Assignment
Multi-Dimensional Typed Arrays

Knud Henriksen
Department of Computer Science
University of Copenhagen†

May 22, 2006

Contents

1 Introduction

This assignment has several sections. section 2 defines what an $N-$dimensional array is. Section 3 describes the actual assignment, and Section 4 describes how multi-dimensinal typed arrays can be implemented in different ways.

†Address: Universitetsparken 1, DK–2100 Copenhagen Ø, Denmark

2 Specification of an Array

An *array* is a finite collection of variables that all have the same name and *type*. The number of variables of the array is called the *length* of the array. All the variables have the same name, but each variable has a unique *position number* within the array. In that sense an array A of type T with length m may be thought of as representing elements of the Cartesian product $T^m \equiv \underbrace{T \times \cdots \times T}_{m}$, i.e. $A \equiv (a_1, \ldots, a_m) \in T^m$.

The individual variables of the array are referred to by specifying the name of the array and the position number within the array. This is done with an *index operator* $[\cdot] : \mathbb{N} \longrightarrow T$, which is defined on the array. The notation $A[i]; i \in \{1, \ldots, m\}$ means the i^{th} variable of the array A.

If there is defined an *assignment operator* $=: T \longrightarrow T$ on the type T, and λ is a variable of type T, the expression $\lambda = A[i]$ means that λ is assigned the value of the i^{th} variable in array A, i.e. $\lambda \equiv a_i$ after the assignment. The expression $A[i] = \lambda$ means that the i^{th} variable in array A is assigned the value of λ, i.e. $A \equiv (a_1, \ldots, a_{i-1}, \lambda, a_{i+1}, \ldots, a_m)$ after the assignment.

In addition to the index operator there is defined an *assignment operator* $=: T^m \longrightarrow T^m$ on an array. If A and B are arrays of the same *type* with lengths m and n respectively the *assignment* expression $A = B$ means that A becomes a copy of B. That is, the number of variables in A (its length) changes from m to n and they are assigned the values of the corresponding variables in array B, i.e. $\forall i \in \{1, \ldots, n\} : a_i \equiv b_i$ after the assignment.

If the type T is simple, i.e. it is *not* an *array* then the array is called *one*-dimensional. A *one*-dimensional array of type T is denoted $Array\langle T, 1\rangle$.

If the type T is itself a *one*-dimensional array of some type T_1 then the *one*-dimensional array of type T is equal to: $Array\langle T, 1\rangle \equiv Array\langle Array\langle T_1, 1\rangle, 1\rangle$. That is, the variables in $Array\langle T, 1\rangle$ are themselves *one*-dimensional arrays, so the result of applying the index operator, $Array\langle T, 1\rangle[i]$, is a *one*-dimensional array, $Array\langle T_1, 1\rangle$. Applying the index operator twice, $Array\langle T, 1\rangle[i][j]$, returns a variable of type T_1. This is because $Array\langle T, 1\rangle[i][j] \equiv Array\langle T_1, 1\rangle[j] \equiv T_1$. An array of this kind, i.e. the type T is itself a *one*-dimensional array is called a *two*-dimensional array or a *multi*-dimensional array, and it is denoted $Array\langle T_1, 2\rangle$. Conceptually, a *two*-dimensional array $Array\langle T, 2\rangle$ can be thought of as a table with two indices, one for the rows and one for the columns, i.e.

$$\begin{array}{ccccc} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \cdots & a_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & a_{m3} & \cdots & a_{mn} \end{array}$$

This can be generalised to any dimension N, so that $Array\langle T, N\rangle$ means an array where the each variable in the array is an array of type T and dimension $N - 1$. That is, to get a specific variable of type T in an $N-$dimensional array, the index operator must be applied N times to the array, i.e. $Array\langle T, N\rangle \underbrace{[i_1] \cdots [i_N]}_{N} \equiv T$.

