a decimal number is assumed.
- If *value* is of `buffer` type, the buffer must contain a 8, 16, 32 or 64 bits integer value (i.e. size of buffer must be 1, 2, 4 or 8).
- In case of `object` for *value*, the specified object must define a `uchar` conversion operator.
- In case of invalid argument type exception of string type is thrown. Use **gettype** to check type of value.

See also

size, **gettype**

References **uchar()**, and **var**.

Referenced by **uchar()**.

◆ `uint()`

```
uint uint ( const var value )
```

Converts value to unsigned integer.

Function tries to convert argument into value of `uint` type (32 bits unsigned integer).

Parameters

value Value to convert. The value must be of any number type, `string`, `buffer` or `object`.

Returns

value converted to `int`

Note

- A `string` value should contain string representation of a decimal, octal, hexadecimal or dual integer number. Hexadecimal numbers are prefixed by '0x'or '0X', dual numbers by '0b' or '0B'. An octal number starts with 0, followed by a digit between 1 and 7. In all other cases a decimal number is assumed.
- If *value* is of `buffer` type, the buffer must contain a 8, 16, 32 or 64 bits integer value (i.e. size of buffer must be 1, 2, 4 or 8).
- In case of `object` for *value*, the specified object must define an `uint` conversion operator.
- In case of invalid argument type exception of string type is thrown. Use **gettype** to check type of value.

See also

size, **gettype**

References **uint()**, and **var**.

Referenced by **GDI::HatchedBrush::HatchedBrush()**, **GDI::MemoryDC::MemoryDC()**, **GDI::MetaDC::MetaDC()**, **RGNDATA::RGNDATA()**, **GDI::SolidBrush::SolidBrush()**, **StatusBar::StatusBar()**, **XMLElement::_appendElements()**, **XMLElement::_getMatchMode()**, **XMLCDATASection::_serialize()**, **XMLComment::_serialize()**, **XMLDocument::_serialize()**, **XMLDocumentFragment::_serialize()**, **XMLDocumentType::_serialize()**, **XMLElement::_serialize()**, **XMLNode::_serialize()**, **XMLProcessingInstruction::_serialize()**, **XMLText::_serialize()**, **JSON::_writeJson()**, **GDI::DC::alphaBlend()**, **GDI::Palette::animate()**, **Menu::AppendMenu()**, **StatusBar::applyPaneSizes()**, **GDI::DC::bitBlt()**, **bsearch()**, **bsearch()**, **Vector::capacity()**, **GDI::Bitmap::create()**, **GDI::Font::create()**, **GDI::Pen::create()**, **StatusBar::create()**, **GDI::Bitmap::CreateBitmap()**, **GDI::Pen::createExt()**, **GDI::Bitmap::createFromFile()**, **GDI::Bitmap::createFromHDC()**, **GDI::Bitmap::createMono()**, **GDI::Region::createRegionExt()**, **GDI::Bitmap::createRGB()**, **GDI::Bitmap::createRGBA()**, **GDI::DC::drawAngleArc()**, **GDI::DC::drawPathData()**, **__SubArrayProxy::elSize()**, **erase()**, **String::erase()**, **Vector::erase()**, **BppXML::fillAttributes()**, **find()**, **String::find()**, **GDI::DC::floodFill()**, **foreach()**, **XMLNode::foreach()**, **fread()**, **fread()**, **fread()**, **fwrite()**, **fwrite()**, **fwrite()**, **GDI::GetBatchLimit()**, **GDI::Bitmap::getBitmapBits()**, **GDI::DC::getBkColor()**, **GDI::DC::getBoundsRect()**,

GDI::DC::getBrushColor(), GetCallbackFunc(), GDI::DC::getCharABCWidths(), GDI::DC::getCharABCWidthsFloat(), GDI::DC::getCharacterPlacement(), GDI::DC::getCharWidth(), GDI::DC::getCharWidthFloat(), XMLElement::getChildElementByAttribute(), GDI::DC::getCurrentObject(), GDI::Region::getData(), CStruct::getDataSize(), GDI::EnhMetaFile::getDescription(), GetDialogBaseUnits(), XMLDocument::getDoctype(), XMLDocument::getDocumentElement(), GDI::EnhMetaFile::getEnhMetaFileBits(), GDI::Palette::getEntries(), XMLElement::getFirstChildElement(), GDI::DC::getFontLanguageInfo(), GDI::DC::getKerningPairs(), XMLNode::getLastChild(), GetLastError(), RegEx::getMatches(), RegEx::getMatchPositions(), MSG::getMsgId(), GDI::DC::getNearestColor(), GDI::Palette::getNearestIndex(), GDI::EnhMetaFile::getPaletteEntries(), StatusBar::getPaneText(), GDI::DC::getPenColor(), GDI::DC::getPixel(), GDI::EnhMetaFile::getPixelFormat(), WinBase::getScrollBarMode(), XMLNode::getSibling(), GetStaticCallbackFunc(), CStruct::getStructPacking(), RegEx::getSubExpressions(), RegEx::getSubExpressions(), GetSysColor(), GetSysErrorMsg(), GDI::DC::getSystemPaletteEntries(), GDI::DC::getSystemPaletteUse(), GDI::DC::getTextAlign(), GDI::DC::getTextColor(), XMLNode::getTextContent(), GDI::GDIObject::getType(), GDI::DC::gradientFill(), GDI::FontEnumeratorBase::handleFont(), StatusBar::handleMessage(), WinBase::handleMessage(), Window::handleMessage(), WinMsgHandler::handleMessage(), HIWORD(), WinBase::initCommonControls(), insert(), String::insert(), String::insert(), Vector::insert(), XMLNode::insertAt(), Menu::insertItemAt(), Menu::InsertMenu(), Menu::insertMenuAt(), XMLNode::insertNode(), insrange(), XMLNode::isEqualNode(), GDI::DC::isRectVisible(), lbound(), lbound(), String::length(), LoadImage(), LOWORD(), memmove(), memsize(), move(), newbuffer(), newcbuffer(), newstring(), newvector(), __SubArrayProxy::offsetOf(), __SubstructProxy::offsetOf(), CStruct::offsetOf(), String::OP_VREF(), Vector::OP_VREF(), String::OP_VSET(), GDI::FontEnumeratorBase::operator()(), RegEx::operator()(), __SubArrayProxy::operator[](), __SubArrayProxy::operator[]=(), BppXML::parse(), GDI::DC::patBlt(), GDI::DC::realizePalette(), XMLNode::removeChildAt(), XMLNode::replaceChildAt(), reserve(), GDI::Palette::resize(), resize(), reverse(), rfind(), RegEx::search(), XMLNode::serialize(), GDI::DriverDC::setAbortHandler(), WinBase::setBackgroundColor(), GDI::SetBatchLimit(), GDI::Bitmap::setBitmapBits(), GDI::DC::setBkColor(), GDI::DC::setBkColorRGB(), GDI::DC::setBkMode(), GDI::DC::setBkOpaque(), GDI::DC::setBkTransparent(), GDI::DC::setBoundsRect(), GDI::DC::setBrushColor(), Vector::setCapacity(), GDI::Bitmap::setDimensions(), GDI::Palette::setEntries(), WinBase::setForegroundColor(), WinBase::setIdleMsg(), MSG::setMsgId(), StatusBar::setPaneText(), GDI::DC::setPenColor(), GDI::DC::setPixel(), WinBase::setScrollBarMode(), SetStaticCallbackHandler(), GDI::DC::setSystemPaletteUse(), GDI::DC::setTextAlign(), GDI::DC::setTextColor(), settimer(), shrink(), Vector::shrink(), __SubArrayProxy::size(), Dictionary::size(), size(), Vector::size(), __SubstructProxy::sizeOf(), CStruct::sizeOf(), sleep(), slice(), Vector::slice(), spawn(), strerase(), GDI::DC::stretchBlt(), GDI::DC::stretchDIBits(), strfind(), JSON::stringify(), strinsert(), strinsert(), strlen(), String::substr(), substr(), swap(), GDI::DC::textOutExt(), time(), GDI::DC::transparentBlt(), ubound(), ubound(), uint(), vecsize(), verase(), vinsert(), vmove(), vslice(), and vswap().

◆ uintptr()

```
uintptr uintptr ( const var value )
```

Converts value to unsigned integer of size same as address pointers of system.

Function tries to convert argument into value of `uintptr` type (32 or 64 bits unsigned integer).

Parameters

value Value to convert. The value must be of any number type, `string`, `buffer` or `object`.

Returns

value converted to `int`

Note

- A `string` value should contain string representation of a decimal, octal, hexadecimal or dual integer number. Hexadecimal numbers are prefixed by '0x'or '0X', dual numbers by '0b' or '0B'. An octal number starts with 0, followed by a digit between 1 and 7. In all other cases a decimal number is assumed.
- If *value* is of `buffer` type, the buffer must contain a 8, 16, 32 or 64 bits integer value (i.e. size of buffer must be 1, 2, 4 or 8).
- In case of `object` for *value*, the specified object must define an `intptr` conversion operator.
- In case of invalid argument type exception of string type is thrown. Use **gettype** to check type of value.

See also

size, **gettype**

References **uintptr()**, and **var**.

Referenced by **MSG::getWParam()**, **WinBase::handleMessage()**, **WinBase::setIdleMsg()**, **MSG::setWParam()**, and **uintptr()**.

◆ `ulong()`

```
ulong ulong ( const var value )
```

Converts value to unsigned long integer.

Function tries to convert argument into value of `ulong` type (64 bits unsigned integer).

Parameters

value Value to convert. The value must be of any number type, `string`, `buffer` or `object`.

Returns

value converted to `int`

Note

- A `string` value should contain string representation of a decimal, octal, hexadecimal or dual integer number. Hexadecimal numbers are prefixed by '0x'or '0X', dual numbers by '0b' or '0B'. An octal number starts with 0, followed by a digit between 1 and 7. In all other cases a decimal number is assumed.
- If *value* is of `buffer` type, the buffer must contain a 8, 16, 32 or 64 bits integer value (i.e. size of buffer must be 1, 2, 4 or 8).
- In case of `object` for *value*, the specified object must define an `ulong` conversion operator.
- In case of invalid argument type exception of string type is thrown. Use **gettype** to check type of value.

See also

size, **gettype**

References **ulong()**, and **var**.

Referenced by **Rational::Rational()**, **Rational::approximateConvergents()**, **Rational::approximateIfNeeded()**, **Rational::forceApproximate()**, **Rational::fromDouble()**, **Rational::gcd()**, **Rational::getMaxDenomThreshold()**, **Rational::lcm()**, **Rational::setMaxDenomThreshold()**, **Rational::simplify()**, and **ulong()**.

◆ ushort()

```
ushort ushort ( const var value )
```

Converts value to unsigned short integer.

Function tries to convert argument into value of **ushort** type (16 bits unsigned integer).

