Contents

0.1 Arrays

An *array* is an indexed collection of variables, all of the same type. Almost every programming language offers arrays and arrays are probably the most widely used data structure. The strong points of arrays are:

- They are easy to use.
- They are efficient and powerful. The access to an arbitrary element of an array takes constant time.
- They are compact and have essentially no overhead in space. The space required is the space for the elements in the array plus a small number of additional storage cells (two for the implementation given in the next section).

We assume here that the elements of an array are indexed starting at zero and call the number of variables in an array the size of the array. It is customary to use subscript notation for the variables in an array. Thus, if A is an array of size sz, the variables in A are denoted $A[0]$, $A[1]$, ... , $A[sz-1]$. Figure 1 shows the complete definition of an array data type.

We discuss the implementation of three array types. Static arrays (arrays, whose size is known at compile time) and dynamic arrays (arrays, whose size is known at the time of their declaration), are the subject of Section 0.1.1 and unbounded arrays (arrays, whose size can grow during execution) are the subject of Section 0.1.2. The analysis of unbounded arrays will introduce the technique of amortization. Section **??** discusses some fine points.

0.1.1 Static and Dynamic Arrays

We discuss static arrays and dynamic arrays. Unbounded arrays whose size can grow during the execution are discussed in the next section.

1. Definition

An instance A of the parameterized data type *array*<T> is an indexed collection of variables of type T. The number of variables in the collection is called the size of the array. The variables are indexed from 0 to $size - 1$.

2. Creation

array<T> $A(int\ n)$;	creates an instance A of type *array*<T> with index set $[0..n-1]$.

3. Operations

$T\&$ $A[int\ i]$	returns the i-th variable in the collection. *Precondition*: $0 \leq i < size$.
int A.size( )	returns the size of A.
void A.init(T x)	assigns x to $A[i]$ for every $i \in [0 .. size - 1]$.

4. Implementation

The efficiency of our implementation of arrays is stated in Theorem 1.

Figure 1: The specification of an array data type.

```
array<int> A(5);   // static  array, size is known at compile time
array<int> A(n);   // dynamic array, size depends on the value of n
```

The size of a static array is known at compile time. Static arrays are particularly easy to implement. One simply allocates the appropriate number of variables on the stack. In order to access $A[i]$ we compute its address in a register and then load the element. For the following piece of code, we assume that the starting address in register *R2*, that the index is stored in *R1*, and that the objects of type T fit into single memory cells, and that we want to load the contents of $A[i]$ into register *R4*.

```
R3 = R2 + R1;
R4 = S[R3];
```

Arrays whose size becomes only known at the time of declaration are slightly harder to implement. The compiler allocates a storage location of type *pointer to T* and an integer holding the size n of the array on the stack. The

