Chapter 11

Deques – Double Ended Data Structures

Chapter Overview

In this chapter we will not only introduce the deque data structure from the standard template library, but also use the data type to illustrate two important general search techniques, depth and breadth first search. Next, we will once again revisit the topic of inheritance, introduced in Chapter 9, showing how inheritance can be used to construct *frameworks*, which are skeleton applications used as the basis for solving similar problems. Finally, we conclude the chapter by presenting a simplified deque implementation, similar to the standard library data structure.

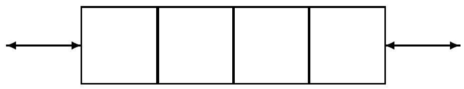
- The deque abstraction
- Depth and Breadth first searching
- Frameworks
- A Simplified Implementation

11.1 The Deque abstraction

The deque data type (pronounced either "deck" or "DQ") is one of the most interesting data structures in the standard template library. Of all the STL containers, the deque is the least conventional. It represents a data type that is seldom considered to be one of the "classic" data abstractions, as are vectors, lists, sets or trees. Nevertheless, the deque is a powerful and versatile abstraction.

The operations provided by the deque data type, shown in Figure 11.1, are a combination of those provided by the classes vector and list. Like a vector, the deque is a randomly accessible structure. This means that instances of the class deque can be used in most situations in which a vector might be employed. Like a list, elements can be inserted into the middle of a deque, although such insertions are not as efficient as they are with a list.

The term deque is short for *Double-Ended QUEue*, and describes the structure well. The deque is a combination of stack and queue, allowing elements to be inserted at either end. Whereas a vector only allows efficient insertion at one end, the deque can perform insertion in constant time at either the front or the end of the container. Like a vector, a deque is a very space efficient structure, using far less memory for a given size collection than will, for example, a list. However, again like a vector, insertions into the middle of the structure are permitted, but are not efficient. An insertion into a deque may require the movement of every element in the collection, and is thus $O(n)$ worst case.

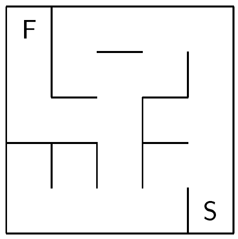


One of the most common uses for a deque is as an underlying container for either a stack or a queue. The deque is a preferable container for such employment if the size of the collection remains relatively stable during the course of execution, while if the size varies widely a list or vector is preferable. In many cases the decision concerning which structure is most appropriate can only be made by performing direct measurement of program size or execution time.

Because the meaning of the operations on a deque are similar to those of a vector or a list we will not describe them in detail. Instead, we will proceed to an example program that makes use of the features provided by a deque.

11.2 Application–Depth and Breadth First Search

In this section we will examine a program that will discover a path through a maze, such as the one shown below. We will assume that the starting point for the search is always in the lower right corner of the maze, and the goal is the upper left corner.

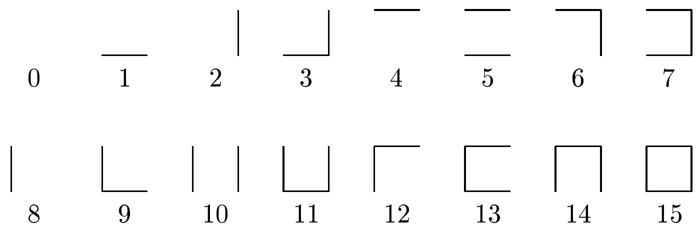


Constructors and Assignment	
deque<T> d;	default constructor
deque<T> d (anInt);	construct with initial size
deque<T> d (anInt, a_T_value);	construct with initial size and initial value
deque<T> d (aDeque);	copy constructor
d = aDeque;	assignment of deque from another deque
d.swap (aDeque);	swap contents with another deque
Element Access and Insertion	
`d[i]`	subscript access, can be assignment target
`d.front ()`	first value in collection
`d.back ()`	final value in collection
`d.insert (iterator, value)`	insert value before iterator
`d.push_front (value)`	insert value at front of container
`d.push_back (value)`	insert value at back of container
Removal	
`d.pop_front ()`	remove element from front of vector
`d.pop_back ()`	remove element from back of vector
`d.erase (iterator)`	remove single element
`d.erase (iterator,iterator)`	remove range of elements
Size	
`d.size ()`	number of elements currently held
`d.empty ()`	true if vector is empty
Iterators	
deque<T>::iterator itr	declare a new iterator
d.begin ()	starting iterator
d.end ()	stopping iterator
d.rbegin ()	starting iterator for reverse access
d.rend ()	stopping iterator for reverse access