Parameters

value Value to convert. The value must be of any number type, `string`, `buffer` or `object`.

Returns

value converted to **int**

Note

- A `string` value should contain string representation of a decimal, octal, hexadecimal or dual integer number. Hexadecimal numbers are prefixed by '0x'or '0X', dual numbers by '0b' or '0B'. An octal number starts with 0, followed by a digit between 1 and 7. In all other cases a decimal number is assumed.
- If *value* is of `buffer` type, the buffer must contain a 8, 16, 32 or 64 bits integer value (i.e. size of buffer must be 1, 2, 4 or 8).
- In case of `object` for *value*, the specified object must define an **ushort** conversion operator.
- In case of invalid argument type exception of string type is thrown. Use **gettype** to check type of value.

See also

size, **gettype**

References **ushort()**, and **var**.

Referenced by **TTPOLYCURVE::TTPOLYCURVE()**, **XMLDOMException::XMLDOMException()**, **XMLNode::XMLNode()**, **XMLNode::getNodeType()**, **HIBYTE()**, **HIWORD()**, **LoadSysBitmap()**, **LoadSysCursor()**, **LoadSysIcon()**, **LOBYTE()**, **LOWORD()**, **MAKEWORD()**, **BppXML::parse()**, and **ushort()**.

◆ `utf8buf2str()`

```
string utf8buf2str ( const buffer src,
                     bool         raw = false )
```

Converts UTF-8 string to string.

Function converts UTF-8 string contained in *src* into internal string representation.

Parameters

src buffer containing UTF-8 string

raw If set to *true*, contents of *src* is copied to result without conversion to internal character set. Flag is ignored on Unicode builds.

Returns

converted string

Note

The function does not check whether the content of *src* contains a valid UTF-8 string. If it does not, the result is undefined.

References **buffer()**.

◆ vector()

```
vector vector ( const var value )
```

Converts value to **vector**.

Function tries to convert argument into value of **vector** type.

Parameters

value Value to convert. The value must be of type **vector**, buffer, charbuffer, dictionary or object.

Returns

value converted to **vector**

Note

- In case of a **vector** function simply returns a reference (not a copy!) of argument. If argument is const, result is const too.
- In case of a buffer argument each byte of argument is converted into vector element of type byte.
- In case of a charbuffer argument each character of argument is converted into vector element of internal character type.
- In case of a dictionary argument result vector consists of values from dictionary.
- Values of object type must have a **vector** conversion operator.
- In case of invalid argument type exception of string type is thrown. Use **gettype** to check type of value.

See also

size, **gettype**

References **var**, and **vector()**.

Referenced by **XMLElement::_appendElements()**, **JSON::_writeJson()**, **StatusBar::applyPaneSizes()**, **argv()**, **call()**, **callafter()**, **callfail()**, **callsucc()**, **XMLNode::cloneNode()**, **dictgetkeys()**, **dictgetvalues()**, **BppXML::fillAttributes()**, **XMLNode::foreach()**, **WinBase::getAcceleratorArray()**, **getargs()**, **XMLElement::getChildElements()**, **XMLNode::getChildNodes()**, **WinBase::getChildrenHandles()**, **__SubArrayProxy::getData()**, **XMLDocument::getElementsByTagName()**, **XMLElement::getElementsByTagName()**, **FileDialog::getFilter()**, **EditCtrl::getInsertionPosition()**, **RegEx::getMatches()**, **RegEx::getMatchPositions()**, **XMLElement::getNextSiblingElement()**, **XMLElement::getPreviousSiblingElement()**, **XMLNode::getSibling()**, **RegEx::getSubExpressions()**, **RegEx::getSubExpressions()**, **getusrargs()**, **gmtime()**, **XMLNode::insertNode()**, **localtime()**, **mktime()**, **newvector()**, **XMLDOMException::operator string()**, **BppXML::parse()**, **WinBase::setIdleMsg()**, **settimer()**, **JSON::stringify()**, **strsplit()**, **T()**, **vclear()**, **Vec()**, **Vector::vec()**, **vecsize()**, **vector()**, **verase()**, **vinsert()**, **vmove()**, **vslice()**, and **vswap()**.

◆ wchar_t()

```
wchar_t wchar_t ( const var value )
```

Converts value to wide character.

Function tries to convert argument into value of **wchar_t** type.

Parameters

value Value to convert. The value must be of any number or character type, `string`, `buffer` or `object`.

Returns

value converted to **int**

Note

- The size of **wchar_t** depends on runtime environment used.
- A `string` value should contain string representation of a decimal, octal, hexadecimal or dual integer number. Hexadecimal numbers are prefixed by '0x'or '0X', dual numbers by '0b' or '0B'. An octal number starts with 0, followed by a digit between 1 and 7. In all other cases a decimal number is assumed.
- If *value* is of `buffer` type, the buffer must contain a 8, 16, 32 or 64 bits integer value (i.e. size of buffer must be 1, 2, 4 or 8).
- In case of `object` for *value*, the specified object must define a **wchar_t** conversion operator.
- In case of invalid argument type exception of string type is thrown. Use **gettype** to check type of value.

See also

size, **gettype**

References **var**.

Referenced by **Bp2String_w_buf()**, **str2wcsBuf()**, **BppString::w_str()**, and **BppDictionaryIO::writeValue()**.

B++ Reference © 2005-2026 by R.-Erik Ebert

All parts of B++ are provided under terms and conditions of MIT license.

Date and time functions

B++ predefined functions

Collaboration diagram for Date and time functions:



Functions

`long time ()`
Get C-timestamp.
`uint timer ()`
Gets elapsed time.
`string ctime (long t)`
Gets string representation of a C-timestamp.
`vector localtime (long t)`
Makes a localized time point from C-timestamp and split it into different parts.
`vector gmtime (long t)`
Makes a GMT time point from C-timestamp and split it into different parts.
`long mktime (vector tm)`
Makes C-timestamp from localized time point.
`string strftime (const string fmt, var tm)`
Function converts date and time information from a given calendar time tm to character string according to format string fmt.

Detailed Description

Function Documentation

◆ ctime()

```
string ctime ( long t )
```

Gets string representation of a C-timestamp.

Function makes a readable string from timestamp. It behaves like same named C function.

Parameters

t C-timestamp, e.g. result of a **time()** call. On 16-Bit-platforms argument type is *int* (32 bits).

Returns

string representation of argument, e.g. *Sat May 10 21:52:54 2008*

◆ gmtime()

```
vector gmtime ( long t )
```

Makes a GMT time point from C-timestamp and split it into different parts.

Behavior of function is similarly to same named C function. However, because B++ doesn't support structure types it uses a vector for result.

Parameters

t C-Timestamp, on 16-Bit-platforms argument type is *int* (32 bits).

Returns

vector of following components:

Index>	Value
0	Seconds after minute (0 .. 59).
1	Minutes after hour (0 .. 59).
2	Hours after midnight (0 .. 23).
3	Day of month (1 .. 31).
4	Month (0 .. 11; January = 0).
5	Year (current year minus 1900).
6	Day of week (0 .. 6; Sunday = 0).
7	Day of year (0 .. 365; January 1 = 0), -1 if not supported.
8	Positive value if daylight saving time is in effect; 0 if daylight saving time is not in effect; negative value if status is unknown.

Note

On Windows CE informations about day of year (index 7) and daylight saving time are currently not supported, i.e. both values will be -1.

See also

time, **ctime**, **localtime**, **mktime**

References **vector()**.

◆ localtime()

```
vector localtime ( long t )
```

Makes a localized time point from C-timestamp and split it into different parts.

Behavior of function is similarly to same named C function. However, because B++ doesn't support structure types it uses a vector for result.

Parameters

t Timestamp to localize, on 16-Bit-platforms argument type is *int* (32 bits).

Returns

vector of following components:

Index	Value
0	Seconds after minute (0 .. 59).
1	Minutes after hour (0 .. 59).
2	Hours after midnight (0 .. 23).
3	Day of month (1 .. 31).
4	Month (0 .. 11; January = 0).
5	Year (current year minus 1900).
6	Day of week (0 .. 6; Sunday = 0).
7	Day of year (0 .. 365; January 1 = 0), -1 if not supported.
8	Positive value if daylight saving time is in effect; 0 if daylight saving time is not in effect; negative value if status is unknown.

Note

On Windows CE informations about day of year (index 7) and daylight saving time are currently not supported, i.e. both values will be -1.

See also

time, **ctime**, **gmtime**, **mktime**

References **vector()**.

◆ mktime()

```
long mktime ( vector tm )
```

Makes C-timestamp from localized time point.

Behavior of function is similarly to same named C function. However, because B++ doesn't support structure types it uses a vector for argument.

Parameters

tm vector of following componente:

Index	Value
0	Seconds after minute (0 .. 59).
1	Minutes after hour (0 .. 59).
2	Hours after midnight (0 .. 23).
3	Day of month (1 .. 31).
4	Month (0 .. 11; January = 0).
5	Year (current year minus 1900).
6	Day of week (0 .. 6; Sunday = 0).
7	Day of year (0 .. 365; January 1 = 0), -1 if not supported.
8	Positive value if daylight saving time is in effect; 0 if daylight saving time is not in effect; negative value if status is unknown.

Returns

C-timestamp built from *tm*

Note

- In case of invalid argument type or invalid size of *tm*, function throws exception of `string` type.
- Components 6 (Day of week) and 7 (Day or year) from *tm* are ignored for conversion.
- If *tm* is a constant, it will not be modified. Otherwise it is filled by normalized values for all its parts.

See also

time, **ctime**, **localtime**, **gmtime**

References **vector()**.

◆ `strftime()`

```
string strftime ( const string fmt,
                  var          tm )
```

Function converts date and time information from a given calendar time tm to character string according to format string fmt.

Behavior of function is similarly to same named C function.

Parameters

fmt Format string.

The format string consists of zero or more conversion specifiers and ordinary characters (except %). All ordinary characters, including the terminating null character, are copied to the output string without modification. Each conversion specification begins with % character, followed by the character that determines the behavior of the specifier. At least the following format specifiers are available:

Specifier	Meaning
%%	Character %
%a	Abbreviated weekday name
%A	Full weekday name
%b	Abbreviated month name
%B	Full month name
%c	Date and time representation
%d	Day of the month (01-31)
%H	Hour in 24h format (00-23)
%I	Hour in 12h format (01-12)
%j	Day of the year (001-366)
%m	Month as a decimal number (01-12)
%M	Minute (00-59)
%p	Either AM or PM according to the given time value
%S	Second (00-59)
%U	Week number (00-53), Sunday as 1st day of week
%w	Day of week (0-6), 0=Sunday
%W	Week number (00-53), Monday as 1st day of week
%x	Preferred date representation
%X	Preferred time representation
%y	Year without century (00-99)
%Y	Year with century
%Z	Time zone name, empty if no time zone

Depending on platform of runtime there may be more specifiers supported. Please refer to documentation of strftime function in C-runtime of compiler used to build B++ library.

tm Localized time point from C-timestamp (vector of 9 elements like result of localtime) or C-timestamp (like result of **time()** function).