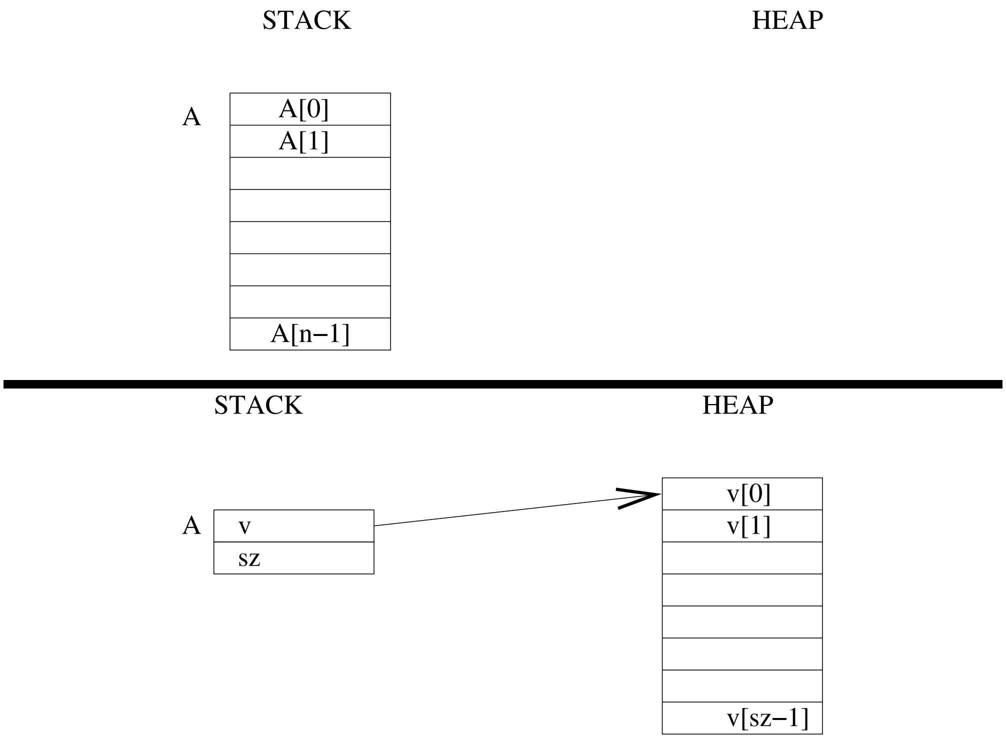

Figure 2: The top part shows the storage allocation for an array whose size is known at compile time, the bottom part shows the storage allocation for an array whose size is only known at the time of declaration.

space for the n variables of type T is allocated on the heap when the declaration of the array is processed. Figure 2 illustrates the difference. The code to access $A[i]$ is slightly more complicated now. We assume again that the value of i is stored in register *R1* and the address of A is stored in register *R2*. Then the following sequence of instructions will load the contents of $A[i]$ into register *R4*.

```
R3 = S[R2];     // address of A[0]
R1 = R3 + R1;   // address of A[i]
R4 = S[R1];
```

The indirection of going through a pointer requires an additional statement. Observe however, that the first statement is independent of the array index and hence can be optimized away, if a register can be reserved to hold the address of $A[0]$. In either case the access to an array element takes constant time. Initializing all elements of an array to a particular value takes time proportional to the size of the array. The time for the construction of an array is the time to allocate the memory for the array plus the time to initialize the array. Memory allocation time is constant when the array is allocated on the stack and depends on the memory manager otherwise. Programming languages have different conventions about initialization. In C++, the convention is to initialize all elements of the array by the default constructor of type T and hence initialization takes time proportional to the size of the array. Built-in types have no default constructor and the initial values are unspecified and may depend on the execution history. Some compilers initialize to the null value of the built-in type. In either case initialization takes at most linear time. The time to destruct an array is no more than the declaration time.

Theorem 1 *Declaration of an array of size n takes time $O(n)$ if n is known at compile time and takes memory allocation time $T_{alloc}(n)$ plus time $O(n)$ otherwise. Initialization takes linear time and array access takes constant time. The time for destruction is bounded by the time for construction.*

We close this section by giving the complete C++-implementation of the array type of Figure 1. An array has two private variables, a pointer to T, which contains the location of the array on the heap, and an integer sz, which contains the size of the array. The constructor stores the desired size of the array in sz and then allocates space for the appropriate number of variables on the heap, the destructor frees the space on the heap, the access operator first checks whether the index i is in the index range of the array (the C-macro *assert*(*condition*) raises an error if *condition* evaluates to *false*) and then returns $v[i]$, *size* returns the size of the array, and $init(x)$ sets all elements of the array to x.

```
#include <assert.h>
template<class T>
class array  {
  T* v;
  int sz;
public:
  array(int n) { sz = n; v = new T[n]; }
  ~array()     { delete[] v;  }
  T& operator[](int i)
               { assert (i >= 0 && i < sz );
                 return v[i];
               }
  int size() const { return sz;}
  void init(const T& x)
               { for (int i = 0; i < sz; i++) v[i] = x; }
};
```

Why did we implement our own array data type and not simply use the built-in array type of C++? We have two reasons:

- Our arrays are safer and more flexible than the built-in arrays of C++. Our arrays offer an index-out-of-bounds check and our arrays know their size.

