Document Number: X3J16/92-0005 R
WG21/N0083
Date: March 11, 1992
Project: Programming Language C++
Reply To: Uwe Steinmüller
uwe.steinmueller@sniap.mchp.sni.de

Generic Dynamic Arrays in C++

Uwe Steinmüller

Alexander von Zitzewitz

SNI AP 44

Siemens Nixdorf Informationssysteme AG 1992

1. Why are dynamic arrays needed in C++?

Arrays are a widely used low-level abstraction for storing elements of a common type. C++ provides arrays as a builtin type. So why should one want to implement a generic array class? There are basically the following reasons for doing this:

- C++ does not provide dynamic arrays. The size of an array must be known at compile time and cannot be changed at runtime. This implies a practice of using fixed sized limits. In many cases these "maximum" limits are artificial and lead to programs which are very hard to modify or extend. On the other hand using dynamic arrays in C is error prone and boring.
- C++ does not support copying and assignment of arrays.
- Generic dynamic arrays are an important building block for the implementation of many kinds of container classes, so nearly every serious application will need them.

2. Proposal for a generic dynamic array class

Since the proposed class should be used as a low-level building block it only provides basic functionality.

The name "DynArray" for this class has been chosen since "array" is a builtin type of C++, and the alternative "vector" may be expected to be a mathematical class (and would have different member functions).

```
template <class T> class DynArray {

public:
    //  Constructors
    DynArray();                 // Array of size 0
    DynArray(unsigned size); // Array of given size

    //  Destructor
    ~DynArray();                // destructs elements and frees dynamic store

    //  Copy and assign
    DynArray(const DynArray& source);
```

```
const DynArray& operator=(const DynArray& source);

//  Element access
T& operator[](unsigned index);
const T& operator[](unsigned index) const;

//  Low level access to array elements
T* base();               //  get address of first element
const T* base() const;

// append
const DynArray& operator+=(T);
const DynArray& operator+=(const DynArray& source);

//  Size
unsigned size() const;      //  returns size of array
unsigned size(unsigned sz); //  change size to sz

//  Low level helper functions
void swap(DynArray& other); //  swap contents };
```

3. Description of the public interface

3.1. Constructors

Declaration:

`DynArray()`

Synopsis:

This is the default constructor. It creates a DynArray of size null.

Result:

None

Exceptions:

None

Declaration:

`DynArray(unsigned size)`

Synopsis:

Creates an array of size 'size'. If size is null, same as DynArray() above. Otherwise enough dynamic store for 'size' elements will be allocated. It is guaranteed that the default constructor is called for each element.

Result:

None

Exceptions:

OutOfMemory

Declaration:

```
DynArray(const DynArray& source)
```

Synopsis:

This is the copy constructor. It creates a DynArray with the same size as the DynArray "source" and initializes all elements of the new DynAray with a copy of the corresponding element in DynArray source. (new[i] == source[i] for 0 <= i < source.size()).

Result:

None

Exceptions:

OutOfMemory

3.2. Destructor

Declaration:

```
~DynArray()
```

Synopsis:

Calls the destructor for each element (if there is one) and frees all dynamic memory used by the array.

Result:

None

Exceptions:

None

3.3. Assignment

Declaration:

```
const DynArray& operator=(const DynArray& source)
```

Synopsis:

This operation calls the destructor for each element of the receiver. Then frees all dynamic store used by the "this" DynArray, if necesssary, then creates a DynArray with the same size as the DynArray "source" and initializes all elements of the new DynAray with a copy of the corresponding element in DynArray source. ((*this)[i] == source[i] for 0 <= i < source.size()).

If a DynArray is assigned to itself it just returns a reference to source.

Result:

Const reference to DynArray

Exceptions:

OutOfMemory

3.4. Indexing operators

Declaration:

```
T& operator[](unsigned index)
const T& operator[](unsigned index) const
```

Synopsis:

Returns reference to object at index 'index' (index is zero based). No range checking will be performed, so that the result will be undefined, if 'index >= size()', this is because DynArray is to be a low level abstraction having low overhead. If the implementor provides a debug version of the library, an assert should check for this.