Returns

Converted date and time information on success, otherwise empty string.

Note

In case of invalid argument type exception of string type is thrown.

See also

localtime, **time**, **ctime**

References **var**.

◆ time()

```
long time ( )
```

Get C-timestamp.

Returns

number of seconds elapsed since 1970/01/01 0:00 GMT

Note

Function behaves like **time** function of C standard library, on 16-Bit-platforms return type is *int* (32 bits) instead of *long*.

References **uint()**.

Referenced by **MSG::setTime()**.

◆ timer()

```
uint timer ( )
```

Gets elapsed time.

This function is intended to be used for time measurements.
On Windows-CE platform it returns elapsed time since start of system, on all other platforms elapsed time since start of current program (more precise: time since start of B++ runtime).

Returns

elapsed time in milliseconds

Note

Depending on platform, resolution of timer may be lower than 1 millisecond.

Example:

```
main()
{
   uint t1 = timer();
   // do anything here
   ...
   uint t2 = timer();
   print(t2-t1, " ms needed for operation.\n");
}
```

Definition at line **2189** of file **bppufunc.cpp**.

References **CLK_TCK**.

Referenced by **BppTimerList::Entry::Entry()**, and **BppTimerList::handleTimer()**.

B++ Reference © 2005-2026 by R.-Erik Ebert

All parts of B++ are provided under terms and conditions of MIT license.

Random number functions

B++ predefined functions

Collaboration diagram for Random number functions:



Functions

```
int rand ()
```

Get random number.

```
void srand (int seed)
```

Initialize random number generator.

Detailed Description

Function Documentation

◆ rand()

```
int rand ( )
```

Get random number.

Function computes a pseudo-random number. It behaves like same named C function.

Returns

random number between 0 and 0x7FFF

Note

Multiple calls to this function will produce an approximately uniformly distributed series of random numbers. However, random number generator should be initialized using **srand** function to prevent from generating same series for each run of your program.

See also

srand

◆ srand()

```
void srand ( int seed )
```

Initialize random number generator.

Function initializes random number generator by value of *seed*. It behaves like same named C function.

Parameters

seed value to initialize random number generator with

Note

On start of program random number generator is initialized by value 1. In order to get "really random" series of random numbers, usually a time depended value for *seed* is used in calls to this function.

Random number generator example:

```
main(;i,j,l,seed)
{
   // create 5 different series of random numbers
   for (l = 0; l< 5; ++l)
   {
      seed = timer(); // use elapsed time for initialization
      srand(seed);    // initialize generator
      print("initial value; ",seed,"\n");
      // create series of 100 numbers and print
      // 10 numbers per line
      for (i=0; i<10; ++i)
      {
         for (j=0; j<10; ++j) print(rand()," ");
         print("\n");
      }
   }
}
```

See also

rand, **timer**, **print**

B++ Reference © 2005-2026 by R.-Erik Ebert

All parts of B++ are provided under terms and conditions of MIT license.

Program related functions

B++ predefined functions

Collaboration diagram for Program related functions:



Functions

`vector getargs ()`
Get command line arguments passed to B++.
`vector getusrargs ()`
Get command line user arguments passed to B++.
`int argcnt ()`
Get number of function arguments.
`var arg (int idx)`
Get function argument of caller by index.
`vector argv ()`
Gets vector of functions actual parameters.
`var call (var f, vector args=null)`
Calls function *f* with actual parameters from *args*.
`void callafter (const var callable, vector args=null)`
Installs an attached function extension that is executed immediately after the calling function exits.
`void callfail (const var callable, vector args=null)`
Installs an attached function extension that is executed immediately after the calling function exits, if the calling function threw an exception.
`void callsucc (const var callable, vector args=null)`
Installs an attached function extension that is executed immediately after the calling function exits, if the calling function did not throw an exception.
`bool cleartimer (intptr handle)`
Removes single timer.
`void cleartimers ()`
Removes all timers.
`var clone (const var val)`
Clones a value of any type.
`bool compile (const string src, const string srcname="", bool debug=false)`
Compiles source code at runtime.
`bool compile (FILE fp, const string srcname="", bool debug=false)`
Compiles source file at runtime.
`function compilefn (const string src, const string args="", const string rtype="", var ownerClass=null, const string srcname="", bool debug=false)`
Compiles source of a single function or element function at runtime.
`var litexpr (const string src, const string srcname="")`
Parses a literally expression or single literally value from string.
`bool loadmodule (const string filename)`
Load module at runtime.
`intptr`

`settimer (uint milliseconds, const var callable, bool cyclic=false, vector args=null)`
Installs timer.

Detailed Description

Function Documentation

◆ arg()

```
var arg ( int idx )
```

Get function argument of caller by index.

Function returns argument of specified index from calling function. It may be used in conjunction with **argcnt** to construct functions of variable sized argument lists.

Parameters

idx index of argument to get

Returns

callers function argument for index *idx*. If argument is `const`, result is `const` too.

Note

In case of invalid *idx* value an error is reported.

Example for using arg and argcnt functions

```
// define a function
var f(var x)   // only first argument has got a name
{
   uint n = argcnt();            // get number of arguments passed
   print("\n",n," arguments:\n");
   for (uint i=0;i<n;++i)
      print("\t",arg(i));   // output argument values
   print("\n---\n");
   return x; // return value of named (first) argument
}

main()
{
   print(f(1,3,"test"));
   // should print
   // 3 arguments:
   //    1  3  test
   // ---
   // 1
}
```

See also

argcnt

References **var**.

Referenced by **BppString::BppString()**, **_tmain()**, **Bp2String_append()**, **Bp2Point::ctor()**, **XMLNode::foreach()**, and **Bp2WinMsgHandler::setOwner()**.

◆ argcnt()

```
int argcnt ( )
```

Get number of function arguments.

Function retrieves number of arguments passed to calling function.

Returns

number of arguments of caller

> **Note**
>
> If called from `main` function in a command line program, all command line arguments (including programm name) are passed as string parameters to function, i.e. arguments are same as result of **getargs** function.

See also

arg, **getargs getusrargs**

Referenced by **EditCtrl::EditCtrl()**, **StatusBar::StatusBar()**, **Window::Window()**, **GDI::Region::combineRegions()**, and **XMLNode::foreach()**.

◆ argv()

```
vector argv ( )
```

Gets vector of functions actual parameters.

Function returns vector of all arguments passed to calling function. It may be used in conjunction with **call** function in order to pass arguments to another function.

Returns

Vector containing all actual parameters of caller. If **argv** is called from an element function, first element of result is the calling object. Constant state of arguments is preserved in result vector.

See also

argcnt, **arg**

References **vector()**.

Referenced by **BppCharPtrArray::BppCharPtrArray()**, **BppCharPtrArray::BppCharPtrArray()**, **_tmain()**, **GDI::Region::combineRegions()**, **BppInterpreter::execute()**, and **BppVirtualMachine::execute()**.

◆ call()

```
var call ( var    f,
           vector args = null )
```

Calls function *f* with actual parameters from *args*.

Function does a generic call to another function passing elements of *args* as actual parameters to called function. It is mainly intended to dispatch variable number of arguments to any function.

Parameters

f Function or functor object to be invoked

args Container of arguments to pass to *f*.

Returns

Returns return value of *f* if any, otherwise result is undefined.

Note

Function may be used for call to element functions of objects too. In that case first element of *args* must be the object itself.
If *f* is an object, it must define **operator()** (or element function **OP_CALL**).

Examples for using call function

Building a C-like printf function:

```
int printf(string format)
{
   vector v = argv();
   print(call(newstring,v));
   return int(size(v))-1;
}
```

Simple implementation of a .NET-like delegate

```
class Delegate
{
private:
   object _obj;
   function _func;
public:
   Delegate(object obj, function func)
   {
      _obj = obj;
      _func = func;
   }
   operator ()() {
     vector v = argv();
     if (_obj) v[0] = _obj; // Object is first parameter for call.
     else verase(v,0,1);
     return call(_func,v);
  }
}
```

References **var**, and **vector()**.

Referenced by **XMLNode::foreach()**.

◆ callafter()

```
void callafter ( const var callable,
                 vector    args = null )
```

Installs an attached function extension that is executed immediately after the calling function exits.

Parameters

callable Function or functor object to be called when the function exits.

args Container of arguments to be passed to *callable*.

Note

For *callable*, a simple function, a member function, or a function object can be specified. If *callable* is a member function, the first value passed in *args* must be a reference to the corresponding object.

References **var**, and **vector()**.

◆ callfail()

```
void callfail ( const var callable,
                vector   args = null )
```

Installs an attached function extension that is executed immediately after the calling function exits, if the calling function threw an exception.

Parameters

callable Function or functor object to be called when the function exits.

args Container of arguments to be passed to *callable*.

Note

- For *callable*, a simple function, a member function, or a function object can be specified. If *callable* is a member function, the first value passed in *args* must be a reference to the corresponding object.
- The exception is passed to the caller of the current function. It cannot be handled by the attached extension.

References **var**, and **vector()**.

◆ callsucc()

```
void callsucc ( const var callable,
                vector   args = null )
```

Installs an attached function extension that is executed immediately after the calling function exits, if the calling function did not throw an exception.

Parameters

callable Function or functor object to be called when the function exits.

args Container of arguments to be passed to *callable*.

Note

For *callable*, a simple function, a member function, or a function object can be specified. If *callable* is a member function, the first value passed in *args* must be a reference to the corresponding object.

References **var**, and **vector()**.

◆ cleartimer()

```
bool cleartimer ( intptr handle )
```

Removes single timer.

The function removes (uninstalls) a synchronous timer previously installed by **settimer**.

Parameters

handle Handle (ID) of timer to be removed.

Returns

true on success

References **intptr()**.

◆ cleartimers()

```
void cleartimers ( )
```

Removes all timers.

The function removes (uninstalls) all synchronous timers previously installed by **settimer**.

◆ clone()

```
var clone ( const var val )
```

Clones a value of any type.

Function creates a clone of *val*. It may be used to create full (deep) copy of any container type, string or object instance. For all other types (including weak references) it simply returns argument itself.

Parameters

val Value to clone.

Returns

cloned value

> **Note**
>
> Function also creates a copy of bytecode functions (type BVT_BYTECODE) having static variables. In this case the result is another function with same implementation as the original but own instances of static variables.

References **val**, and **var**.

Referenced by **BppValue::BppValue()**, **XMLNode::cloneNode()**, **LOGPALETTE::getStructDesc()**, **RGNDATA::getStructDesc()**, **TTPOLYCURVE::getStructDesc()**, **GDI::DC::mkClone()**, **GDI::GDIObject::mkClone()**, and **BppInterpreter::safeAssign()**.