- We were able to illustrate the separation between *abstract data type* and *implementation of a data type* in a simple setting. The specification in Figure 1 defines an array data type which can be used without (!!) knowing its implementation. Theorem 1 gives information about the efficiency of the data type. The implementation given at the end of the section is a detailed proof that the stated functionality and efficiency can be obtained. At this point, many readers will shrug their shoulders. What's the deal? Specification and implementation are so similar, that we would have saved a lot of ink, had we not distiguished between the two. But wait, read the specification given in Figure 3. You can probably not guess how the type is implemented. But nevertheless you can immediately start to use it. That's what abstraction is about.

We want you to acquire the habit of clearly separating between specification and implementation. The implementation proves that the functionality and efficiency stated in the specification can indeed be achieved. A user of a data type has no need to know the implementation. The specification suffices, if done properly.

0.1.2 Unbounded Arrays

Sometimes, we would like to have the convenience and efficiency of arrays without having to specify the size of the array. For example, we may want to read data from an input file of unknown size and store the data in an array. With dynamic arrays we have to make two passes over the data. In the first pass we determine the number of elements in the file and in the second pass we actually read the data into the array. This is inconvenient. In this section we will introduce *unbounded arrays* which give the illusion of an array with index set $[0 .. \infty]$. The access time will still be constant and the space requirement will be proportional to the maximal index ever used. Figure 3 shows the specification of an unbounded array data type. The standard template library of C++ provides unbounded arrays under the name *vector*.

The interface of unbounded arrays is similar to the interface of arrays. In the constructor the default element is specified, but there is no need to specify the size of the array. The access operator is split into a read access for which we use the subscript operator and a write access for which we use the function name *put*. In Section 0.1.3 we will discuss why we make this distinction. Observe that the subscript operator now returns an object of type T and no longer a variable of type T. Thus the subscript operator can no longer be used to write to an array.

The idea behind the implementation of unbounded arrays is simple. Let n be the current size of our unbounded array A. Then we have a standard dynamic array, call it v, of size at least n. We use vsz to denote the size of v. The first n locations contain the elements of A and the remaining elements are a reservoir for accesses beyond the current size of the array. The elements of v beyond n contain $tdef$. When the array is created, we start with $vsz = 1$; any other constant would also work. In order to achieve space efficiency, we maintain the invariant that the size of v is at most twice the size of A.

How do we realize the access operations? If the index i is less than vsz we simply access $v[i]$. We also update the size of A, if necessary, i.e., set the size of A to the maximum of $i + 1$ and its current value. If i is larger than vsz, we need to work slightly harder.

1. Definition

An instance A of the parameterized data type *uarray*<T> is an indexed unbounded collection of variables of type T. The variables are indexed from 0 to ∞. The initial value of each variable is *tdef*, an element of T which is specified when the array is created. One plus the largest index every used in an access operation (read or write access) is called the size of the array. The size is zero initially.

2. Creation

uarray<T> $A(T\ \mathit{def})$;	creates an instance A of type *uarray*<T> and default element *def*.

3. Operations

int A.size( )	returns the size of A.
T $A[\mathit{int}\ i]$	returns the contents of the i-th variable in the collection. *Precondition*: $0 \le i$.
void A.put(*int* i, T e)	stores e in the i-th variable of the collection. *Precondition*: $0 \le i$.

4. Implementation

The efficiency of our implementation of unbounded arrays is stated in Theorem 3.

Figure 3: The specification of an unbounded array data type.

We allocate a new dynamic array *vnew* of size $2vsz$ and move the contents of all elements of v to the new array. We then delete v and replace v by *vnew*. We also set *vsz* to $2vsz$. If v is large enough now ($i < vsz$), we perform the access. Otherwise, we repeat and double *vsz* again. The doubling strategy guarantees that $vsz \le 2i$ after the reallocation, i.e., the space used for v is at most the size of A.

