AttrList

COLLABORATORS			
	TITLE : AttrList		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		December 6, 2024	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

AttrList

1.1 Implementation notes

```
                The AttrList class
              ------------------------
              ($Date: 1994/08/01 15:51:51 $)
```

The AttrList class manages sets of attribute tags which are used extensively for specification of A++ objects. To the class user these attribute tag sets correspond to the well known TagItem arrays used by the Amiga® operating system.

.

```
 How to create AttrList objects
 How to read and write an AttrList
 Higher support methods
 Declaring own attribute tags

------------------------------------------------------------
 -> Back to the root menu..
```

1.2 attrlist_1

A++ classes refer to AttrLists as method parameter with an AttrList reference as formal parameter:

```
   Class::method(AttrList& attrs);
```

When calling such a method an AttrList object must be provided either from reference to an already existing or by creating a temporary object like this:

```
   object.method( AttrList(TAG_TYPE1, 16, TAG_TYPE2, 88, TAG_END) );
```

Don't forget the braces around your taglist and the 'AttrList' in front. Actually, this way a temporary AttrList object is created from the given taglist and then given by reference to the invoked method.

There are three constructors to create an AttrList object:

```
   AttrList(struct TagItem* );
   // from a 'TAG_END' terminated array

   AttrList(Tag tag1,LONG data1,...,TAG_END);
   // create from a 'TAG_END' terminated parameter list.
   // each 'tag' must match a 'data' parameter!
   // Tag values (not Tag data) must have the TAG_USER bit set!

   AttrList(const AttrList& copy);  // copy constructor
   // copy taglist of the given AttrList object (deep copy)
```

Assignment is also possible:

```
   AttrList a1(MY_TAG,23,YOUR_TAG,02,OUR_TAG,1971, TAG_END);

   AttrList a2();    // empty AttrList (no memory allocated)

   a2 = a1; // copy a1 tags to new allocated memory for a2 (deep copy)
```

1.3 attrlist_2

There are two Iterator classes provided to access an AttrList object in an easy and conform way. The AttrIterator is applicable also to const AttrLists. Always read the attribute tags by creating an AttrIterator:

```
   AttrList &attrs = ...;  // somewhere above
   ...
   AttrIterator next(attrs);
   // initialise to read from the head of the taglist

   while (next())
   {
      switch (next.tag())
      {
         case MY_TAG : date = next.data(); break;
      }
   }
```

Since the AttrIterator class also works for const AttrLists the only list access is to read the tag value ('Tag tval = next.tag()') and the tag data ('LONG dval = next.data();').

If you want to start scanning the attribute list again use 'next.reset()'.

In case you don't want to scan the whole list but find a special tag instead use:

```
   BOOL AttrIterator::findTagItem(Tag findTag);
   // if 'findTag' was in the AttrList returns TRUE and makes the TagItem
   // accessable via 'next.tag()' and 'next.data()'
```

.
The following iteration by calling 'next()' sets the iterator to the successor of the found tag.

To write to an AttrList create an 'AttrManipulator' object. Note that this is

only possible for non-const AttrLists. The AttrManipulator adds the following methods to the AttrIterator class:

```
void AttrManipulator::writeTag(Tag new);
void AttrManipulator::writeData(LONG new);
```

Example:

```
AttrManipulator next(attrs);

while (next())
{
   switch (next.tag())
   {
      case WRITE_TAG : next.writeData(0L); break;
      case DELETE_TAG : next.writeTag(TAG_IGNORE); break;
   }
}
```

1.4 attrlist_3

There are some methods which supply further powerful means to work on AttrLists as a whole.

1. BOOL addAttrs(AttrList& attrs);

The first method adds additional Tags to an AttrList: all Tags that are present in 'attrs' are added to the AttrList object together with their respective values. Those Tags that are already present in the AttrList object are not touched.
The 'attrs' list will be changed during the process!

2. BOOL updateAttrs(const AttrList& attrs);

This method copies the values of those Tags present in both the AttrList object and the 'attrs' to the AttrList. Tags in 'attrs' that are not already in the AttrList are not being considered.

3. ULONG AttrList::mapAttrs((struct TagItem *)&mapTaglist[0]);
   ULONG AttrList::mapAttrs(AttrList& attrs);

..converts attribute tags present in the mapTaglist as tag value to new tags given as corresponding tag data with deleting tags that are not within the mapTaglist. Tag values are not touched. mapAttrs() includes filterAttrs():

```
Tag filterTaglist[] = {  PGA_Top, PGA_Total, TAG_END };
ULONG AttrList::filterAttrs( &filterTaglist[0] );

AttrList attrs( PGA_Top, PGA_Total, TAG_END );
ULONG AttrList::filterAttrs( attrs );
```

..removes all tags that are not appearing in the filterTags list.
Note that filterTags is a pointer to an array of Tags, not TagItems!

Both methods return the number of tags that are still left. So, on a 0 return no further setAttributes() calls should be necessary.

1.5 attrlist_4

A++ introduces a way of making Attribute Tags type-safe. Type checking on the tag value is achieved by using special macros that shadow a simple call of a method that only receives the tag value as a specific type and returns it immediately, using function parameter type checking.
The method

```
static LONG T::confirm(T* tagValue) { return (LONG)tagValue; }
```

is usually defined inline for the class that supplies objects being used as Attribute Tag values.
Along with that goes a macro that combines the Attribute Tag with this method:

```
// the plain Attribute Tag
#define ATT_GiveMeT    (ATT_Dummy+1)

// the corresponding type-safe Attribute Tag, replacing both tag and data
#define ATT_GiveMeTObj(object) ATT_GiveMeT,T::confirm(object)
```

For conformity reasons, a type-safe Attribute Tag is named after the corresponding Tag, concatenated with an 'Obj' suffix.

Note, that for classes being derived, such a type-safe Attribute Tag is crucial, since derived classes' objects need to be cast into the requested base class.

Within the class that gets the Attribute Taglist with such a type-safe Attribute the Tag Data value obtained with 'AttrIterator::data() need to be converted into the type that was demanded from the 'confirm()' method:

```
ULONG A::setAttributes(const AttrList& attrs)
{
   AttrIterator next(attrs);
   T* t_ptr;

   while (next())
   {
      if (next.tag()==ATT_GiveMeT)
      {
         t_ptr = (T*)next.data();      // this cast is crucial!
         // if you want to check t_ptr for a T-derived class
         // use the Runtime Type Inquiry Mechanism
         T_Derived* td_ptr;
         if (NULL != (td_ptr = ptr_cast(T_Derived,t_ptr)))
            cout << "<ATT_GiveMeT> had a 'T_Derived' object!\n";
      }
   }

   ...
```

```
}
```