◆ compile() [1/2]

```
bool compile ( const string src,
               const string srcname = "",
               bool         debug = false )
```

Compiles source code at runtime.

Function compiles source code at runtime and integrates it to running program.

Parameters

src string containing B++ source code to compile

srcname Name of source. If specified *srcname* is included in debug output in case of an error during compiling.

debug If *true*, generated bytecode contains additional information about local variable names and line nmbers in source code for debugging purposes.

Returns

`true` if *src* was compiled successfully, otherwise `false`

Note

- Other than `#include` processing instructon code is compiled and integrated at runtime.
- This function may be called anywhere in your program. However, result is mapped into globally context replacing already existing symbols of same names (if any) by new versions. That allows modifying behavior of running programm (e.g. depending on any conditions).

Example:

```
// define a test function
testfunc(n) {
   print("testfunc(\"",n,"\") called.\n");
}

main(;fstr) {
   // call testfunc
   testfunc("Hello World");
   // create source of alternate function
   fstr =
     R"(testfunc(n) {
         print("alternate version of testfunc called:",n,"\n");
       })";
   if (compile(fstr))
      testfunc("Hello World");
}
```

See also

compile(FILE,string), **loadmodule**, **compilefn**

◆ compile() [2/2]

```
bool compile ( FILE         fp,
               const string srcname = "",
               bool         debug = false )
```

Compiles source file at runtime.

Function compiles source file at runtime and integrates it to running program.

Parameters

fp file opened for binary read

srcname Name of source. If specified *srcname* is included in debug output in case of an error during compiling.

debug If *true*, generated bytecode contains additional information about local variable names and line nmbers in source code for debugging purposes.

Returns

`true` if contents of *fp* was compiled successfully, otherwise `false`

Note

- Other than for `#include` processing instructon code is compiled and integrated at runtime.
- This function may be called anywhere in your program. However, result is mapped into globally context replacing already existing symbols of same names (if any) by new versions. That allows modifying behavior of running programm (e.g. depending on any conditions).

See also

compile(string,string), **loadmodule**, **compilefn**

◆ compilefn()

```
function compilefn ( const string src,
                     const string args = "",
                     const string rtype = "",
                     var          ownerClass = null,
                     const string srcname = "",
                     bool         debug = false )
```

Compiles source of a single function or element function at runtime.

Function compiles source code of a function body at runtime without integration into running program.

Parameters

src	Source code of function body to compile.
args	Contents of function argument list in same notation as for a 'normal' function decvlaration.
rtype	Name of return type of function. Empty value means no specific type.
ownerClass	Type or class name of owner class, i.e. class to create element function for. If not specified, a simple function is created. In case of class name, argument may be <ii>const</ii>.
srcname	Name of source. If specified *srcname* is included in debug output in case of an error during compiling.
debug	If *true*, generated bytecode contains additional information about local variable names and line nmbers in source code for debugging purposes.

Returns

Compiled function (or element function) if contents of *src* was compiled successfully, otherwise `null`.

Note

- Function allows dynamic craetion of any function at runtime and assign it e.g. to a variable or call it directly without sideeffects for code of running program.
- Inside body (contents of *src*) all globally defined symbols (but not local symbols of caller) and members of ownerclass (if specified by *ownerClass*) can be accessed.
- If a member function is created, result must be called in context of an object of *ownerClass* type.

Example

```
// Evaluate expressions at runtime
var evalExpr(string expr)
{
   var f = compilefn("return ("+expr+");");
   if (f) return f();
   return null;
```

```
}
void main()
{
   while (true)
   {
     print("Expression: ");
     string s = gets();
    if (size(s) == 0) break;
    print(" Result: ",evalExpr(s),"\n")
  }
}
```

compile(string,string), loadmodule

References **var**.

◆ getargs()

```
vector getargs ( )
```

Get command line arguments passed to B++.

Function builds a vector of all command line arguments passed to B++ on run. When B++ command line version is running, first argument is always the program name itself (e.g. bp2.exe).

Returns

Vector of all command line arguments. All elements of returned vector are of `string` type.

> **Note**
>
> When B++ is used as a library, i.e. inside any program, owner program is resposnible for passing arguments to B++ if needed.

See also

getusrargs, **BppVirtualMachine::execute**

References **vector()**.

◆ getusrargs()

```
vector getusrargs ( )
```

Get command line user arguments passed to B++.

Function builds a vector of all command line user arguments passed to B++ on run. First element of result vector is first command line argument behind hash sign(#) in command line.

Returns

Vector of all command line user arguments. All elements of returned vector are of `string` type.

> **Note**
>
> When B++ is used as a library, i.e. inside any program, owner program is resposnible for passing arguments to B++ if needed.

Example for using getargs and getusrargs functions:

```
main(;vec1,vec2,n,i)
{
   vec1 = getargs();    // get all arguments
   vec2 = getusrargs(); // get user arguments
   print("args:\n");
   n = vecsize(vec1);
   for (i=0;i<n;++i) print(vec1[i],"\n");
   print("usrargs:\n");
   n = vecsize(vec2);
   for (i=0;i<n;++i) print(vec2[i],"\n");
}
```

See also

getargs, **BppVirtualMachine::execute**

References **vector()**.

◆ litexpr()

```
var litexpr ( const string src,
              const string srcname = "" )
```

Parses a literally expression or single literally value from string.

Parameters

src Source string to parse

srcname Source name for use in error messages

Returns

Result of expression in *src*

> **Note**
>
> If parsing fails, i.e. in case of invalid expression, exception of type `string` containing error description is thrown.

References **var**.

Referenced by **JSON::parse()**.

◆ loadmodule()

```
bool loadmodule ( const string filename )
```

Load module at runtime.

Function loads a pre-compiled byte code module at runtime and integrates it to running program.

Parameters

filename Name of file containing module

Returns

`true` on success, otherwise `false`

> **Note**
>
> Other than for `#use` processing instructon code is integrated at runtime.
> This function may be called anywhere in your program. However, result is mapped into globally context replacing already existing symbols of same names (if any) by new versions. That allows modifying behavior of running programm (e.g. depending on any conditions).

See also

compile(string,string), compile(FILE, string)

◆ settimer()

```
intptr settimer ( uint      milliseconds,
                  const var callable,
                  bool      cyclic = false,
                  vector    args = null )
```

Installs timer.

The function installs an synchronous timer.

Parameters

milliseconds	Duration (interval) in milliseconds.
callable	Function, element function or functor that is to be executed after the specified time interval.
cyclic	Flag for the execution mode; the value `true` means cyclic (repeated) execution, `false` a one-time execution.
args	Container of arguments for *`callable`*.

Returns

Handle (ID) of the installed timer.

Note

- If an element function (method pointer) is specified for *`callable`*, *`args`* must contain the reference to the associated object instance as its first value.
- If a time interval of zero milliseconds is specified, no timer is installed. The result is also zero.
- The return value can be used to remove an installed timer using **cleartimer** function.

References **intptr()**, **uint()**, **var**, and **vector()**.

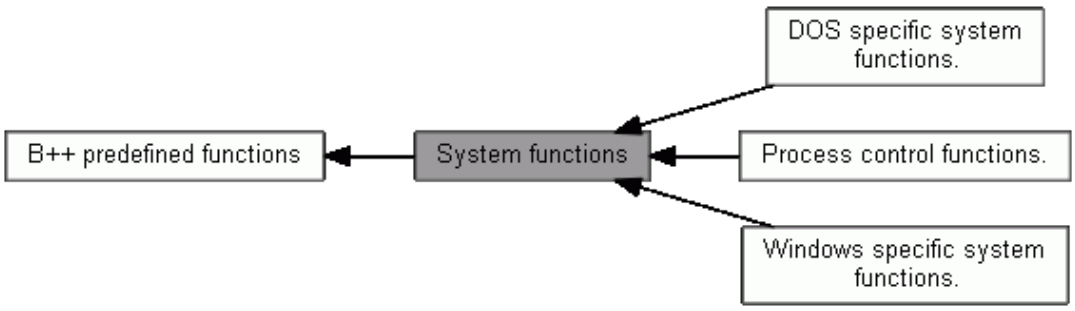
B++ Reference © 2005-2026 by R.-Erik Ebert

All parts of B++ are provided under terms and conditions of MIT license.

System functions

B++ predefined functions

Collaboration diagram for System functions:



Topics

Process control functions.
The functions in this section are used to execute and control child processes.
Windows specific system functions.
All functions of this section are available in Windows builds only.
DOS specific system functions.
All functions of this section are available in DOS builds only.

Functions

intptr addr (const **var** any)
Get address of value data.
bool chdir (const **string** dirpath)
Changes current working directory.
string fullpath (const **string** relpath)
Makes fully qualified path to file or directory.
string getcwd ()
Gets current working directory.
intptr memmove (**intptr** dst, **intptr** src, **uint** n)
Does raw copy of specified block in memory.
uint memsize ()
Gets size of available memory.
string setlocale (**int** category, const **string** locale)
Sets or gets current runtime localization scheme.
void **sleep** (**uint** milliseconds)
Suspends execution for specified time.

Environment access functions

> **Note**
>
> For Windows CE builds environment is emulated in registry (key HKLM\Environment). There is only one globally defined environment, i.e. modification of an environment variable may

affect other processes too.
In DOS and Windows Desktop builds process local environment area is used.

```
dictionary getenv ()
```
Gets dictionary containing copy of environment variables.

```
string getenv (const string varname)
```
Get value of specified environment variable.

```
bool setenv (const string varname, const string value)
```
Set value of an environment variable.

Error reporting functions

```
int errno ()
```
Gets code of last system error.

```
int errno (int err)
```
Sets code of last system error.

```
int nerr ()
```
Gets number of system error codes.

```
string strerror ()
```
Gets last system error message.

```
string strerror (int errcode)
```
Gets error message for specified code.

Dynamic library (DLL) functions.

Note

For DOS builds handling of Dynamic Link Libraries (DLLs) is supported using Kiez*Soft* DOSDLL library. Please refer to documentation of this library for restrictions and detailed using instructions.

```
intptr LoadLibrary (const string dllname)
```
Load a dynamic library.

```
int FreeLibrary (intptr hdll)
```
Release dynamic library.

```
intptr GetProcAddress (intptr hdll, const string name)
```
Get address of function or data area exported from dynamic library.

```
intptr CallLibFunc (intptr procaddr,...)
```
Call a function inside dynamic library.

Callback support functions.

Beginning at version 1.5 B++ provides two sets of native callback functions intended to use as callback arguments of system API functions.