Table 11.1: Summary of deque operations

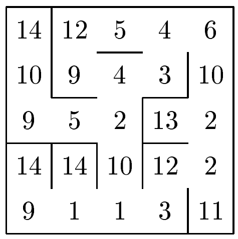
Our purpose in presenting this example is not only to contrast two different types of search techniques, but also to demonstrate the operations of the deque data type, and finally to show how a deque can be used either in a stack-like or queue-like fashion. These two broad approaches to searching are known as *depth first search* and *breadth first search*.

We want the maze searching program to be general, able to solve any two dimensional maze and not simply the example maze shown above. We therefore design a scheme so that the description of the maze can be read from an input file. Different files can be used to test the program on a variety of different mazes. To see how to do this, note that a maze can be described as a sequence of squares, or *cells*. The example maze shown above, for example, is a five by five square of 25 cells. Each cell can be characterized by a number, which describes the surrounding walls. Sixteen numbers are sufficient. In this fashion we have the following vocabulary for describing cells:



The pattern of the numeric values becomes apparent if one considers the number not as a decimal value, but as a binary pattern. The 1's position indicates the presence or absence of a south wall, the 2's position the east wall, the 4's position the north wall, and the 8's position the west wall. A value such as 13 is written in binary as 1101. This indicates there are walls to the north, west, and south, but not the east.

Using this scheme, the example maze could be described by 25 integer values. In the following we have superimposed these values on the maze, to better illustrate their relationship to the original structure.



This *external* representation of the maze must be mapped on to an *internal* representation. The internal representation need not match the external representation, as long as there is a means of conversion between the two. The internal representation will again be a sequence of cells. Each cell is an instance of the class cell. Instances of class cell maintain three data fields:

- A number. This is an integer value used to identify the cell. Cells are numbered consecutively from left to right and top to bottom.
- A list of neighboring cells. Each cell will have an entry in this list for all other neighbor cells that can be reached.
- A Boolean value, named visited, that will be used to mark a cell once it has been visited. Traversing a maze often results in dead-ends, and the need to back up and start again. Marking visited cells avoids repeating effort and potentially walking around in circles.

A class description for cell is shown below. The member function addNeighbor simply inserts a value into the list of neighbors. The member function visit will encode the searching algorithm. We will defer a description of this until after we have outlined the rest of the program.

```
class cell {
public:
        // constructor
    cell (int n) : number(n), visited(false) { }

        // operations
    void addNeighbor (cell * n) { neighbors.push_back(n); }
    void visit (deque<cell *> &);

protected:
    int number;
    bool visited;
    list <cell *> neighbors;
};
```

The class that represents the entire maze structure is called maze, and has the following structure:

```
class maze {
public:
    maze (istream &);
    void solveMaze ();

protected:
    cell * start;
    bool finished;
    deque <cell *> path;
};
```

The class maze maintains three data fields. The first is a pointer to the starting cell. The second is a Boolean flag that is set to true once the goal cell has been reached. The third data field is a deque, used to hold the path (or paths) currently being traversed.