3 The Assignment

The implementation of multi-dimentional typed arrays can be done in many ways, see sec4 This assignment is to implement multi-dimensional typed arrays using recursive templates and partially specialized templates.

This means: Implement an array of any type and any dimension as suggested in section 4.3. Error handling should be implemented, e.g. index errors, memory allocation errors etc.

Implement the template classes and test them to the point that you can convince other programmers that they work. That is, do all the member functions work?, and if there are non-member functions/operators, do they work? Show that they do.

Notice! For the suggested solution shown in section 4.3, it is the user´s responsibility that the number of parameters are N for the parameterized constructor (it cannot be checked!).

4 Analysis

This section describes four different approaches to implement *multi*-dimensional *dynamic typed* arrays, as they are descibed in section 2.

The suggested implementations vary from a very naive implementation of an array with fixed *type* and *dimension*, see section 4.1, to the more advanced implementation of a general array, with parameterized *type* and *dimension*, see section 4.3.

The naive approach is good for a person who writes a program every now and then, and do not think of reusing the code.

The more advanced approach is good for a person who writes many programs, and wants to reuse the code. The implementation of the more advanced approach is general, and can be included in a library of general data structures.

The section is divided into four subsections. The first subsection describes two naive implementations where both the *dimension* `N` and the type `Type` are part of the class name.

The first subsection describes a naive implementation where both the *dimension* `N` and the type `Type` are part of the class name. This approach results in an implementation which is different for each dimension and each type. For the first approach the syntax for accessing the individual entries in the array is different from the usual index operator `[·]`. To index an entry the function operator `(·)` is used. The second approach uses the usual index operator `[·]` for accessing the entries of the array.

Subsection two describes two refinements of the naive approaches explained in subsection one. In these approaches only the *dimension* `N` is part of the class name and the classes are implemented as templates with the type `Type` as parameter. These approaches result in implementations which are different for each dimension. For the first refinement the function operator `(·)` is used for accessing the individual entries of the array. The second refinement uses the usual index operator `[·]` for accessing the entries of the array.

Subsection three describes the way *multi*-dimensional *dynamic typed* arrays are implemented in the library. In this approach an `N`-dimensional array of some general type

`Type` is implemented as a recursive template class with both the `Type` and the *dimension* `N` as parameters. In order to make this work it is necessary to implement a partial specialization of the general template class for dimension `N == 1`.

The last subsection contains a comparison of the four suggested implementations.

4.1 Dimension and Type in Name — `ArrayND_Type`

This section describes two naive but very different approaches of how to implement *multi*-dimensional *dynamic typed* arrays. A common feature for both approaches is that both the *dimension* and the *type* are part of the class name.

The first subsection describes an implementation which is self-contained and does not rely on the implementation of other classes. That approach is cumbersome to implement and it has some unwanted restrictions, e.g. it is *not* possible to extract a row (a *one*-dimensional array) from a *two*-dimensional array.

The second subsection describes an implementation which relies on the implementation of other classes, but it much more easy to implement (all the classes are almost identical), and it has some nice properties which can be useful in some applications, e.g. it *is* possible to extract a row (a *one*-dimensional array) from a *two*-dimensional array.

4.1.1 First Implementation of `ArrayND_Type`

This subsection describes a naive approach to implement a *multi*-dimensional *dynamic typed* array. For each *dimension* `N` and each type `Type` a new class with the systematic name `ArrayND_Type` must be implemented. In the systematic class name, `N` is the actual dimension and `Type` is the name of the actual type.

This approach is restricted in the way that it does *not* provide the ability to extract a row (a *one*-dimensional array) from a *two*-dimensional array.

