Chapter 5

Basic Data Types

5.1 One Dimensional Arrays (array)

1. Definition

An instance A of the parameterized data type *array*<E> is a mapping from an interval $I = [a..b]$ of integers, the index set of A, to the set of variables of data type E, the element type of A. $A(i)$ is called the element at position i. The array access operator ($A[i]$) checks its precondition ($a \leq i \leq b$). The check can be turned off by compiling with the flag `-DLEDA_CHECKING_OFF`.

#include < *LEDA/array.h* >

2. Types

array<E>:: *item* the item type.

array<E>:: *value_type* the value type.

3. Creation

array<E> A(*int* a, *int* b);
creates an instance A of type *array*<E> with index set $[a..b]$.

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

array<E> A; creates an instance A of type *array*<E> with empty index set.

Special Constructors

array<E> A(*int* *low*, *const* E& x, *const* E& y);
creates an instance A of type *array*<E> with index set $[low, low+1]$ initialized to $[x, y]$.

array<*E*> *A*(*int low*, *const E*& *x*, *const E*& *y*, *const E*& *w*);

creates an instance A of type *array*<*E*> with index set $[low, low + 2]$ initialized to $[x, y, w]$.

array<*E*> *A*(*int low*, *const E*& *x*, *const E*& *y*, *const E*& *z*, *const E*& *w*);

creates an instance A of type *array*<*E*> with index set $[low, low + 3]$ initialized to $[x, y, z, w]$.

4. Operations

Basic Operations

E&	*A*[*int x*]	returns $A(x)$. *Precondition*: $a \leq x \leq b$.
E&	*A*.get(*int x*)	returns $A(x)$. *Precondition*: $a \leq x \leq b$.
void	*A*.set(*int x*, *const E*& *e*)	sets $A(x) = e$. *Precondition*: $a \leq x \leq b$.
void	*A*.copy(*int x*, *const array*<*E*>& *B*, *int y*)	sets $A(x) = B(y)$. *Precondition*: $a \leq x \leq b$ and $B.low() \leq y \leq B.high()$.
void	*A*.copy(*int x*, *int y*)	sets $A(x) = A(y)$. *Precondition*: $a \leq x \leq b$ and $low() \leq y \leq high()$.
void	*A*.resize(*int a*, *int b*)	sets the index set of A to $[a..b]$ such that for all $i \in [a..b]$ which are not contained in the old index set $A(i)$ is set to the default value of type E.
void	*A*.resize(*int n*)	same as $A.resize(0, n-1)$.
int	*A*.low( )	returns the minimal index a of A.
int	*A*.high( )	returns the maximal index b of A.
int	*A*.size( )	returns the size $(b - a + 1)$ of A.
void	*A*.init(*const E*& *x*)	assigns x to $A[i]$ for every $i \in \{\ a \ldots b\ \}$.
bool	*A*.C_style( )	returns *true* if the array has "C-style", i.e., the index set is $[0..size - 1]$.
void	*A*.swap(*int i*, *int j*)	swaps the values of $A[i]$ and $A[j]$.

void *A*.permute( ) the elements of A are randomly permuted.

void *A*.permute(*int* l, *int* h) the elements of A[l..h] are randomly permuted.

Sorting and Searching

void *A*.sort(*int* (∗*cmp*)(*const* *E*& , *const* *E*&))

sorts the elements of A, using function *cmp* to compare two elements, i.e., if $(in_a, \ldots, in_b)$ and $(out_a, \ldots, out_b)$ denote the values of the variables $(A(a), \ldots, A(b))$ before and after the call of sort, then $cmp(out_i, out_j) \leq 0$ for $i \leq j$ and there is a permutation π of $[a..b]$ such that $out_i = in_{\pi(i)}$ for $a \leq i \leq b$.

void *A*.sort( ) sorts the elements of A according to the linear order of the element type E. *Precondition*: A linear order on E must have been defined by $compare(constE\&, constE\&)$ if E is a user-defined type (see Section 1.3)..

void *A*.sort(*int* (∗*cmp*)(*const* *E*& , *const* *E*&), *int* l, *int* h)

sorts sub-array A[l..h] using compare function *cmp*.

void *A*.sort(*int* l, *int* h) sorts sub-array A[l..h] using the linear order on E. If E is a user-defined type, you have to define the linear order by providing the compare function (see Section 1.3).

int *A*.unique( ) removes duplicates from A by copying the unique elements of A to $A[A.low()], ..., A[h]$ and returns h ($A.low( ) - 1$ if A is empty). *Precondition*: A is sorted increasingly according to the default ordering of type E. If E is a user-defined type, you have to define the linear order by providing the compare function (see Section 1.3).

int *A*.binary_search(*int* (∗*cmp*)(*const* *E*& , *const* *E*&), *const* *E*& x)

performs a binary search for x. Returns an i with $A[i] = x$ if x in A, A.low() $- 1$ otherwise. Function *cmp* is used to compare two elements.
Precondition: A must be sorted according to *cmp*.

int *A*.binary_search(*const* *E*& x)

as above but uses the default linear order on E. If E is a user-defined type, you have to define the linear order by providing the compare function (see Section 1.3).

int *A*.binary_locate(*int* (∗*cmp*)(*const E*& , *const E*&), *const E*& *x*)

Returns the maximal i with $A[i] \leq x$ or *A.low*( )-1 if $x < A[low]$. Function *cmp* is used to compare elements. *Precondition*: *A* must be sorted according to *cmp*.

int *A*.binary_locate(*const E*& *x*)

as above but uses the default linear order on E. If E is a user-defined type, you have to define the linear order by providing the compare function (see Section 1.3).

Input and Output

void *A*.read(*istream*& *I*) reads $b-a+1$ objects of type E from the input stream I into the array A using the *operator* $\gg$ (*istream*&, *E*&).

void *A*.read( ) calls *A*.read(*cin*) to read *A* from the standard input stream *cin*.

void *A*.read(*string s*) As above, uses string *s* as prompt.

void *A*.print(*ostream*& *O*, *char space* = ' ')

prints the contents of array *A* to the output stream *O* using *operator* $\ll$ (*ostream*&, *const E*&) to print each element. The elements are separated by character *space*.

void *A*.print(*char space* = ' ') calls *A*.print(*cout*, *space*) to print *A* on the standard output stream *cout*.

void *A*.print(*string s*, *char space* = ' ')

As above, uses string *s* as header.

ostream& *ostream*& *out* $\ll$ *const array*<*E*>& *A*

same as *A.print*(*out*); returns *out*.

istream& *istream*& *in* $\gg$ *array*<*E*>& *A*

same as *A.read*(*in*)); returns *in*.

Iteration

STL compatible iterators are provided when compiled with `-DLEDA_STL_ITERATORS` (see *LEDAROOT*/*demo*/*stl*/*array.c* for an example).

5. Implementation

Arrays are implemented by C++vectors. The access operation takes time $O(1)$, the sorting is realized by quicksort (time $O(n \log n)$) and the binary_search operation takes time $O(\log n)$, where $n = b - a + 1$. The space requirement is $O(n * sizeof(E))$.