The constructor for the class maze, shown in Figure 11.1, reads the maze description from an input file (passed as argument), converting from the external representation to the internal representation. The first two integer values in the file represent the number of rows and columns of the maze. In order to set the links properly, a vector is maintained that represents the cells in the row *previous* to the row currently being read from the input file. (This is the row to the immediate north of the current). Recall that the two argument form used in the constructor for this vector initializes each entry to the second argument value, in this case a null pointer value. After each new cell has been processed, the entry in this vector for the corresponding column has changed. The following picture illustrates the use of this vector. Here the first two rows have been processed, and the first two columns in the third row. The boxed elements indicate the current value of the vector. Note that the value of the vector with the same column number as the present element is the neighbor to the north, while the value with the index one smaller than the current column is the neighbor to the east.

$$\begin{array}{ccccc} 14 & 12 & 5 & 4 & 6 \\ 10 & 9 & \boxed{4} & \boxed{3} & \boxed{10} \\ \boxed{9} & \boxed{5} & 2 & & \end{array}$$

As each new value is read, it is determined whether or not it has a link to the north (to the previous row) or to the west (to the most recently processed cell). If so, then links are established. Notice that links cannot be created to the east and south, since these cells have not yet been created. However, note that a link to the south or east corresponds to a link from the north or west in the adjoining cells, so by making both sets of connections at once it is only necessary to recognize north and west connections. After the cell has been completely processed, it is assigned to the corresponding position in the vector, and the next element is read.

$$\begin{array}{ccccc} 14 & 12 & 5 & 4 & 6 \\ 10 & 9 & 4 & \boxed{3} & \boxed{10} \\ \boxed{9} & \boxed{5} & \boxed{2} & 13 & \end{array}$$

This process continues until all the maze description values have been read. Having entered the maze data, we can now describe the algorithm used to solve the maze. The fundamental problem occurs when there is a choice of several directions to pursue. In our example maze, this occurs immediately after the first step, when there are possible moves

```
maze::maze (istream & infile)
    // initialize maze by reading from file
{
    int numRows, numColumns;
    int counter = 1;
    cell * current = 0;

        // read number of rows and columns
    infile >> numRows >> numColumns;

        // create vector for previous row
    vector <cell *> previousRow (numRows, 0);

        // now read data values
    for (int i = 0; i < numRows; i++)
        for (int j = 0; j < numColumns; j++) {
            current = new cell(counter++);
            int walls;
            infile >> walls;
                // make north connections
            if ((i > 0) && ((walls & 0x04) == 0)) {
                current->addNeighbor (previousRow[j]);
                previousRow[j]->addNeighbor (current);
                }
                // make west connections
            if ((j > 0) && ((walls & 0x08) == 0)) {
                current->addNeighbor (previousRow[j-1]);
                previousRow[j-1]->addNeighbor (current);
                }
            previousRow[j] = current;
            }
        // most recently created cell is start of maze
    start = current;
    finished = false;
}
```

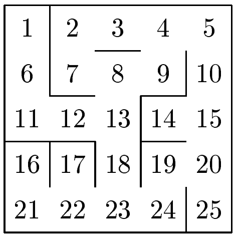
Figure 11.1: Constructor for class maze.

both north and west. One or the other paths must be selected. However, since a selection may ultimately be wrong (resulting in a dead end), it is important to keep track of the alternative possibilities. We do so using a deque. At each step the deque will hold pointers to cells that are known to be reachable, but have not yet been visited.

Describing the first few steps in the process will clarify the approach. There is only one cell reachable from the starting position, and thus initially the deque contains only one element. The following also repeats the maze, with the cells showing their given number.

the deque

front	20	back



This value is pulled from the deque, and the neighbors of the cell inserted back into the deque. This time there are two neighbors, so the deque will have two entries:

front	19	15	back

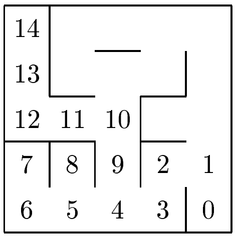
Only one value can be explored at any time. So the first element is removed from the deque, and its neighbors inserted, and so on repeatedly. Two steps later we again have a choice, and both neighbors are inserted into the deque. At this point, the deque as the following contents:

front	22	18	15	back

The next cell to be explored will be 22, but cells 18 and 15 are also known to be reachable, and are waiting to be considered should the current path not prove to be a solution. This in fact occurs when we reach cell 16, at which point the deque looks as follows:

front	16	17	18	15	back

Since cell 16 adds no new values to the deque (having no unvisited neighbors) the next entry is automatically popped from the deque. In this fashion we start pursuing the path from 17, which also immediately dead-ends. Finally, the entry 18 is popped from the deque, and the search continues. The solution is ultimately found in fifteen steps. The following shows the path to the solution, with the cells numbered in the order in which they were considered.