Conceptually, a 3-dimensional array of size $2 \times 4 \times 3$ can be thought of as 2 layers of 2-dimensional arrays of size 4×3, i.e.

$$\begin{matrix} d_{000} & d_{001} & d_{002} \\ d_{010} & d_{011} & d_{012} \\ d_{020} & d_{021} & d_{022} \\ d_{030} & d_{031} & d_{032} \end{matrix} \qquad\qquad \begin{matrix} d_{100} & d_{101} & d_{102} \\ d_{110} & d_{111} & d_{112} \\ d_{120} & d_{121} & d_{122} \\ d_{130} & d_{131} & d_{132} \end{matrix}$$

Such structures can be stored in linear memory by first storing all the rows of layer 1 followed by the rows of layer 2, as shown in figure 1.

Offset	0	1	2	3	4	5	6	7	8	9	10	11	12		24
$data* \longrightarrow$	d_{000}	d_{001}	d_{002}	d_{010}	d_{011}	d_{012}	d_{020}	d_{021}	d_{022}	d_{030}	d_{031}	d_{032}	d_{100}	• • •	d_{132}

Figure 1: Memory layout for a 3-dimensional array of size $2 \times 4 \times 3$.

Generally, if a 3-dimensional array has the size $l \times m \times n$ and is indexed by the indices i, j, k all starting from 0, the (i, j, k)th entry is stored at memory location

$$A(i,j,k) \longrightarrow data + mni + mj + k \tag{1}$$

where the expression $mni + mj + k$ is the offset from the start, i.e. the memory location which `data*` points to. If the indices all start from 1, the (i, j, k)th entry is stored at memory location

$$A(i, j, k) \longrightarrow data + mn(i-1) + m(j-1) + (k-1) \tag{2}$$

This storing scheme can be generalized to any dimension `N`.

A sketch of an implementation of a 3-dimensional array of type `Type` with indices starting at 1 is shown below. In this implementation the entries of the array is stored in memory as shown in figure 1, and the indices start at 1, that is equation (2) is used to access the entries of the array.

```
typedef unsigned int uint

class Array3D_Type {
public:
    Array3D_Type(uint Length1, uint Length2, uint Length3)
        : length1(Length1), length2(Length2), length3(Length3),
          data(new Type[Length1 * Length2 * Length3])
    {}

    ~Array3D_Type() { delete[] data; }

    Type& operator()(uint Index1, uint Index2, uint Index3)
    {
        return data[length2*length3*(Index1-1) + length2*(Index2-1) + (Index3-1)];
    }

    Type const& operator()(uint Index1, uint Index2, uint Index3)
    {
        return data[length2*length3*(Index1-1) + length2*(Index2-1) + (Index3-1)];
    }

    // ...
private:
    uint  length1, length2, length3;
    Type* data;
};
```

Assume the class shown above has been implemented for `N`≡`3` and `Type`≡`int` yielding a class named `Array3D_int`. An example of how to use a 3-dimensional array of type `int` is shown below.

```
void Usage()
{
    Array3D_int C(5, 3, 7);
    C(3, 2, 5) = 9;
}
```

Advantages of the Implementation

+ To implement an array of dimension `N` and type `Type` only *one* new class must be implemented.
+ The constructor has a fixed number of parameters.

Disadvantages of the Implementation

– The syntax for indexing an array is done with the function operator `(·)` instead of the usual index operator `[·]`.
– Both the *dimension* `N` and the type `Type` are 'hard-coded' into the class, and also into its name, e.g. to implement a *eight*-dimensional array of type `double` a new class `Array8d_double` must be implemented.
– Polution of the namespace because each array must have a unique name including both the *dimension* `N` and the type `Type`.
– Each class has a different implementation.

4.1.2 Second Implementation of `ArrayND_Type`

This subsection describes a refinement of the naive implementation of *multi*-dimensional *dynamic typed* arrays described in section 4.1.1. The refinement consists of making the implementation of all arrays similar, but still have both the *dimension* `N` and the *type* `Type` as part of the class name.

Conceptually, a `N`-dimensional array can be thought of as a *one*-dimensional array where each entry is a `(N-1)`-dimensional array. This is shown in figure 2 for a 3-dimensional array of size $l \times m \times n$.