The detailed code is given at the end of the section.

The Amortized Analysis of Array Access: How long does an array reference take? The time for array accesses varies widely. If $i < vsz$, the access takes constant time. If $i \ge vsz$, *vsz* is doubled until $i < vsz$ and a new array of this size is allocated. The time for the access depends on i and may be arbitrarily large. We will next show that we can *amortize* the cost of the expensive accesses over the cheap accesses and obtain small overall access time. We will first give the argument and then make some general remarks about amortization.

Theorem 2 *n calls of* `operator[]` *or* `put` *for an* `uarray` *take time at most $O(n + sz)$, where sz is the final size of the array.*

Proof: We start with three simple observations:

- The time for an array access is $O(1)$ plus the time needed for reallocating v. If v is not reallocated, the time is $O(1)$.
- $sz - 1$ is the largest index used in an access operation.
- Whenever vsz is changed, it is doubled. Let 2^k be the final value of vsz. When vsz was doubled for the last time, it was because the access with index $sz - 1$ did not fit into the array of size 2^{k-1}, i.e., $2^{k-1} \leq sz - 1$ or $2^{k-1} < sz$. Thus $2^k \leq 2sz$.

During the life-time of the *uarray*, we allocate dynamic arrays of size 2^0, 2^1, ... , 2^k. The cost of allocating an array of size 2^i is

$$O(2^i + T_{alloc}(2^i)) = O(2^i)$$

by our assumption about the time to allocate a chunk of size 2^i on the heap (see Section **??**).

The total time for reallocating v is therefore bounded by

$$O(\sum_{i=0}^{k} 2^i) = O(2^k) = O(sz) \ .$$

∎

Theorem 2 is most interesting if $sz = O(n)$, i.e., the final size of the array is bounded by some constant times the number of array accesses. In other words, a fraction of the array elements is actually accessed. If this assumption is not satisfied, one should use hashing data structures instead arrays. We discuss hashing in Section **??**.

So assume that $sz = O(n)$. Under this assumption, Theorem 2 tells us that total execution time divided by the number of operations is constant. We say that the execution time "per operation" is constant or that the *amortized execution time* of an access to an unbounded array is constant. Amortization is an extremely powerful technique and we will see it frequently in this book. A general discussion of amortization can be found in Section Amortization of the Foundations Chapter (TODO: take as basis Kurt's contribution to Ottman's book).

The meaning of amortization is about the same as in common live, but the construction above avoids a pitfall of common life. "I am going to cycle to work every day from now on and hence it is justified to buy a luxury bike. The cost per ride is going to be small". Does this kind of reasoning sound familiar to you? The bicylce is bought, it rains, and all good intentions are gone. The bike has not amortized. *Uarrays* are more careful than this. They do not start with a huge array (= a luxury bike), but with an tiny array (= a second-hand bike), and move to larger arrays only when justified (= after sufficiently many trips).

A Detailed Implementation: We give a complete C++ implementation of `uarrays`.

⟨*uarray.h*⟩≡

```
#ifndef UARRAY
#define UARRAY

#include <assert.h>

template<class T>
class uarray {
  int vsz;  // allocated memory, size of v
  T *v;     // the elements to be stored
  int n;    // 1 + largest index ever used
  T tdef;   // default value
  void make_room(int i);
public:
  ⟨uarray body⟩
};
⟨make room⟩
#endif
```

When an `uarray` is constructed we just allocate a small array for a start. Any size would do; we start with the minimum possible, an array of size one. Starting with a slightly larger array would avoid some of the early reallocations. Destruction of an unbounded arrays amount to destroy the underlying dynamic array and *size* simply returns to the size.