The code to perform this search is found in two methods. The overall control is the function solveMaze in class maze. This function pulls cells from the deque as long as the deque remains nonempty and the solution has not yet been found.

```
void maze::solveMaze ()
    // solve the maze puzzle
{
    start->visit (path);
    while ((! finished) && (! path.empty ())) {
        cell * current = path.front ();
        path.pop_front ();
        finished = current->visit (path);
        }
    if (! finished)
        cout << "no solution found\n";
}
```

When each cell is visited, it places all unvisited neighbors into the deque.

```
bool cell::visit (deque<cell *> & path)
    // visit cell, place neighbors into queue
    // return true if solution is found
{
    if (visited)    // already been here
        return false;
    visited = true; // mark as visited
    cout << "visiting cell " << number << endl;
    if (number == 1) {
        cout << "puzzle solved\n";
        return true;
        }

        // put neighbors into deque
    list <cell *>::iterator start, stop;
```

```
    start = neighbors.begin ();
    stop = neighbors.end ();
    for ( ; start != stop; ++start)
        if (! (*start)->visited)
            path.push_front (*start);
    return false;
}
```

The strategy embodied in this code doggedly pursues a single path until it either reaches a dead-end or until the solution is found. When a dead-end is encountered, the most recent alternative path is reactivated, and the search continues. This approach is called *depth-first search*, since it moves deeply into the structure before examining alternatives. Depth first search is the type of search a single individual might perform in walking through a maze.

Suppose, on the other hand, that there is a group of people walking through the maze. When a choice of alternative directions is encountered, the group may decide to split itself into two smaller groups, and pursue each path simultaneously. When another choice is reached the group again splits, and so on. In this manner all potential paths are investigated at the same time. Such a strategy is known as *breadth first search*.

What is intriguing about the maze searching algorithm is that the code for breadth first search is almost identical to the code for depth first search. In fact, all that is necessary is to change the command path.push_front in the visited member function to instead perform a path.push_back.

```
bool cell::visit (deque<cell *> & path)
{
    ...
    for ( ; start != stop; ++start)
        if (! (*start)->visited)
            path.push_back (*start); // ¡- note change
    return false;
}
```

In doing so, we change our use of the deque from being stack-like, to being queue-like. This can be illustrated by once more describing the state of the deque at various points during execution. For example, after the first step, the deque has the following values. Note how the elements are in the opposite order from the one they held in the depth first searching algorithm.

front	15	19	back

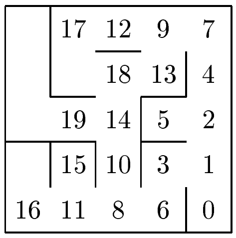
The element 15 is pulled from the deque, but its neighbors, the cells 10 and 14, are placed on the *back* of the queue. The next node to be investigated will therefore not be one of the immediate neighbors of the most recent node, but an entirely different path altogether.

front	15	10	14	back

A few steps later the search has been split several times, and the deque contains the following values:

front	17	21	2	8	8	12	back

As one might expect, a breadth first search is more thorough, but may require more time than a depth first search. Recall that the depth first search was able to discover the solution in 15 steps. The depth first search is still looking after 20 steps. The following describes the search at this point.



Trace carefully the sequence of the last few cells that were visited. Note how the search has jumped around all over the maze, exploring a number of different alternatives at the same time. Another way to imagine breadth first search is that it describes what would happen if one were to pour ink into the maze at the starting location, as the ink slowly permeates every path until the solution is reached.

Depth first and breadth first are both valuable techniques in a variety of searching problems, and arise in a number of different forms and contexts. The following differences can be noted in comparing breadth-first and depth-first searching.