Callbacks are handled by user defined functor objects, i.e, by object instance providing `operator()`.
First set assumes last callback argument to be user data containing pointer to the functor object. This is the typical callback type used mostly by Win32 API functions. This set allows callbacks up to 8 arguments (including functor pointer) and any number of callbacks.
Second set supports callbacks without user data. Therefore functor used for callback handler must be registered from client code. This set is restricted to a maximum of 4 arguments per callback and 5 callbacks of same type (i.e. number of arguments) per process.
In both cases prototypes of native callback functions assume all parameters and return value to be of <ttintptr type. This restricts size of arguments and return value to pointer size, i.e., 4 bytes or 8 bytes (in 64-Bit builds).

> **Note**
>
> In DOS builds callback support is disabled by default, because system calls usually are done by interrupt calls and B++ is not intended to write interrupt service routines. If needed, B++ must be compiled with defined symbol USECALLBACK in compiler options.

`intptr GetCallbackFunc (uint numArgs)`
Gets pointer to native callback function for use as parameter of system functions.
`intptr GetStaticCallbackFunc (uint numArgs, uint idxInstance)`
Gets pointer to static native callback function for use as parameter of system functions.
`void SetStaticCallbackHandler (uint numArgs, uint idxInstance, var handler)`
Installs functor object for one of static callback functions.

Detailed Description

Function Documentation

◆ addr()

```
intptr addr ( const var any )
```

Get address of value data.

Function returns address of data for value *val*. It is intended for use with system specific functions which require variable pointers for arguments.

Parameters

any value or variable of any type except `null`

Returns

pointer representing value address. The returned value depends on data type of *any* as follows:

- For types `string`, `buffer` and `charbuffer` function returns start address of data area. The result can be interpreted as an array or string in C-style.
- For all numeric types current function does same as explicit conversion into `intptr` type.
- In case of any other type, value itself is returned as a pointer. This allows passing of reference data (e.g. file pointer or object) to native functions.

References **intptr()**, and **var**.

Referenced by **GDI::DriverDC::DriverDC()**, **GDI::InfoDC::InfoDC()**, **GDI::MetaDC::MetaDC()**, **GDI::AddFontResource()**, **Menu::AppendMenu()**, **StatusBar::applyPaneSizes()**, **GDI::EnhMetaFile::copy()**, **GDI::Bitmap::create()**, **GDI::Font::create()**, **GDI::Bitmap::createFromFile()**, **GDI::DC::enumBrushes()**, **GDI::DC::enumFontFamiliesEx()**, **GDI::DC::enumFonts()**, **GDI::DC::enumPens()**, **GDI::EnhMetaFile::enumRecords()**, **GDI::EnhMetaFile::fromBits()**, **GDI::EnhMetaFile::fromFile()**, **GDI::Bitmap::getBitmapBits()**, **GDI::DC::getCharacterPlacement()**, **GDI::EnhMetaFile::getDescription()**, **GDI::EnhMetaFile::getEnhMetaFileBits()**, **EditCtrl::getInsertionPosition()**, **GDI::DC::getMiterLimit()**, **GDI::DC::getTextExtent()**, **GDI::DC::getTextFace()**, **Menu::InsertMenu()**, **XMLNode::isAncestor()**, **BppValue::isNull()**, **XMLNode::isSameNode()**, **MessageBox()**, **GDI::BrushEnumeratorBase::operator()()**, **GDI::FontEnumeratorBase::operator()()**, **GDI::PenEnumeratorBase::operator()()**, **GDI::RemoveFontResource()**, **GDI::Bitmap::setBitmapBits()**, **GDI::DC::setMiterLimit()**, **StatusBar::setPaneText()**, **GDI::DC::stretchDIBits()**, **GDI::DC::textOut()**, and **GDI::DC::textOutExt()**.

◆ CallLibFunc()

```
intptr CallLibFunc ( intptr procaddr,
                           ... )
```

Call a function inside dynamic library.

Function invokes library function at address specified by *procaddr*.

Parameters

procaddr Adress of function to invoke. Use **GetProcAddress** to get address from function name.

... Comma separated list of parameters needed by called function.

Returns

result of function call

Note

- Because B++ knows nothing about function called inside the library, there is no way to do any checks. So programmer is responsible to take care about passing correct parameters.
- All parameters must be passed as values of same width as **intptr** type, values convertible to **intptr** may be passed directly where for all other types addresses should be used.
- On Windows x86 or DOS platforms functions inside dynamic library must be implemented using C- or STDCALL-calling conventions. Furthermore they should return a result value of **intptr** size.
- Meaning of returned value depends on implemantation of library function. If library function returns nothing, result is undefined.
- Number of function arguments is restricted to a maximum of 16 for platforms other than Windows x86 or DOS.

Example:

In the following we assume a dynamic library "database.dll" implementing some database access functions. This library exports a login function of following C-signature:

```
int login(const char* userName, const char* passwd);
```

This function needs two parameters (user name and password). On success it returns a session id for database access, otherwise 0.
Sample program below shows all steps for call the function.

```
// Example for DLL function call
void main()
{
   // first load dynamic library
   intptr hdll = LoadLibrary("database.dll");
   // get address of exported function
   intptr proc = GetProcAddress(hdll,"login");
```

```
    // now ask user for her/his name and password
    print("Enter your name: ");
    string name = gets();
    print("Enter your password: ");
    string pwd = gets();
    // try to login
    // dont forget to pass addresses of strings
    intptr session = CallLibFunc(proc,addr(name),addr(pwd));
    if (session)
    {
       // do anything
    }
    else
       print("Database access denied!\n");
    // release library
    FreeLibrary(hdll);
}
```

See also

CallCLibFunc, **LoadLibrary**, **GetProcAddress**, **FreeLibrary**, **addr**

References **intptr()**.

Referenced by **Menu::Menu()**, **Menu::AppendMenu()**, **GetDialogBaseUnits()**, **GetSysColor()**, **GetWindowLong()**, **GetWindowLongPtr()**, **Menu::InsertMenu()**, **MessageBox()**, **PostMessage()**, **SendMessage()**, **Window::setMainMenu()**, **SetParent()**, **SetWindowLong()**, and **SetWindowLongPtr()**.

◆ chdir()

```
bool chdir ( const string dirpath )
```

Changes current working directory.

Parameters

dirpath Directory path to set for working directory. This may be an absolute or relative path.

Returns

true on success

Note

- On Windows and MS-DOS systems function may change current drive too, if drive is specified in *dirpath*. If *dirpath* contains drive letter, followed by a colon (:) only, current drive switched switched without changing current directory path on that drive.
- If *`dirpath`* can't be implicitely converted into `string`, an exeption of type `string` is thrown.
- If Function fails, **errno()** can be uset to determine error code.

◆ errno() [1/2]

```
int errno ( )
```

Gets code of last system error.

Function reads errno value of runtime.

Returns

Code of last error occurred while a runtime function call.

Note

Function may be used to get error information in case of failure of a call to function in runtime library, e.g. an I/O-function.

See also

strerror(), **errno(int)**

Referenced by **BppScanner::readNumber()**.

◆ errno() [2/2]

```
int errno ( int err )
```

Sets code of last system error.

Function sets errno value of runtime and returns changed value.

Parameters

err New value of system error code

Returns

Value of system error code after setting (should be same as *err*).

Note

Function may be used to reset pending system error (e.g. after a numerical overflow)

See also

errno()

◆ FreeLibrary()

```
int FreeLibrary ( intptr hdll )
```

Release dynamic library.

Function releases a dynamic library previously loaded by LoadLibrary function.

Parameters

hdll Handle of library to release.

Returns

1 on success, otherwise (i.e. on invalid handle) 0

See also

LoadLibrary

References **intptr()**.

Referenced by **BppVirtualMachine::~BppVirtualMachine()**, **BppVirtualMachine::cloneSymbols()**, **BppVirtualMachine::loadNativeModule()**, and **BppVirtualMachine::reinit()**.

◆ fullpath()

```
string fullpath ( const string relpath )
```

Makes fully qualified path to file or directory.

Function builds fully qualified path for relative path specified in *relpath*.

Parameters

relpath Relative path specification.

Returns

Fully qualified path for *relpath* on success, otherwise empty string.

Note

- If *relpath* is empty or contains a dot only, result is same as for call of **getcwd**.
- On Windows and MS-DOS systems *relpath* may contain a drive specification. If *relpath* contains drive specification (drive letter, followed ba a colon) only, selected path on that drive is returned.

Referenced by **_findLibrary()**, **DllEntry::create()**, and **LoadLibrary()**.

◆ GetCallbackFunc()

```
intptr GetCallbackFunc ( uint numArgs )
```

Gets pointer to native callback function for use as parameter of system functions.

Each of these functions assumes last argument to be a B++ functor, i.e, an object instance providing call operator. When callback is invoked from system function, functor is called with callback arguments (without last argument).
Therefore system function must support user data pointer and call the callback with user data as last argument.
Return type and type of all parameters for callback functions are assumed to be if **intptr** type, i.e. they have size of native pointer type. From view of client code all simple types up to size of **intptr** are usable for arguments and return type. Values of reference types must be passed as adresses, i.e. using **addr()** function.

Parameters

numArgs Number of arguments for callback, including functor object. Currently a value between 1 and 8 is allowed.

Returns

Address of callback function.

Example: Implementation of a windows enumerator for Win32 API

```
// Functor for callback
class WinEnumerator {
   private:
     charbuffer txtbuf = null;
     intptr func = 0;
   public:
   WinEnumerator() {
      txtbuf = newcbuffer(256); // initialize buffer
      func = GetUser32Func("GetWindowText?"); // get address of GetWindowText fu
         nction
   }
   // call operator
   uint operator()(intptr hwnd) {
      // get window text for handle
      CallLibFunc(func,hwnd,addr(txtbuf),size(txtbuf)-1);
      auto str = string(txtbuf); // convert buffer to string
      if (size(str) > 0)
         print(ulong(hwnd),":",str,"\n");
      return 1;
   }
};

void main() {
    intptr pfunc = GetUser32Func("EnumWindows");
    // get callback function of 2 arguments (first for window handle, second for
        enumerator object)
    intptr pcallback = GetCallbackFunc(2);
    WinEnumerator we; // build instance of functor object
    // call EnumWindows with callback function and address of functor object
    CallLibFunc(pfunc,pcallback,addr(we));
}
```

References **intptr()**, and **uint()**.

Referenced by **GDI::DC::enumBrushes()**, **GDI::DC::enumFontFamiliesEx()**, **GDI::DC::enumFonts()**, **GDI::DC::enumPens()**, and **GDI::EnhMetaFile::enumRecords()**.

◆ getcwd()

```
string getcwd ( )
```

Gets current working directory.

Returns

Full path of current working directory.

◆ getenv() [1/2]

```
dictionary getenv ( )
```

Gets dictionary containing copy of environment variables.

Returns

Ordered dictionary containing copies of all environment variable values. Keys are the names of the environment variables.

Example

```
// list names and values of all environment variables:
void main()
{
   auto env = getenv();
   vector names = dictgetkeys(env);
   uint n = size(names);
   for (uint i=0;i<n;++i) print(names[i],"=",env[names[i]],"\n");
}
```

Note

Function won't work for Windows CE (always returns empty dictionary).