⟨*uarray body*⟩≡

```
uarray(T def)
{
  vsz = 1;
  v = new T[vsz];
  n = 0;
  tdef = def;
  v[0] = def;
}
~uarray()  { delete [] v; }
int size() { return n; }
```

We come to array access. Suppose we want to read or write the i-th element of our array. If i is a valid index into v, i.e., $i < vsz$, we update the size of our uarray and access the i-element of v. If $i \geq vsz$, we reallocate v by means of the call *make_room*(i). This function is discussed below.

$\langle$*uarray body*$\rangle$+$\equiv$

```
T operator[](int i) {
  if ( i >= vsz ) make_room(i);
  if ( i >= n)    n = i + 1;
  assert(i >= 0);
  return v[i];
}
void put(int i, const T& e) {
  if ( i >=  vsz ) make_room(i);
  if ( i >= n    ) n = i + 1;
  assert(i >= 0);
  v[i] = e;
}
```

We finally come to the function *make_room*. It is called with an index i that is not valid for v. We double the size of v until $i < vsz$. We then reallocate v, move the contents of the first n elements of v to the new version of v and initialize the remaining elements to *tdef*.

$\langle$*make room*$\rangle$$\equiv$

```
template<class T>
void uarray<T>::make_room(int i)
{ do { vsz *= 2; } while ( i >= vsz );
  T* vnew = new T[vsz];
  assert(vnew != 0);        // fires if new failed to allocate memory
  for (int j = 0;  j < n;  j++)   vnew[j] = v[j];
  for (int j = n;  j < vsz;  j++) vnew[j] = tdef;
  delete [] v;
  v = vnew;
}
```

We summarize the discussion in

Theorem 3 *Declaration of an uarray takes constant time. The total cost of n access operations into an array of final size sz is $O(n + sz)$. This is constant amortized time if $sz = O(n)$. Operation size takes constant time and the cost of destruction is accounted for by the other operations.*

0.1.3 Fine Points

We discuss three topics: (1) the lack of persistence of addresses in *uarrays*, the role of exponential growth in the amortization argument for *uarrays*, and real-time *uarrays*.

Persistence of Addresses of Variables: The subscript operator of *arrays* (not *uarrays*) returns a variable of type T and hence we can use the subscript operator on both sides of the assignment operator. We can also take the address of an array element, store it in a pointer, and later assign to the address stored in the pointer.

```
A[3] = A[4] + 1;
T* p = &A[3];
(*p) = 5;
```

Why did we give unbounded arrays a different interface? The reason is that there is a crucial difference between the variables in an array and the variables in an unbounded array. The addresses of variables in an array do not change over time. It is therefore safe to store addresses of variables in arrays.

This is different for unbounded arrays. The variables in unbounded arrays may move around. Their addresses may change as the effect of a reallocation of the underlying dynamic array. It is therefore unsafe to establish a pointer to an variable in an unbounded array. Our interface of unbounded arrays makes it impossible to do so.

Of course, our interface to unbounded arrays is somewhat less elegant than the interface to arrays.

```
A[3] = A[4] + 1;        // elegant and easy to read
A.put(3,A[4] + 1);      // inelegant and harder to read
```

How about the following solution? We let the array operator return a variable of type T as in the case of arrays, i.e.,

```
T&  operator[](int i) {
      if (i >= n) make_room(i);
      assert(i >= 0);
      return v[i];
}
```

and warn the reader that taking the address of a variable in an unbounded array is unsafe. The syntax of the language does not exclude it, but we could include a warning in the specification of unbounded arrays. Something like

WARNING: It is unsafe to establish a pointer to a variable in an unbounded array.

The warning will not do. The problem is deeper and occurs for any data structure which moves user data around internally. On the systems of the authors, the program

```
uarray<int> A(0);                                   // default value 0
for (i = 0;  i < 20;  i++) A[i] = 1 + A[i+1];
```

produces the array $\langle 0,0,1,0,1,1,1,0,1,1,1,1,1,1,1,0,1,1,1,1\rangle$ when the modified definition of the access operator is used, although it should produce all ones. Why?

In the assignment $A[i] = 1 + A[i+1]$, there are two occurrences of the subscript operator. Assume that the first subscript operator is evaluated first[1].