- Since all paths of length one are investigated before examining paths of length two, and all paths of length two before examining paths of length three, a breadth-first search is guaranteed to always discover a path from start to goal containing fewest steps, whenever such a path exists.

- Since one path is investigated before any alternatives are examined, a depth first search *may*, if it is lucky, discover a solution more quickly than the equivalent breadth-first algorithm. Notice this occurs here, where the goal is encountered after examining only 15 locations in the depth-first algorithm, while the goal is only reached after

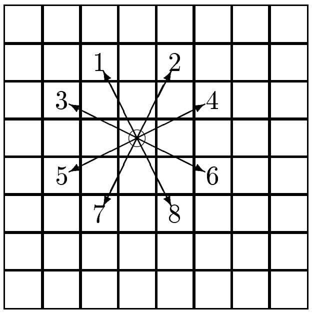


Figure 11.2: Legal knight moves

25 iterations in the breadth-first algorithm. But this benefit is not certain, and a bad selection of alternatives to pursue can lead to many dead-end searches before the proper path to the goal is revealed.

- In particular, suppose for a particular problem that some but not all paths are infinite, and there exists at least one path from start to goal that is finite. Breadth-first search is guaranteed to find a shortest solution. Depth first search may have the unfortunate luck to pursue a never-ending path, and can hence fail to find a solution.

11.3 Application–A Framework for Backtracking

If we generalize the approach used in the depth first search solution of the maze, we discover a technique termed *backtracking*. To use backtracking, a problem must have the characteristic that a solution is discovered as a sequence of steps. At some of these steps there may be multiple alternative choices for the next step, and insufficient information to decide which alternative will ultimately be the correct choice. A stack is used to record the state of the computation at the point of choice, permitting the program to subsequently "restart" the calculation from that point and pursue a different alternative.

To illustrate the idea of backtracking, we will analyze a classic puzzle involving the knight chess-piece. In chess, a knight can legally move in an "L" shaped pattern, either one forward or backward and two left or right, or two forward or backward and one left or right. Figure 11.2 illustrates the legal moves for a knight starting in the given position on a conventional 8-square chess board. A piece may not move off the board, so near the edges of the board the number of legal moves may be less than eight.

A *knights-tour* is a sequence of 64 moves in which a knight visits, using only legal moves, each and every square on the board once. The classic knights-tour problem is to discover a knights tour starting from a specific location. For example, the following table shows the steps in a complete knights-tour starting from the upper left corner.

1	10	31	64	33	26	53	62
12	7	28	25	30	63	34	51
9	2	11	32	27	52	61	54
6	13	8	29	24	35	50	41
3	18	5	36	49	40	55	60
14	21	16	23	46	57	42	39
17	4	19	48	37	44	59	56
20	15	22	45	58	47	38	43

As an illustration of backtracking, consider the following state in which our program finds itself rather early in the search for a solution after having successfully performed 57 moves. There is no legal unvisited location to which the piece at move 57 can advance. The program will "back-up" to move 56, and try a different alternative. But in fact there is no other alternative possible at move 56, and so the program will back up to move 55, and then to 54, 53, and 52. It is only at move 52 that a new untried legal alternative is discovered, namely to move to the bottom left corner. This move is tried, but then immediately abandoned since there is no successor. No further alternative is possible for move 52, nor for move 51, or 50. We must backtrack all the way to move 49 before we can find another possibility, which is to make the new move 50 be the now vacant location of the previous move 52.

1	10	31		33	26	57	42
12	7	28	25	30	43	50	
9	2	11	32	27	34	41	56
6	13	8	29	24	49	44	51
3	18	5	38	35	40	55	
14	21	16	23	48	37	52	45
17	4	19	36	39	46		54
20	15	22	47		53		

Non-recursive programs that solve a problem using backtracking generally have a very similar structure. We can use this observation to develop a generic *framework* for such problems. A framework is a class (or, in more complicated situations, a set of classes and functions), that together provide the skeleton outline for the solution to some problem, but do not provide any specific details. The most common frameworks are associated with graphical user interfaces, but many other types of frameworks are possible. To generate a solution to a specific problem the programmer specializes the framework, generally using inheritance.