See also

getenv(string), **setenv**

References **dictionary()**.

Referenced by **_TmpLocale::_TmpLocale()**.

◆ getenv() [2/2]

```
string getenv ( const string varname )
```

Get value of specified environment variable.

Function returns value of environment variable *varname*.

Parameters

varname Name of environment variable to get value from.

Returns

Copy of environment variable value if it exists or empty string otherwise.

Example

```
// list all parts of "Path" environment variable:
void main()
{
   string path = getenv("Path");
   vector parts = strsplit(path,";");
   uint n = size(parts);
   for (uint i=0;i<n;++i) print(parts[i],"\n");
}
```

See also

getenv(), **setenv**

◆ GetProcAddress()

```
intptr GetProcAddress ( intptr      hdll,
                        const string name )
```

Get address of function or data area exported from dynamic library.

Function returns address of symbol *name* (function or data area) exported by a dynamic library.

Parameters

hdll library handle

name name of exported symbol

Returns

address of symbol on success, otherwise 0

Note

- Returned function address can be used for argument of **CallLibFunc** call.
- In case of a data area returned address points to start of area.
- If Function fails, i.e. if requested symbol doesn't exist, 0 is returned.
- In Windows builds *name* may contain a question sign (?). This sign is replaced by 'W' in unicode versions and by 'A' in non-unicode versions.

See also

LoadLibrary, **CallLibFunc**

References **intptr()**.

Referenced by **BppVirtualMachine::loadNativeModule()**.

◆ GetStaticCallbackFunc()

```
intptr GetStaticCallbackFunc ( uint numArgs,
                               uint idxInstance )
```

Gets pointer to static native callback function for use as parameter of system functions.

Parameters

numArgs Number of arguments for callback. Currently a value between 0 and 4 is allowed.

idxInstance Index of instance of callback function; value between 0 and 4 allowed.

Returns

Address of callback function.

Note

Before use of callback, client code must install a functor object by call to SetStaticCallbackhandler.

See also

SetStaticCallbackHandler

References **intptr()**, and **uint()**.

Referenced by **GDI::DriverDC::setAbortHandler()**.

◆ LoadLibrary()

```
intptr LoadLibrary ( const string dllname )
```

Load a dynamic library.

Function loads dynamic library specified by *dllname* into current process and returns a handle to it.

Parameters

dllname Name of library. The name specified is the file name of library module (including extension if any). It may contain a relative or absolute path specification.

Returns

handle to loaded library or 0 (if not found)

Note

- The returned handle is needed as argument for following calls to **GetProcAddress** and **FreeLibrary** functions.
- A library loaded by **LoadLibrary** function should be released using **FreeLibrary** if no longer needed.
- If no absolute full path is specified in *dllname*, a system dependend serach for library is done. Please refer to platform documentation (or DOSDLL manual for DOS builds) for search strategy used.
- If *dllname* contains an asterisk (*), the asterisk character is replaced by build specific name extension, consisting of following (optional) parts:
  - u : marks UTF-16 build
  - d : marks Debug build
  - _64 : marks build for 64 Bits platform

Example for using dynamic library functions:

```
// This sample program works for Win32 Desktop platforms.
// For Windows CE load coredll.dll instead of user32.dll .
#define MB_ICONINFORMATION 0x40
// function shows a simple message box
ShowMessage(string title, string msg)
{
   intptr hinst = LoadLibrary("user32.dll");
   // ? will be replaced by 'W' or 'A', depending if unicode build or not
   intptr fptr = GetProcAddress(hinst,"MessageBox?");
   // use addr function to get C-character pointers from strings
   CallLibFunc(fptr,0,addr(msg),addr(title),MB_ICONINFORMATION);
   FreeLibrary(hinst);
}

main()
{
   ShowMessage("Hello World","This is a message\nfrom B++.");
}
```

See also

FreeLibrary, **GetProcAddress**, **CallLibFunc**, **addr**

References **intptr()**.

Referenced by **BppVirtualMachine::loadNativeModule()**.

◆ memmove()

```
intptr memmove ( intptr dst,
                 intptr src,
                 uint   n )
```

Does raw copy of specified block in memory.

Function behaves like same named C function, i.e. it copies *n* bytes from *src* to *dst*, where overlapping of memory areas is allowed.

Parameters

dst Destination address of copy operation

src Source address of copy operation

n Number of bytes to copy

Returns

Destination address (same as *dst*)

Note

- Function does implicit cast of argument types.
- On DOS platforms function allows copying blocks larger than 64kB too.
- In case of invalid arguments or an error while copy operation, it throws exception of `string` type.
- **Caution:** Function allows manipulation of any memory area and may crash system!

References **intptr()**, and **uint()**.

Referenced by **movemem()**, **GDI::BrushEnumeratorBase::operator()()**, **GDI::EnhMetaFile::EnumProc::operator()()**, **GDI::FontEnumeratorBase::operator()()**, **GDI::PenEnumeratorBase::operator()()**, **BppCharBuffer::operator+=()**, and **BppCharBuffer::operator=()**.

◆ memsize()

```
uint memsize ( )
```

Gets size of available memory.

Function size of largest free memory block on heap available for allocation.

Returns

Number of free bytes in memory.

Note

- Because result is of `uint` type, maximum return value is 2^{32}-1. On 64 Bit systems real amount of available memory may be larger.
- On systems using virtual memory (Windows, Linux ...) the result is the virtual memory size, not physical.

References **uint()**.

◆ nerr()

```
int nerr ( )
```

Gets number of system error codes.

Returns

Number of error codes defined by system.

Note

Function may be used to validate an error code or enumerate system error messages.

Example for enumerate system error codes:

```
void main()
{
    int n = nerr();
    for (int i=0;i<n;++i)
       print(i,"\t",strerror(i),"\n");
}
```

See also

strerror(int errcode)

◆ setenv()

```
bool setenv ( const string varname,
              const string value )
```

Set value of an environment variable.

Function defines a new environment variable or changes value of an existing one.

Parameters

varname Name of environment variable so define or change.

value Value to set for environment variable. If specified value is an empty string, variable is removed from environment.

Returns

`true` on success, otherwise `false`

See also

getenv

◆ setlocale()

```
string setlocale ( int          category,
                   const string locale )
```

Sets or gets current runtime localization scheme.

Function behaves like same named C function (and internally uses that function).

Parameters

category Category to set/get localization parameters for. Valid values are

- `LC_ALL` (0): all categories
- `LC_COLLATE` (1): string comparison
- `LC_CTYPE` (2): character fuctions
- `LC_MONETARY` (3): currency format
- `LC_NUMERIC` (4): numeric formats
- `LC_TIME` (5): time format

locale Locale string, e.g. "en-US", "German_Germany.437", ".OCP", ".UTF-8" or "C". Empty string means 'System default settings', `null` returns current settings without any changes.

Returns

String of settings after the call or empty string in case of an error.

Note

- Function (especially valid values for *locale* string) depends on Operating System. On Windows systems, if the character string contains a character set description following a dot, then a sequence of digits is interpreted directly as a code page ID. Otherwise, the string is interpreted as a character set name, and an attempt is made to determine the corresponding code page ID. Conversely, on Unix systems, if a sequence of digits is provided, an attempt is made to determine the corresponding character set name.
- On DOS-Platforms it is useless, because only "C"-locale is available.
- The "C" locale is the default setting that takes effect when the bytecode interpreter starts up.

Referenced by **_TmpLocale::_TmpLocale()**, and **_TmpLocale::~_TmpLocale()**.

◆ `SetStaticCallbackHandler()`

```
void SetStaticCallbackHandler ( uint numArgs,
                                uint idxInstance,
                                var  handler )
```

Installs functor object for one of static callback functions.

Parameters

uint numArgs Number of arguments for callback. Currently a value between 0 and 4 is allowed.

idxInstance Index of instance of callback function; value between 0 and 4 allowed.

handler Functor object used to handle callback. Argument can be an `object`, a weak reference to object or address (type intptr) of object.

Note

- If *handler* is `object`, its reference counter is incremented, i.e. it can't be released until it is unregistered, either by set another handler or call SetStaticCallbackHandler with `null` as `handler` argument.
- If the *handler* is a weak reference or an address, the client code must make sure that the handler exists for as long as it is needed.
- In case of invalid values for *numArgs* or *idxInstance* an exception of `string` type is raised.

See also

GetStaticCallbackFunc

References **uint()**, and **var**.

Referenced by **GDI::DriverDC::setAbortHandler()**.

◆ sleep()

```
void sleep ( uint milliseconds )
```

Suspends execution for specified time.

Function stops current thread (process in DOS) for specified number of milliseconds.

Parameters

milliseconds Duration in milliseconds.

References **uint()**.

Referenced by **sleep_ms()**.

◆ strerror() [1/2]

```
string strerror ( )
```

Gets last system error message.

Function calls **errno()** and returns error message accociated to its result.

Returns

Description of last error occurred while a runtime function call.

Note

Function may be used to get error information in case of failure of a call to function in runtime library, e.g. an I/O-function.

See also

errno(), **strerror(int errcode)**

◆ strerror() [2/2]

```
string strerror ( int errcode )
```

Gets error message for specified code.

Function returns error message accociated to *errcode*.

Parameters

errcode Code of error to get description for

Returns

Description of specified error code. On invalid coed it returns "Unknown error".

See also

errno(), **strerror()**

B++ Reference © 2005-2026 by R.-Erik Ebert

All parts of B++ are provided under terms and conditions of MIT license.

Process control functions.

B++ predefined functions » System functions

The functions in this section are used to execute and control child processes. More...

Collaboration diagram for Process control functions.:



Functions

`bool existspid (int pid)`
Checks if specified process exists.
`bool kill (int pid)`
Terminates the specified process.
`FILE popen (const string cmd, const string mode)`
Creates a pipe and executes a command.
`int pclose (FILE stream)`
Waits for processing command specified on call to **popen** and closes the stream on the associated pipe.
`int spawn (uint mode, const string cmd, const string args=null)`
Spawns and runs a child process.
`int system (const string cmd, bool ignoreErr=true)`
Runs a command line as a synchronous child process.
`int waitpid (int pid)`
Waits for the specified process to terminate and returns its result.

Detailed Description

The functions in this section are used to execute and control child processes.

Platform-specific details are noted in the descriptions of the individual functions.

Function Documentation

◆ existspid()

```
bool existspid ( int pid )
```

Checks if specified process exists.

Parameters

pid Identifier of the process whose existence is to be checked. Typically, this is the result of a preceding call to **spawn** function.

Returns

If the process identified by the *pid* exists, the result is *true*; otherwise, *false*.

Note

- This function is not available under MS-DOS because asynchronous processes are not supported in that environment.
- The *pid* argument is strongly typed; therefore, an implicit type conversion is performed if necessary.
  In case of an invalid argument type, an exception of type `string` is thrown.