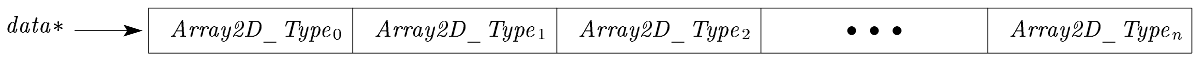

Figure 2: Memory layout for a 3-dimensional array of size $l \times m \times n$ when a 3-dimensional array is implemented as a one-dimensional array of 2-dimensional arrays.

The similarity of the classes is obtained by letting the index operator `[·]` of an `N`-dimensional array return an `(N-1)`-dimensional array. The price of making all the array classes similar is that to implement an `N`-dimensional array, all the lower-dimensional arrays must also be implemented.

That is, for each *dimension* `N` and each type `Type`, `N` new classes with the systematic names `ArrayND_Type`,...,`Array1D_Type` must be implemented. In the systematic class names, `N` is the actual dimension and `Type` is the name of the actual type.

The use of the index operator `[·]` instead of the function operator `(·)` is a cosmetic choice, because it is more pleasant to view the expression `A[l][m][n]` instead of `A(l)(m)(n)`. Also, the expression `A[l][m][n]` is the usual expression for indexing an ordinary `C++` 3-dimensional array.

This approach offers the ability to extract a row (a *one*-dimensional array) from a *two*-dimensional array.

A sketch of an implementation of a 3-dimensional array of type `Type` with indices starting at 1 is shown below. In this implementation the entries of the array are stored in memory as shown in figure 2, and the indices start at 1.

```
class Array3D_Type {
public:
    Array3D_Type() : length(0), data(0) {}

    Array3D_Type(uint Length1, uint Length2, uint Length3)
    {
        Initialize(Length1, Length2, Length3);
    }

    ~Array3D_Type() { delete[] data; }

    void Initialize(uint Length1, uint Length2, uint Length3)
    {
        length = Length1;
        data   = new Array2D_Type[length];
        for (uint i = 0; i < length; ++i) data[i].Initialize(Length2, Length3);
    }

    Array2D_Type& operator[](uint Index) { return data[Index-1]; }
    Array2D_Type const& operator[](uint Index) { return data[Index-1]; }

    // ...
private:
    uint length;
    Array2D_Type* data;
};
```

From the sketch it can be seen that this implementation is very different from the implementation shown in section 4.1.1. The main differences are that in this implementation the class `Array3D_Type` has a default constructor and a member function `Initialize(uint, uint, uint)`.

The default constructor is needed because an `N`-dimensional array allocates memory for a *one*-dimensional array of `(N-1)`-dimensional arrays with the operator `new`, i.e. in the example it is the the line `data = new Array2D_Type[length];`. In the language `C++` it is not possible to specify arguments to the individual entries of an ordinary `C++` array of objects, so the default constructor is always used, see [Stroustrup, 1997, page 250].

The member function `Initialize(`$\texttt{uint}_1$`,...,`$\texttt{uint}_N$`)` first allocates memory for its ordinary *one*-dimensional `C++` array of `(N-1)`-dimensional arrays, i.e. the type `Array(N-1)D_Type`. For this allocation the default constructor of `Array(N-1)D_Type` is used, so all these arrays are empty, i.e. have length zero.

Next, it calls the member function `Array(N-1)D_Type::Initialize(`$\texttt{uint}_1$`,...,`$\texttt{uint}_{(N-1)}$`)` of each entry in its ordinary *one*-dimensional `C++` array of `(N-1)`-dimensional arrays.

In the example implementation shown above, the member function `Initialize(uint, uint, uint)` calls the function `Array2D_Type::Initialize(uint, uint)` for each entry in its ordinary *one*-dimensional `C++` array of `Array2D_Type` arrays.

The member function `Initialize(uint, uint, uint)` should not be available to the user, but in the sketch it is just a public member in order to not confuse things. It can be hidden for the user, but it not the issue here.