11.3.1 Specialization using Inheritance

As we noted in Chapter 9, *inheritance* is a powerful mechanism for quickly and easily creating new data abstractions that are variations or extensions of existing software. To use inheritance, the programmer writes the name of the existing class (called the *parent* class) following the name of the new class. By doing so, the new class (called the *child* class, or sometimes the *derived* class) is then treated as an extension of the older class. All data fields, all member functions of the existing structure are therefore immediately accessible in the new structure. In addition, the programmer can add new data fields and new functions.

We will illustrate the use of inheritance in the development of our framework for backtracking problems. The solution to a generic backtracking problem can be described as follows:

```
template <class T> bool backtrackFramework<T>::run ()
{
    // initialize the problem
    initialize ();
    done = false;

    // do the main loop
    while ((! done) && (! theStack.empty ())) {
        // if we can't advance from current state
        // then pop the stack
        if (! advance(theStack.top ()))
            theStack.pop ();
        }

    // return true if stack is not empty
    return ! theStack.empty ();
}
```

A procedure `initialize` is used to establish whatever conditions need be set to start the problem; including pushing the first state on the stack. A Boolean variable `done` indicates when the problem is finished. This variable may be set by the application specific code to terminate the loop early if, for example, a solution is found. The heart of the algorithm is a simple loop. At each step a procedure called `advance` is called, passing as argument the current state. If it is possible to advance to a new state then execution continues, otherwise the topmost state is popped off the stack, and execution backtracks to a previous point.

The method just given is from a class called `backtrackFramework`. The template parameter is the class used to encode state information. The complete class description is as follows:

```
//
//      class backtrackFramework
//          general framework for solving problems
//          involving backtracking
//

template <class T> class backtrackFramework {
public:
    // protocol for backtrack problems
    virtual void    initialize     ();
    virtual bool    advance        (T newstate);
    virtual bool    run            ();

protected:
    stack < deque <T> >  theStack;
    bool            done;
};
```

Note that the general purpose class has no information specific to the knights tour problem. In order to specialize this general approach to make a solution to the knights-tour problem, we first need to describe how we record information concerning the current position.

The encapsulation for the "state" of the search at any point will be an instance of a class we will call `Position`. A `Position` corresponds to a location on the chess board. A position will maintain a pair of `x` and `y` values corresponding to coordinates on the board, a variable `moveNumber` that will record the sequence of moves in the solution, and a fourth integer variable, named `visited`, that will indicate what subsequent moves have been attempted. To encode this last value, we will simply try, at each step, the moves in order and numbered as in Figure 11.2. A zero stored in the `visited` variable indicates the position has not yet been used on the knights tour, while a non-zero value indicates the position has been used on the knights tour, and furthermore the type of move that was used to generate the next step.

The following is the class description for `Position`. The output operator is used to print the final result.

```
//
//      class Position
//          record a position in the knights move tour
```

```
//

class Position {     // position in chessboard
public:
    void        init         (int, int);
    Position *  nextPosition ();

protected:
    // data fields
    // x and y are coordinate positions
    int    x, y;
    // moveNumber records the sequence of steps
    int    moveNumber;
    // visited is a bit vector marking what positions have been visited
    int    visited;

    // internal method used to find the next move
    Position * findMove(int visitedPosition);

    // friends
    friend class knightsTour;
    friend ostream & operator << (ostream & out, Position & v);
};
```

The chessboard itself will simply be declared as a two-dimensional matrix of positions, named `board`. Because the array constructor does not permit the initialization of each position individually, initialization of each element is performed with a loop that simply invokes the `init` method for each value. We will see this shortly in the initialization portion of the program.