◆ `kill()`

```
bool kill ( int pid )
```

Terminates the specified process.

Parameters

pid Identifier of the process to be terminated. Typically, this is the result of a preceding call to **spawn** function.

Returns

If the process identified by the *pid* exists, the result is *true*; otherwise, *false*.

Note

- This function is not available under MS-DOS because asynchronous processes are not supported in that environment.
- The *pid* argument is strongly typed; therefore, an implicit type conversion is performed if necessary.
  In case of an invalid argument type, an exception of type `string` is thrown.
- It is not guaranteed that the process has actually terminated by the time the function returns.
  A program that relies on this for subsequent execution should call #waitpid(pid) in addition to #kill(pid).

Referenced by **killProcess()**, and **processExists()**.

◆ pclose()

```
int pclose ( FILE stream )
```

Waits for processing command specified on call to **popen** and closes the stream on the associated pipe.

Parameters

stream Return value from the previous call to **popen**.

Returns

Returns the exit status of the command, or -1 if an error occurs.

Note

- On DOS platform or Windows applications without console, function is emulated using a temporary file and a call to **system** function.
- If function fails, **errno()** can be called to get error information.

Referenced by **pipeClose()**.

◆ popen()

```
FILE popen ( const string cmd,
             const string mode )
```

Creates a pipe and executes a command.

Parameters

cmd Command to be executed.

mode Mode of the returned stream, may contain following characters:

- r : The calling process can read the spawned command's standard output using the returned stream.
- w : The calling process can write to the spawned command's standard input using the returned stream.
- t : Open in binary mode.
- b : Open in text mode.

Returns

Returns a stream associated with one end of the created pipe. The other end of the pipe is associated with the spawned command's standard input or standard output. The function return NULL on an error.

Note

- On DOS platform or Windows applications without console, function is emulated using a temporary file and a call to **system** function.
- If function fails, **errno()** can be called to get error information.

Referenced by **pipeOpen()**.

◆ spawn()

```
int spawn ( uint        mode,
            const string cmd,
            const string args = null )
```

Spawns and runs a child process.

The function launches the program *cmd* as a child process, using the arguments specified in *args*.

Parameters

mode Execution Mode. The following modes are defined:

Symbol	Value	Description
P_WAIT	0	Synchronous process execution
P_NOWAIT	1	Asynchronous process execution
P_OVERLAY	2	Replace the current process with the child process
P_NOWAITO	3	same as P_NOWAIT
P_DETACH	4	Asynchronous process execution without child process access to standard input and output

cmd Name or path of the program to be executed. If the command does not contain any path separators, a system-specific search is performed across the directories specified in the PATH environment variable. Otherwise, only the specified path is used. If no file extension is specified for *cmd*, MS-DOS and Windows search for executable files with the extensions .com, .exe, .bat and .cmd (in that order; .cmd only on Windows).

args Command-line arguments passed to the program being executed. These are passed to the child process as if they had been entered on the command line. The first argument passed to child process (index 0) is always the value of *cmd*.

Returns

For a successful call, the return value is:

- the exit code of the invoked program for mode P_WAIT;
- always zero for mode P_DETACH;
- the process ID of the created child process for P_NOWAIT or P_NOWAITO.

In mode P_OVERLAY, the function does not return, so there is no result.
If the call fails, the result is -1.

Note

- As MS-DOS does not support asynchronous process execution, only the P_WAIT and P_OVERLAY modes are supported. Any unsupported or invalid values for *mode* are treated as P_WAIT.
  P_DETACH is only supported on Windows.
  On Windows and UNIX systems, invalid or unsupported mode values are treated as P_NOWAIT.

- On Windows systems, the return value corresponds to the process handle rather than the value returned by the Windows API function `GetProcessId`.
- If the call fails, an error code can be determined using **errno()**.
- The function arguments are strongly typed; therefore, implicit type conversion is performed where necessary. In case of invalid argument types, a `string` exception is thrown.
- Unlike the **system** function, spawn does not use the command-line interpreter. Therefore, direct execution of internal shell commands is not possible. However, file redirections and pipes within *args* are permitted.

References **uint()**.

◆ system()

```
int system ( const string cmd,
             bool         ignoreErr = true )
```

Runs a command line as a synchronous child process.

Function invokes command line interpreter with specified command line and returns execution result of call.

Parameters

cmd Command line passed to system. Line may contain name (and optionally path) of programm to run or name of internal command followed by any parameters.

ignoreErr If true (default) call will accept return value -1, otherwise exception of string type is thrown in that case.

Returns

Execution result value of command.

Note

Function is not available in Windows-CE builds. Furthermore call fails, if there is no command shell specified by environment variable (COMSPEC on DOS/Windows, SHELL on UNIX systems) of system (e.g. on DIP-DOS of Atari Portfolio).
If call fails, function throws exception.

◆ waitpid()

```
int waitpid ( int pid )
```

Waits for the specified process to terminate and returns its result.

Parameters

pid Identifier of the process whose termination is to be awaited. This is typically the result of a preceding call to spawn function.

Returns

If the process identified by the *pid* exists, the result is the return value (exit code) of that process. Otherwise, the result is -1.

Note

- This function is not available under MS-DOS because asynchronous processes are not supported in that environment.
- The *pid* argument is strongly typed; therefore, an implicit type conversion is performed if necessary.
  In case of an invalid argument type, an exception of type `string` is thrown.

Referenced by **spawnProcess()**, and **waitForProcess()**.

B++ Reference © 2005-2026 by R.-Erik Ebert

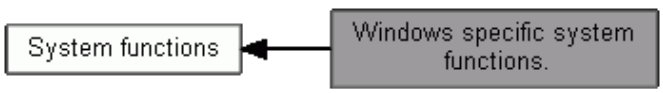
All parts of B++ are provided under terms and conditions of MIT license.

Windows specific system functions.

B++ predefined functions » **System functions**

All functions of this section are available in Windows builds only. More...

Collaboration diagram for Windows specific system functions.:



Functions

```
intptr GetUser32Func (const string name)
```

Get address of function or data area exported from user32.dll library.

```
intptr GetShell32Func (const string name)
```

Get address of function or data area exported from shell32.dll library.

```
intptr GetGdi32Func (const string name)
```

Get address of function or data area exported from gdi32.dll library.

intptr GetKernel32Func (const **string** name)
Get address of function or data area exported from kernel32.dll library.
intptr GetOle32Func (const **string** name)
Get address of function or data area exported from ole32.dll library.
intptr GetCoreFunc (const **string** name)
Alias of GetUser32Func.
uint GetLastError ()
Get last error code.
string GetSysErrorMsg ()
Get readable message for system error code.
string GetSysErrorMsg (**uint** errcode)
Get readable message for system error code.

Detailed Description

All functions of this section are available in Windows builds only.

Function Documentation

◆ GetCoreFunc()

```
intptr GetCoreFunc ( const string name )
```

Alias of GetUser32Func.

Because in most functions of WindowsCE-API are contained in coredll.dll library, there are no user32, shell32, kernel32 and gdi32 libraries in WindowsCE.
This function alias is intended to be used in programms running on WinCE only.

Parameters

name name of exported symbol

Returns

address of symbol on success, otherwise 0

Note

- Returned function address can be used for argument of **CallLibFunc** call.
- In case of a data area returned address points to start of area.
- If Function fails, i.e. if requested symbol doesn't exist, 0 is returned.
- The *name* argument may contain a question mark (?). This sign is replaced by 'W' in unicode versions and by 'A' in non-unicode versions.
- In WindowsCE builds symbol is retrieved from coredll.dll library.

See also

LoadLibrary, **CallLibFunc**, **GetProcAddress**, **GetUser32Func**

References **intptr()**.

◆ GetGdi32Func()

```
intptr GetGdi32Func ( const string name )
```

Get address of function or data area exported from gdi32.dll library.

Function returns address of symbol *name* (function or data area) exported by the library. It is intended to simplify using of Windows-API functions.

Parameters

name name of exported symbol

Returns

address of symbol on success, otherwise 0

Note

- Returned function address can be used for argument of **CallLibFunc** call.
- In case of a data area returned address points to start of area.
- If Function fails, i.e. if requested symbol doesn't exist, 0 is returned.
- The *name* argument may contain a question mark (?). This sign is replaced by 'W' in unicode versions and by 'A' in non-unicode versions.
- In WindowsCE builds symbol is retrieved from coredll.dll library.

See also

LoadLibrary, **CallLibFunc**, **GetProcAddress**, **GetCoreFunc**

References **intptr()**.

◆ GetKernel32Func()

```
intptr GetKernel32Func ( const string name )
```

Get address of function or data area exported from kernel32.dll library.

Function returns address of symbol *name* (function or data area) exported by the library. It is intended to simplify using of Windows-API functions.

Parameters

name name of exported symbol

Returns

address of symbol on success, otherwise 0

Note

- Returned function address can be used for argument of **CallLibFunc** call.
- In case of a data area returned address points to start of area.
- If Function fails, i.e. if requested symbol doesn't exist, 0 is returned.
- The *name* argument may contain a question mark (?). This sign is replaced by 'W' in unicode versions and by 'A' in non-unicode versions.
- In WindowsCE builds symbol is retrieved from coredll.dll library.

See also

LoadLibrary, **CallLibFunc**, **GetProcAddress**, **GetCoreFunc**

References **intptr()**.

◆ GetLastError()

```
uint GetLastError ( )
```

Get last error code.

Function simply wraps Windows API function of same name.

Returns

Last error code of calling thread.

See also

GetSysErrorMsg

References **uint()**.

Referenced by **processExists()**.

◆ GetOle32Func()

```
intptr GetOle32Func ( const string name )
```

Get address of function or data area exported from ole32.dll library.

Function returns address of symbol *name* (function or data area) exported by the library. It is intended to simplify using of Windows-API functions.

Parameters

name name of exported symbol

Returns

address of symbol on success, otherwise 0

Note

- Returned function address can be used for argument of **CallLibFunc** call.
- In case of a data area returned address points to start of area.
- If Function fails, i.e. if requested symbol doesn't exist, 0 is returned.
- The *name* argument may contain a question mark (?). This sign is replaced by 'W' in unicode versions and by 'A' in non-unicode versions.

See also

LoadLibrary, **CallLibFunc**, **GetProcAddress**

References **intptr()**.

◆ GetShell32Func()

```
intptr GetShell32Func ( const string name )
```

Get address of function or data area exported from shell32.dll library.

Function returns address of symbol *name* (function or data area) exported by the library. It is intended to simplify using of Windows-API functions.

Parameters

name name of exported symbol

Returns

address of symbol on success, otherwise 0

Note

- Returned function address can be used for argument of **CallLibFunc** call.
- In case of a data area returned address points to start of area.
- If Function fails, i.e. if requested symbol doesn't exist, 0 is returned.
- The *name* argument may contain a question mark (?). This sign is replaced by 'W' in unicode versions and by 'A' in non-unicode versions.
- In WindowsCE builds symbol is retrieved from coredll.dll library.