The lower dimensional arrays `Array(N-1)D_Type`; $N \geq 2$ are implemented analogously, see the example below. Notice, the implementation of an `(N-1)`-dimensional array is almost a textual copy of the implementation of an `N`-dimensional array. The only difference is the case `N == 1` which must be implemented differently, see the example below.
An example implementation of a 2-dimensional array is shown below.

```
class Array2D_Type {
public:
    Array2D_Type() : length(0), data(0) {}

    Array2D_Type(uint Length1, uint Length2) { Initialize(Length1, Length2); }

    ~Array2D_Type() { delete[] data; }

    void Initialize(uint Length1, uint Length2)
    {
        length = Length1;
        data   = new Array1D_Type[length];
        for (uint i = 0; i < length; ++i) data[i].Initialize(Length2);
    }

    Array1D_Type& operator[](uint Index) { return data[Index-1]; }
    Array1D_Type const& operator[](uint Index) { return data[Index-1]; }

    // ...
private:
    uint length;
    Array1D_Type* data;
};
```

From the sketch of the implementation, it can be seen that an `N`-dimensional array and an `(N-1)`-dimensional array are implemented analogously, as long as `N >= 2`. Also, it can be seen that the index operator is much more simple than the one shown in section 4.1.1.

The only difference between an `N`-dimensional array and an `(N-1)`-dimensional array, `N >= 2`, is the the number of parameters of to the parameterized constructor, and the number of parameters of to the member function `Initialize(uint`$_1$`,...,uint`$_m$`)`.

A sketch of an implementation of a *one*-dimensional array is shown below. It is almost a copy of the implementaion of the `N`-dimensional array shown above, except that the index operator `[·]` returns a value of type `Type` instead of an `ArrayND_Type`.

```
class Array1D_Type {
```

```
public:
    Array1D_Type() : length(0), data(0) {}

    Array1D_Type(uint Length) { Initialize(Length); }

    ~Array1D_Type() { delete[] data; }

    void Initialize(uint Length)
    {
        length = Length;
        data   = new Type[length];
    }

    Type& operator[](uint Index) { return data[Index-1]; }
    Type const& operator[](uint Index) { return data[Index-1]; }

    // ...
private:
    uint length;
    Type* data;
};
```

Assume the classes shown above have been implemented for `N=3,2,1` and `Type`≡`int` yielding a classes named `Array3D_int`, `Array2D_int`, `Array1D_int`. An example of how to use a 3-dimensional array of type `int` is shown below.

```
void Usage()
{
    Array3D_int C(5, 3, 7);
    C[3][2][5] = 9;
}
```

Advantages of the Implementation

+ The syntax for indexing an array is the usual index operator `[·]`.
+ The implementation of all array classes is similar for each *type* and *dimension*, so the implementation of a new array class can be done by 'copy-and-paste'.
+ The constructor has a fixed number of parameters.

Disadvantages of the Implementation

– Both the *dimension* and the `Type` are 'hard-coded' into the array type and also its name, e.g. to implement a *eight*-dimensional array of type `double` 8 new classes `Array1D_double,...,Array8D_double` must be implemented for each `Type`.
– Polution of the namespace because each array must have a unique name including both the *dimension* `N` and the type `Type`.

4.2 Dimension in Name — `ArrayND<Type>`

This section describes refinements to the naive approaches in sections 4.1.1 and 4.1.2 of how to implement *multi*-dimensional *dynamic typed* arrays. A common feature for both refinements is that *only* the *dimension* is part of the class name, and the classes are implemented as templates with the *type* as parameter.

The first subsection describes an implementation which is self-contained and does not rely on the implementation of other classes. That approach is cumbersome to implement and it has some unwanted restrictions, e.g. it is *not* possible to extract a row (a *one*-dimensional array) from a *two*-dimensional array.