The const version will operate on constant arrays.

Result:

(Const) reference to element index of the DynArray.

Exceptions:

None

3.5. Appending

Declaration:

```
const DynArray& operator+=(T)
const DynArray& operator+=(const DynArray& source)
```

Synopsis:

Appends an object of type T (or DynArray of T's) to the Target.

Result:

Const reference to DynArray

Exceptions:

OutOfMemory

3.6. Miscellaneous

Declaration:

```
T* base()
const T* base() const;
```

Synopsis:

This operation is very problematic as a pointer to the internal representation is returned and it assumes the internal representation to be a contiguous C++ array. By this pointer the user can manipulate the internal array without using DynArrays member functions, this is quite dangerous, because there are many situations the returned pointer will become invalid. The pointer will become invalid in the following circumstances:

- The array is destructed
- The size of the array is changed
- The array is assigned to

Again there is a version for const arrays.

Returns a pointer to the first element of DynArray. Returns zero, if the size of the array is null. Here we could use a conversion operator. By using a named member function, there will be no implicit conversions and the user is responsible for using the pointer.

Result:

(Const) pointer to the base of the internal array.

Exceptions:

None

Declaration:

```
unsigned size() const
```

Synopsis:

Returns the size of the array.

Result:

Unsigned

Exceptions:

None

Declaration:

```
unsigned size(unsigned new_size)
```

Synopsis:

Changes the size of the array to 'new_size'. If 'new_size' is smaller than the current size, the destructors for the remaining elements must be called. If 'new_size' is bigger than the current size, new elements must be initialized by their default constructor. If there is not enough memory for the new size, the OutOfMemory exception will be thrown. The member function returns 'new_size'.

Result:

Unsigned

Exceptions:

OutOfMemory

Declaration:

```
void swap(DynArray& other);
```

Synopsis:

Swaps the contents of the receiver with the contents of 'other' without moving or copying elements. This is a useful low-level operation, which should be part of the standard, because it cannot be efficiently implemented without access to the private part of the class.

Result:

None

Exceptions:

None

Notes

The element type T must be a basic type or should have the following characteristics:

- A default constructor must be provided.
- Copying and assignment must be possible.

4. Class PointerDynArray

Class DynArray<void*> should be part of the standard library. For efficiency reasons there could exist a specific implementation of this class that makes use of the fact, that there are no constructors or destructors for pointers. DynArrays of pointers should be implemented by a generic subclass of DynArray<void*>. This subclass might look like this:

```
template <class T> class PointerDynArray : public DynArray<void*> {

public:

    PointerDynArray()
    {
        //  Creates array of size zero
    }

    PointerDynArray(unsigned s) : DynArray<void*>(s)
    {
        //  Creates array of size 's'. No pointer initialization.
    }

    //  Destructor generated by the compiler
    //  Copying and assignment generated by the compiler

    T*& operator[](unsigned i)
    {
        return (T*&)DynArray<void*>::operator[](i);
    }

    T*const& operator[](unsigned i) const
    {
        return (T*const&)DynArray<void*>::operator[](i);
    }

    const PointerDynArray& operator+=(T* t)
    {
        return (PointerDynArray&)DynArray<void*>::operator+=((void*)t);
    }

    const PointerDynArray& operator+=(const PointerDynArray& source)
    {
        return (PointerDynArray&)
               DynArray<void*>::operator+=((DynArray<void*>)source);
    }

    T** base()
    {
        return (T**)DynArray<void*>::base();
    }

    T*const* base() const
    {
        return (T*const*)DynArray<void*>::base();
    }

    void swap(PointerDynArray& other)
    {
        DynArray<void*>::swap(other);
    }
};
```

Since all member functions of this class are inline and basically do nothing else but casting, there will be no unnecessary duplication of code. I think that this class should also become part of the standard. If not, every user will have to implement this class (or one that looks very similar) by himself.

Acknowledgements

Thanks to Keith Gorlen, Tony Hansen and the library working group.

References

Stroustrup, Bjarne. The C++ Programming Language, 2nd Edition. Eddison Wesley 1991.