See also

LoadLibrary, **CallLibFunc**, **GetProcAddress**, **GetCoreFunc**

References **intptr()**.

◆ GetSysErrorMsg() [1/2]

```
string GetSysErrorMsg ( )
```

Get readable message for system error code.

Function retrieves error message for last system error code.

Returns

Error message for last system error code, empty string if no message available.

Note

GetLastError is called internally to get error code.

See also

GetLastError

◆ GetSysErrorMsg() [2/2]

```
string GetSysErrorMsg ( uint errcode )
```

Get readable message for system error code.

Function retrieves error message for specified code.

Parameters

errcode Error code to get message for (optional)

Returns

Error message for specified code, empty string if no message available.

See also

GetLastError

References **uint()**.

◆ GetUser32Func()

```
intptr GetUser32Func ( const string name )
```

Get address of function or data area exported from user32.dll library.

Function returns address of symbol *name* (function or data area) exported by the library. It is intended to simplify using of Windows-API functions.

Parameters

name name of exported symbol

Returns

address of symbol on success, otherwise 0

Note

- Returned function address can be used for argument of **CallLibFunc** call.
- In case of a data area returned address points to start of area.
- If Function fails, i.e. if requested symbol doesn't exist, 0 is returned.
- The *name* argument may contain a question mark (?). This sign is replaced by 'W' in unicode versions and by 'A' in non-unicode versions.
- In WindowsCE builds symbol is retrieved from coredll.dll library.

See also

LoadLibrary, **CallLibFunc**, **GetProcAddress**, **GetCoreFunc**

References **intptr()**.

Referenced by **Menu::Menu()**, **Menu::AppendMenu()**, **GetDialogBaseUnits()**, **GetSysColor()**, **GetWindowLong()**, **GetWindowLongPtr()**, **Menu::InsertMenu()**, **MessageBox()**, **PostMessage()**, **SendMessage()**, **Window::setMainMenu()**, **SetParent()**, **SetWindowLong()**, and **SetWindowLongPtr()**.

B++ Reference © 2005-2026 by R.-Erik Ebert

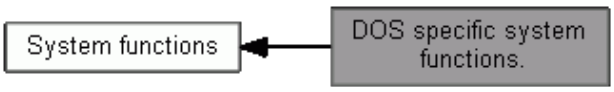
All parts of B++ are provided under terms and conditions of MIT license.

DOS specific system functions.

B++ predefined functions » System functions

All functions of this section are available in DOS builds only. More...

Collaboration diagram for DOS specific system functions.:



Functions

`null setreg (const string regname, int value)`
Set register value.
`int getreg (const string regname)`
Get register value.
`null segread ()`
Read segment registers.
`int int86 (int intnr)`
Do x86 interrupt call.
`int int86x (int intnr)`
Do extended x86 interrupt call.

Port I/O functions

`int inportb (int portid)`
Read byte from I/O port.
`null outportb (int portid, int value)`
Write byte to I/O port.
`int inport (int portid)`
Read word from I/O port.
`null outport (int portid, int value)`
Write word to I/O port.

Graphic BIOS functions.

Functions of this section allow rudimentary graphic programming using BIOS interrupt calls.

If more functions needed, they may be implemented via **int86** or **int86x** calls.

`int setscrmode (int mode)`
Set screen mode.
`null setpixel (int x, int y, int color)`
Set pixel in graphics mode.
`null line (int x1, int y1, int x2, int y2, int color)`
Draw a line.

Detailed Description

All functions of this section are available in DOS builds only.

Function Documentation

◆ getreg()

```
int getreg ( const string regname )
```

Get register value.

Function gets value of register specified by *regname*.

Parameters

regname Name of register to get value from; possible names are:
`ax, bx, cx, dx, di, si, al, ah, bl, bh, cl, ch, dl, dh, cflag, flags, ds, es, ss, cs`.

Returns

value of specified register

Note

- Register names in *regname* argument are not case sensitive, i.e. "AX" means same as "ax".
- Function reads values from internal register structure rather directly reading processor registers. Contents of this structure is modified explicitely using functions **setreg** or **segread** and implicitely by **int86** and **int86x** functions.

See also

setreg, **segread**, **int86**, **int86x**

◆ inport()

```
int inport ( int portid )
```

Read word from I/O port.

Function reads 16 bits word from I/O port specified by *portid*.

Parameters

portid id (number) of I/O port to read word from

Returns

word read

◆ inportb()

```
int inportb ( int portid )
```

Read byte from I/O port.

Function reads single byte from I/O port specified by *portid*.

Parameters

portid id (number) of I/O port to read byte from

Returns

byte read

◆ int86()

```
int int86 ( int intnr )
```

Do x86 interrupt call.

Function does a call to x86 interrupt routine specified by *intnr*. Before call is done it loads processor registers AX, BX, CX, DX, DI, SI and FLAGS with values from internal register data structure. After call it saves all registers into iternal structure and restores original values of processor registers.

Parameters

intnr interrupt number

Returns

value of AX register after interrupt call

See also

setreg, **getreg**, **int86x**

Referenced by **getActiveCp()**, and **getOEMCp()**.

◆ int86x()

```
int int86x ( int intnr )
```

Do extended x86 interrupt call.

Function does a call to x86 interrupt routine specified by *intnr*. Before call is done it loads processor registers AX, BX, CX, DX, DI, SI, DS, ES and FLAGS with values from internal register data structure. After call it saves all registers into iternal structure and restores original values of processor registers.

Parameters

intnr interrupt number

Returns

value of AX register after interrupt call

Note

Before call to this function values of processors segment registers should be transferred to internal register structure using **segread** function.

Example:

```
// interrupt call example - prints current directory path
main(;buf,bufptr)
{
   buf = newstring(200);       // buffer for result
   segread();                  // init variables for seg-registers
   setreg("ah",0x47);          // dosfunc nr.
   setreg("dl",0);             // get path for current drive
   bufptr = addr(buf);         // get address of buffer contents
   setreg("ds",bufptr >> 16);  // high word of buffer address -> segment
   setreg("si",bufptr & 0xFF); // low word of buffer address -> offset
   int86x(0x21);               // call DOS-Interrupt
   print(buf,"\n");
}
```

See also

setreg, **getreg**, **int86**, **segread**

◆ line()

```
null line ( int x1,
            int y1,
            int x2,
            int y2,
            int color )
```

Draw a line.

Function draws a graphics line from point (*x1*,*y1*) to point (*x2*,*y2*).

Parameters

x1 x-coordinate of start point

y1 y-coordinate of start point

x2 x-coordinate of end point

y2 y-coordinate of end point

color color to use for drawing the line

Returns

null

Note

- Display must be switched into a graphics mode for use this function.
- Coordinates of pixel both should be non-negative values. Maximum possible coordinate values are determined by resolution of screen (depending on selected graphics mode.
  The coordinate pair (0,0) specifies upper left corner of screen.
- Because maximum color depth of BIOS graphic modes is restricted to 125, only lowest byte of *color* argument is significant.

Example for using graphic functions:

```
// fill screen with lines
main()
{
   // Resolution is restricted to 240x64 pixels here. This allows running the
   // sample on ATARI-Portfolio too.
   var px=240;  // pixels horizontal
   var py=64;   // pixels vertical
   // switch to graphic mode
   setscrmode(4);  // should work on most display adapters
   // get current time
   var ms = timer();
   var n=0;
   // draw some lines
   for (var x=0;x<px;x+=2)
   {
      n++;
      var x2= px -x;
      line(x,0,x2,py-1,1);
```

```
    }
    for (var y=0;y<py;y+=2)
    {
       n++;
       var y2= py-1 -y;
       line(0,y,px-1,y2,2);
    }
    // get elapsed time
    ms = timer() - ms;
    var r = n*1000 / ms;
    gets();
    // switch back to text mode
    setscrmode(7);
    print(r, " lines per second\n");
}
```

See also

setscrmode, **setpixel**

Referenced by **EditCtrl::getInsertionPosition()**, and **BppVirtualMachine::notifySourceLineChanged()**.

◆ outport()

```
null outport ( int portid,
               int value )
```

Write word to I/O port.

Function writes 16 bits word (low word of *value*) to port *portid*.

Parameters

portid id (number) of I/O port to write word to

value value to write

Returns

null

◆ outportb()

```
null outportb ( int portid,
                int value )
```

Write byte to I/O port.

Function writes single byte (lowest byte of *value*) to port *portid*.

Parameters

portid id (number) of I/O port to write byte to

value value to write

Returns

null

◆ segread()

```
null segread ( )
```

Read segment registers.

Function reads current values of processors segment registers (DS, ES, SS, CS) and stores them into internal register data structure.

Returns

null

Note

Use **getreg** to access values from your program.

See also

getreg, **setreg**

Referenced by **DllEntry::loadOverlay()**.

◆ setpixel()

```
null setpixel ( int x,
                int y,
                int color )
```

Set pixel in graphics mode.

Function sets a single graphics pixel to specified *color*.

Parameters

x x-coordinate of pixel to set

y y-coordinate of pixel to set

color new color value to set for specified pixel

Returns

null

Note

- Display must be switched into a graphics mode for use this function.
- Coordinates of pixel both should be non-negative values. Maximum possible coordinate values are determined by resolution of screen (depending on selected graphics mode.
  The coordinate pair (0,0) specifies upper left corner of screen.
- Because maximum color depth of BIOS graphic modes is restricted to 125, only lowest byte of *color* argument is significant.

See also

setscrmode, **line**

◆ setreg()

```
null setreg ( const string regname,
              int          value )
```

Set register value.

Function sets value of register specified by *regname*.

Parameters

regname Name of register to set; possible names are:
`ax, bx, cx, dx, di, si, al, ah, bl, bh, cl, ch, dl, dh,`
`cflag, flags, ds, es, ss, cs.`

value New value of specified register. For 8-bit registers lowest byte and for 16-bit registers low word of *value* is used.

Returns

null

Note

- Register names in *regname* argument are not case sensitive, i.e. "AX" means same as "ax".
- Function stores values in internal register structure rather immediately modifying processor registers. Contents of this structure is applied to processor registers on calls to **int86** and **int86x** functions.

See also

getreg, **segread**, **int86**, **int86x**

◆ setscrmode()

```
int setscrmode ( int mode )
```

Set screen mode.

Function clears the screen and switches display into specified *mode*. After call cursor position is (0,0).

Parameters

mode Screen mode to switch display into. Meaning of value depends on display adaptor. Usually mode 7 specifies monochrome text mode (MDA).

Returns

display mode used by adaptor after call.

See also

setpixel, **line**

B++ Reference © 2005-2026 by R.-Erik Ebert

All parts of B++ are provided under terms and conditions of MIT license.