The second subsection describes an implementation which relies on the implementation of other classes, but it much more easy to implement (all the classes are almost identical), and it has some nice properties which can be useful in some applications, e.g. it *is* possible to extract a row (a *one*-dimensional array) from a *two*-dimensional array.

4.2.1 First Implementation of `ArrayND<Type>`

This section describes a refinement of the naive approach in section 4.1.1 of how to implement a *multi*-dimensional *dynamic typed* array. The refinement is that the array is implemented as a template class with the type `Type` as parameter. That is, for each *dimension* a new template class with the systematic name `ArrayND` must be implemented. In the systematic class name, `N` is the actual dimension.

A sketch of an implementation of a 3-dimensional array of any type `Type` is shown below. In this implementation the entries of the array is stored in memory as shown in figure 1, and the indices start at 1. That is, equation (2) is used to access the entries of the array.

```
template <typename Type>
class Array3D {
public:
    Array3D(uint Length1, uint Length2, uint Length3)
        : length1(Length1), length2(Length2), length3(Length3),
          data(new Type[Length1 * Length2 * Length3])
    {}

    ~Array3D() { delete[] data; }

    Type& operator()(uint Index1, uint Index2, uint Index3)
    {
        return data[length2*length3*(Index1-1) + length2*(Index2-1) + (Index3-1)];
    }

    Type const& operator()(uint Index1, uint Index2, uint Index3)
    {
        return data[length2*length3*(Index1-1) + length2*(Index2-1) + (Index3-1)];
    }
```

```
    // ...
private:
    uint  length1, length2, length3;
    Type* data;
};
```

Assume the class shown above has been implemented for N≡3 and Type≡int yielding a class named `Array3D_int`. An example of how to use a 3-dimensional array of type `int` is shown below.

```
void Usage()
{
    Array3D<int> C(5, 3, 7);
    C(3, 2, 5) = 9;
}
```

Advantages of the Implementation

+ To implement an array of dimension `N` of *any* type `Type` only *one* new class must be implemented.
+ The constructor has a fixed number of parameters.

Disadvantages of the Implementation

– The syntax for indexing an array is done with the function operator `(·)` instead of the usual index operator `[·]`.
– The *dimension*is 'hard-coded' into the array type and also its name, e.g. to implement a *eight*-dimensional array of some type a new template class `Array8D<Type>` must be implemented.
– Polution of the namespace because each array must have a unique name including the *dimension* `N`.
– Each class has a different implementation.

4.2.2 Second Implementation of `ArrayND<Type>`

This section describes a refinement of the implementation of *multi*-dimensional *dynamic typed* arrays as described in section 4.1.2. The refinement consists of implementing the array as a template class with the type `Type` as parameter. That is, for each *dimension* `N`, `N` new classes with the systematic names `Array1D,...,ArrayND` must be implemented. In the systematic class names `N` is the actual dimension.

A sketch of an implementation of a 3-dimensional array of any type `Type` with indices starting at 1 is shown below. In this implementation the entries of the array are stored in memory as shown in figure 2, and the indices start at 1.

```
template <typename Type>
class Array3D {
public:
    Array3D() : length(0), data(0) {}
```

```
    Array3D(uint Length1, uint Length2, uint Length3)
    {
        Initialize(Length1, Length2, Length3);
    }

    ~Array3D() { delete[] data; }

    void Initialize(uint Length1, uint Length2, uint Length3)
    {
        length = Length1;
        data   = new Array2D<Type>[length];
        for (uint i = 0; i < length; ++i) data[i].Initialize(Length2, Length3);
    }

    Array2D<Type>& operator[](uint Index) { return data[Index-1]; }
    Array2D<Type> const& operator[](uint Index) { return data[Index-1]; }

    // ...
private:
    uint length;
    Array2D<Type>* data;
};
```

The lower-dimensional, i.e. the *two*- and *one*-dimensional arrays must also be implemented.