[1] C++, like many other programming languages do not specify in which order the two sides of an assignment are evaluated, since leafing the order of evaluation open makes it easier for a compiler to generate efficient code, and since the additional burden on the programmer is minor.

We obtain a reference to $A[i]$ (apparently, the compiler did not read our warning). Next $A[i+1]$ is accessed. This may reallocate v and hence invalidate the reference to $A[i]$. The assignment will go to the old location of $A[i]$ and hence the new location will stay at its initial value.

We split array access into reading access and writing access and hence the assignment above has to be written $A.put(i, 1 + A[i+1])$. In function calls, the arguments are evaluated before the body of the function is executed and hence the reallocation of v takes place before the address of $v[i]$ is computed. Thus the assignment is to the new location of the variable $A[i]$ and the code is correct.

In LEDA, the error just discussed occurred in maps and hashing arrays. It was extremely difficult to locate and to repair. Exercises **??** and **??** discusses a solution for *urrays*. The solution allows to use the subscript operator on both sides of an assignment operator.

Exponential Growth and Amortized Analysis: In the implementation of *uarrays*, we doubled the size of the internal array v, whenever it became to small. The doubling strategy lead to constant amortized access time. Which other strategies would work? Assume that we start with an array of size 1 and end with an array of size sz. Let $s_0 = 1$, $s_1, \ldots, s_m$ be the sequence of sizes used for v. Then the total cost of reallocation is $O(\sum_i s_i)$. We need $\sum_i s_i = O(s_m)$ and $s_m = O(sz)$. Both conditions hold, if we use an exponentially increasing sequence of sizes, like $s_i = 2^i$ or $s_i = 3^i$. If we use a slower growing sequence, say $s_i = i^2$, only the second condition would hold, but not the first. Observe that $\sum_{0 \le i \le m} i^2 = \Theta(i^3)$. If we use a faster growing sequence, say $s_i = 2^{2^i}$, only the first condition would hold, but not the second. Observe, that $s_{i+1} = s_i^2$ and hence we would only have $s_m = O(sz^2)$.

Real Time Unbounded Arrays: Array access takes constant amortized time (if $sz = O(n)$). However, individual access times in *uarrays* vary widely. This is unacceptable in real time applications, where worst case and not just amortized time bounds must be guaranteed.

We next show how to guarantee constant worst case access times in unbounded arrays under the assumption that every index in an array access is at most one larger than the maximal previously used index, i.e., that is it not allowed to jump over indices. Formally, read and write access have the precondition $0 \le i \le A.size(\,)$.

In order to make unbounded arrays real time, we have to change the reallocation process. As before, we allocate internal arrays whose sizes are powers of two. Assume that we have just allocated an array of size 2^{k+1}. The old array had size 2^k and hence the allocation was caused by an access to index $i = 2^k$. This holds since all smaller indices are legal for the old array and we are not allowed to jump over indices. The crucial observation is now that there will be no further reallocation for the next 2^k accesses to the array, since indices 2^k, $2^k + 1, \ldots, 2^{k+1} - 1$ must be used before the next reallocation is needed. We exploit this observation by initializing *vnew* and by moving the elements from

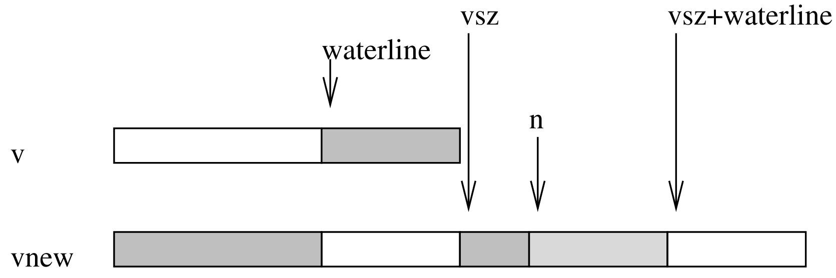

Figure 4: The representation of an unbounded array: The darkly shaded areas of v and *vnew* comprise the variables of the array, the lightly shaded area of *vnew* is already initialized, and the white area of *vnew* is not initialized yet. The white area of v is not in use anymore. The *waterline* moves one unit to the right with every array access.

v to *vnew* in a piecemeal fashion. Each of the next 2^k access operations moves one element from v to *vnew* and initializes one element of *vnew*. In this way we will guarantee that all elements are moved to *nvew* and that *vnew* is completely initialized by the time the next reallocation is due.