A sketch of an implementation of a *two*-dimensional array is shown below, and it can be seen that it almost identical to the implementation of the *three*-dimensional array shown above.

```
template <typename Type>
class Array2D {
public:
    Array2D() : length(0), data(0) {}

    Array2D(uint Length1, uint Length2) { Initialize(Length1, Length2); }

    ~Array2D() { delete[] data; }

    void Initialize(uint Length1, uint Length2)
    {
        length = Length1;
        data   = new Array1D<Type>[length];
        for (uint i = 0; i < length; ++i) data[i].Initialize(Length2);
    }

    Array1D<Type>& operator[](uint Index) { return data[Index-1]; }
    Array1D<Type> const& operator[](uint Index) { return data[Index-1]; }
```

```
    // ...
private:
    uint length;
    Array1D<Type>* data;
};
```

Finally, a sketch of a implementation of a *one*-dimensional array is shown below.

```
template <typename Type>
class Array1D {
public:
    Array1D() : length(0), data(0) {}

    Array1D(uint Length) { Initialize(Length); }

    ~Array1D() { delete[] data; }

    void Initialize(uint Length)
    {
        length = Length;
        data   = new Type[length];
    }

    Type& operator[](uint Index) { return data[Index-1]; }
    Type const& operator[](uint Index) { return data[Index-1]; }

    // ...
private:
    uint  length;
    Type* data;
};
```

Assume the classes shown above have been implemented for `N=3,2,1` and `Type`≡`int` yielding a classes named `Array3D<int>`, `Array2D<int>`, `Array1D<int>`. An example of how to use a 3-dimensional array of type `int` is shown below.

```
void Usage()
{
    Array3D<int> C(5, 3, 7);
    C[3][2][5] = 9;
}
```

Advantages of the Implementation

+ The syntax for indexing an array is the usual index operator `[·]`.
+ The implementation of all array classes is similar for each *dimension*, so the implementation of a new array class can be done by ‘copy-and-paste’.

+ The constructor has a fixed number of parameters.

Disadvantages of the Implementation

– The *dimension*is 'hard-coded' into the array type, e.g. to implement a *eight*-dimensional array of some type `Type`, 8 new template classes `Array1D<Type>,...,Array8D<Type>` must be implemented.

– Polution of the namespace because each array must have a unique name including the *dimension* `N`.

4.3 Recursive Templates — `Array<Type,N>`

This section describes how *multi*-dimensional *dynamic typed* arrays can be implemented using recursive templates.

The implementation is a further refinement of the implementation described in section 4.2.2 The arrays are implemented as a recursive template class with both the type `Type` and the dimension `N` as parameters. That is, an array of *any* type `Type` and *any* dimension `N` can be implemented by implementing only *two* template classes: a general template class for `N > 1`, and a partial specialization for `N == 1`. The partial specialization for `N == 1` is needed in order to stop the recursion.

The implementation also has the nice property that it is possible to extract an `(N-1)`-dimensional array from an `N`-dimensional array using the index oparator `[·]`.

The only disadvantage of the suggested implementation is that one of the constructors must have a variable number of parameters. This disadvantage can be removed by the use of expression templates, see [Marcok, 1998], if one is willing to accept an ugly syntax for specifying the parameters to the constructor of the array. A sketch of how the actual implementation can be done is shown below.

The primary template class for dimension `N > 1` and *any* type `Type` might be implemented as sketched below

```
#include <cstdarg>

typedef unsigned int uint;

template <typename Type, uint N>
class Array {
public:
    Array() : length(0), data(0)
    {
        // Implement the default constructor
    }

    Array(uint Length ...)
    {
        uint lengths[N];
        lengths[0] = Length;
```

```
        va_list paramlist;
        va_start(paramlist, Length);
            for (uint i = 1; i < N; ++i) lengths[i] = va_arg(paramlist, uint);
        va_end(paramlist);
        Initialize(lengths);
    }

    virtual ~Array()
    {
        // Implement the destructor
    }

    void Initialize(uint* lengths)
    {
        // Initialize the Array
    }

    Array<Type,N-1>& operator[](uint Index)
    {
        // Implement the index operator
    }
    Array<Type,N-1> const& operator[](uint Index)
    {
        // Implement the index operator
    }

    // ...
private:
    uint             length;
    Array<Type,N-1>* data;
};
```

The partial specialization for `N == 1` of the general template is shown below.

```
template <typename Type>
class Array<Type,1> {
public:
    Array() : length(0), data(0)
    {
        // Implement the default constructor
    }

    Array(uint Length)
    {
        uint lengths[1];
        lengths[0] = Length;
        Initialize(lengths);
```

```
    }

    virtual ~Array()
    {
        // Implement the destructor
    }

    void Initialize(uint* lengths)
    {
        // Initialize the Array
    }

    Type& operator[](uint Index)
    {
        // Implement the index operator
    }
    Type const& operator[](uint Index)
    {
        // Implement the index operator
    }

    // ...
private:
    uint  length;
    Type* data;
};
```

Assume the primary template class and the partial specialization have been implemented. Then, multidimensional arrays of any dimension and any type can be instantiated by just specifying the template parameters. An example of how to use a 3-dimensional array of type `int` is shown below.

```
void Usage()
{
    Array<int,3> C(5, 3, 7);
    C[3][2][5] = 9;
}
```

Advantages of the Implementation

+ The syntax for indexing an array is the usual index operator `[·]`.
+ The implementation of all array classes, i.e. for any *type* and any *dimension*, is done by implementing only *two* templete classes: a general `N`-dimensional array, $N > 1$, of a generel `Type`, and a partial specialization of that class for `N == 1`.
+ There is no pollution of the namespace, because there is no need to invent new names for each type and dimension.

Disadvantages of the Implementation

– It is more difficult to implement the general template class for the array.
– The constructor has a variable number of parameters.

4.4 Comparison of the Implementations

Four different implementations of *multi*-dimensional *dynamic typed* arrays have been described in sections 4.1.1 — 4.3. All of the proposed implementations have different requirements to the user – some are wanted, and others are not. A comparison of the different implementations is shown in table 1.

References

[Marcok, 1998] Zlatko Marcok. Dynamic arrays with expression templates. *C/C++ Users Journal*, pages 37—42, July 1998.

[Stroustrup, 1997] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 3rd edition, 1997.

Comparison of the Implementations

Dimension and Type in Name — ArrayND_Type

Method	Declaration	Indexing	Implementation	Comments
First Try	ArrayND_Type A($l_1, \ldots, l_N$);	A($i_1, \ldots, i_N$)	For each *type* and each *dimension* the user *must* implement *one* new class.	The implementation is *different* for all classes.
Second Try	ArrayND_Type A($l_1, \ldots, l_N$);	A[i_1]$\cdots$[i_N]	For each *type* and each *dimension* the user *must* implement N new classes.	The implementation is *almost* identical for all N classes.

Dimension in Name — ArrayND<Type>

Method	Declaration	Indexing	Implementation	Comments
First Try	ArrayND<Type> A($l_1, \ldots, l_N$);	A($i_1, \ldots, i_N$)	For each *dimension* the user *must* implement *one* new template class.	The implementation is *different* for all classes.
Second Try	ArrayND<Type> A($l_1, \ldots, l_N$);	A[i_1]$\cdots$[i_N]	For each *dimension* the user *must* implement N new template classes.	The implementation is *almost* identical for all N classes.

Recursive Templates — Array<Type,N>

Method	Declaration	Indexing	Implementation	Comments
Final Try	Array<Type,N> A($l_1, \ldots, l_N$);	A[i_1]$\cdots$A[i_N]	Only *two* classes need to be implemented to handle *all* types and *all* dimensions.	The general class has a constructor with a variable number of parameters.

Table 1: Comparison between the different implementations of *multi*-dimensional *dynamic typed* arrays as described in sections 4.1—4.3.