The details are as follows. At any point of time we have arrays v and *vnew* of sizes $vsz = 2^k$ and 2^{k+1} respectively. As before, we maintain n which is one plus the largest index used. We have $n \le 2^{k+1}$. In addition, we maintain a variable *waterline* with the following properties (see Figure 4):

- $n \le vsz + waterline$.
- the variable $A[i]$ is realized by $vnew[i]$ for $0 \le i < waterline$ and $vsz \le i < vsz + n$ and is realized by $v[i]$ otherwise, i.e., for $waterline \le i < vsz$.
- the variables $vnew[i]$ with $vsz + n \le i < vsz + waterline$ are initialized to *tdef*.

Assume that the properties above hold. The following steps realize the access to $A[i]$:

Increment the waterline: If *waterline* has not reached *vsz* yet, we move one further element from v to *vnew*, initialize one further element of *vnew* and increase *waterline*. This maintains the second and the third property and makes the first property true with n replaced by $n + 1$.

Update the size of A: If i is negative or larger than n, the size of A, the access is illegal. So assume $0 \le i \le n$. If i is equal to n, we increase n by one. Observe this extends A by a variable $A[i]$; the third condition implies that this variable has value *tdef* and hence is properly initialized. We conclude that all three properties hold again. In addition, we have $i < n$.

Access proper: For this step we distiguish cases. If $n \le 2^{k+1} = 2vsz$, we simply access $A[i]$ as indicated by the second item.

So assume $n = 2vsz + 1$ and hence $i = 2vsz$. The first condition guarantees that $waterline > vsz$ and hence the second and third properties guarantee that all variables of v are represented by $vnew$. We throw away v, double vsz, move $vnew$ to v, and allocate a new version of $vnew$. We also copy $v[0]$ to $vnew[0]$ and initialize $vnew[vsz]$ to $tdef$. This guarantees that our three properties hold for $waterline = 1$.

We next give the detailed code. We first define a function *access* that returns a reference to the variable $A[i]$ and then realize operator [] and function *put* in terms of it.

⟨*array access operators*⟩≡

```
private:
T& access(int i)
{ if ( waterline < vsz )
  { vnew[waterline] = v[waterline];
    vnew[vsz + waterline] = tdef;
    waterline++;
  }
  assert( i >= 0 && i <= n );        // illegal to jump over an index
  if ( i == n)  n = i + 1;
  if ( n > 2 * vsz )
  { vsz = 2*vsz;
    delete [] v;
    v = vnew;
    vnew = new T[2*vsz];             // unitialized
    vnew[0] = v[0];
    vnew[vsz] = tdef;
    waterline = 1;
  }
  if ( i < waterline || i >= vsz ) return vnew[i];
  else return v[i];
}
public:
T operator[](int i)         { return access(i); }
void put(int i, const T& e) { access(i) = e; }
```

There is one remark in order here. The implementation above is not real-time. The statement `vnew = new T[2*vsz]` will not only allocate the space for the new version of *vnew*, but will also initialize it, and hence it will require time $\Theta(vsz)$. Can somebody point out the C++ construct that I have to use.

The remainder of the implementation is simple.

⟨*real_time_uarray.h*⟩≡

```
#ifndef REAL_TIME_UARRAY
#define REAL_TIME_UARRAY

#include <assert.h>
```

```
template<class T>
class real_time_uarray {
  int vsz;       // allocated memory, size of v
  T* v;          // the old array
  T* vnew;       // the new array
  int n;         // 1 + largest index ever used
  int waterline; // the waterline
  T tdef;        // default value

public:
  ⟨construction, destruction, and size⟩
  ⟨array access operators⟩
};
#endif
```

To construct a `real_time_uarray`, we simply set *vsz* to one, and n and *waterline* to zero. We also define v and *vnew* as arrays of size 1 and 2, respectively, and initialize $v[0]$ to *tdef*. Observe that the first access must be to $A[0]$ and hence the first array access will initialize *vnew* properly.

⟨*construction, destruction, and size*⟩≡

```
real_time_uarray(T def)
{
  vsz = 1;
  v = new T[vsz];
  vnew = new T[2*vsz];
  n = waterline = 0;
  tdef = def;
  v[0] = def;
}
~real_time_uarray()
{ delete [] v;
  delete [] vnew;
}

int size() { return n; }
```

Theorem 4 *Creation of a real time unbounded array takes constant time. Array access takes memory allocation time in the worst case and takes constant time in the amortized sense. Size takes constant time and the time for destruction is accounted for by the other operations.*

Exercises:

1. Implement a class *gene_array* which has the same semantics as LEDA's *array* class but stores genome information more compactly. Genome information consists of strings over an alphabet of size four. To store a string of length n, your class should consume only $n/4 + O(1)$ bytes.

2. Analyze the amortized cost of an array access if the reallocation of a dynamic array would provide just enough space to accommodate the newly accessed element.

3. Would it be a good idea to make the function `make_room` an `inline`-function? Explain.

4. Implement a richer set of operations on `uarrays` similar to the `vector` class in the standard library, e.g., space reservation, shrinking, iterators.

5. (a) Explain what is going wrong in our incorrect try to get rid of the clumsy function `put`. Hint: Find out what the `C` standard has to say about the order in which the indexing operators are applied.

   (b) Implement a safe implementation without changing the representation of an `uarray`. Hint: Define a template class `uarray_element_ref<T>` which stores a pointer to an `uarray<T>` and an index. Modify `operator[]` so that it returns an `uarray_element_ref`. Now "hide" this modification by defining `uarray_element_ref<T>::operator=(T value)` and `T uarray_element_ref<T>::operator T()` appropriately.[2]

6. We have learned that revealing references to reallocated data is unsafe. So we can solve a the problem by implementing `uarray` without reallocation.

   (a) Build an implementation where v stores only pointers to the elements which are themselves allocated on the heap and never moved.

   (b) Try to estimate how large should `sizeof(T)` be so that the space overhead incurred by our solution is at most a factor of two? Test your guess experimentally.

   (c) *Propose a more space efficient implementation. Hint: Store the elements in small arrays of fixed size (preferably a power of two). v now stores only the pointers to the start of the small arrays. To implement `operator[]`, use the most significant bits of the index to find the small array and the least significant bits to find the element in the small array.

7. Discuss the pros and cons of the two approaches for implementing safe references to elements of dynamic arrays. Give situations where either one is faster. How much does the relative quality of the two alternatives depend on the quality of the compiler?

8. *The unbounded array data structures we considered so far are not well suited for the case that $n \gg k$, i.e., the largest key ever used is much larger than the actual number of operations performed. Repair this by designing an unbounded array data structure where k Operations take

[2] This exercise teaches more about `C++` programming than about algorithmics. Similar tricks are known as *lazy evaluation* and "hidden" objects like `uarray_element_refs` are often called *proxys* or *surrogates* [?].

time $\mathcal{O}(k)$ even if n, the largest index ever used, is much larger than k. (Hint: Use two (unbounded) arrays. One of size $\mathcal{O}(n)$ which may contain uninitialized values and one of size $\mathcal{O}(\min(k, n))$ which stores new valid entries in the order they arrive. Exploit that memory allocation works in time independent of the allocated size if we do not insist on initializing the allocated segment.