A hallmark of object-oriented programming and the responsibility driven design technique we outlined in Chapter 2 is the concept of making data structures responsible for their own operation. The `Position` data structure illustrates this idea. Each position is responsible for finding the next potential move in the solution. The process of finding the solution is performed by the method `nextPosition`. This method returns a pointer to a position, returning a null pointer if no legal alternative exists. In order to discover a position, the method increments the value held in the variable `visited`, using the facilitator method `findMove` to perform the encoding of the number into a position value. If the incremented value of the `visited` variable denotes a position that is legal and not yet visited, it is returned. Otherwise the loop continues. If all 8 possible moves have been examined, then no alternative exists and a null value is returned. Before returning in this case we zero the variable `visited`, so that the position can be revisited along a different path. We saw this in the earlier example of backtracking, where position 52 was first abandoned but later reached from a different direction.

```
Position * Position::nextPosition()
{
    while (++visited < 9) {
        Position * next = findMove(visited);
        // if there is a neighbor not visited then return it
        if ((next != 0) && (next->visited == 0))
            return next;
        }
    // can't move to any neighbor, report failure
    visited = 0;
    return 0;
}
```

The method `findMove` simply translates a value between 1 and 8 into a position, filtering out moves that are not on the board.

```
Position * Position::findMove(int typ)
{    int nx, ny;

    switch(typ) {
        case 1: nx = x - 1; ny = y - 2; break;
        case 2: nx = x + 1; ny = y - 2; break;
        case 3: nx = x - 2; ny = y - 1; break;
        case 4: nx = x + 2; ny = y - 1; break;
        case 5: nx = x - 2; ny = y + 1; break;
        case 6: nx = x + 2; ny = y + 1; break;
        case 7: nx = x - 1; ny = y + 2; break;
        case 8: nx = x + 1; ny = y + 2; break;
        }
    // return null value on illegal positions
    if ((nx < 0) || (ny < 0))
        return 0;
    if ((nx >= boardSize) || (ny >= boardSize))
        return 0;

    // return address of new position
    return & board[nx][ny];
}
```

We are finally in a position to show how to use inheritance to specialize the general purpose backtracking solution we created earlier in order to create a solution to the knights tour problem. To create a solution to the knights tour problem we need to tie the `Position` data structure into our backtracking framework. By saying that the new class knightsTour, inherits from backtrackFramework (a process termed *derivation*), all the data fields and methods of the parent class are made available to the new class. In addition, those methods that were labeled virtual in the parent class can be *overridden*, and provided with new meanings. It is in this fashion that the initialize and the advance methods can be made specific to the current problem. The complete class description is as follows:

```
//
//      class knightsTour
//          solve the n by n knights tour problem
//

class knightsTour : public backtrackFramework<Position *> {
public:
        // redefine the backtracking protocol
    virtual void    initialize      ();
    virtual bool    advance         (Position *);

        // new method
    void            solve           ();
};
```

The initialization method loops over each board position to establish the initial conditions for each value. It then pushes the starting location, board position 0:0, on to the stack. This board position is our initial state.

```
const int boardSize = 8;
Position board[boardSize][boardSize];

void knightsTour::initialize ()
{
    // initialize the parent class
    backtrackFramework<Position *>::initialize ();

    // initialize chessboard
    for (int i = 0; i < 8; i++)
        for (int j = 0; j < 8; j++)
            board[i][j].init(i, j);
```

Overriding, Replacement and Refinement

Placing the modifier `virtual` in the parent class indicates that the given method can potentially be replaced by a function defined in a child class. This is only a potential, it need not be replaced, and if not the function in the parent will be used.

If a child class does override the parent class method, it is a complete replacement of the method from parent class. Sometimes, as in the initialization method shown, we instead want to *combine* the new code with that of the parent, making sure that both are executed. In C++ this is accomplished by invoking the function in the parent class from inside the code for the child class. In order to avoid confusion, the fully qualified name, in this case

```
backtrackFramework<Position *>::initialize ();
```

is used to completely and fully specify exactly what function should be executed.

```
    // set move number on first position
    board[0][0].moveNumber = 1;

    // push initial position
    theStack.push(& board[0][0]);
}
```

To complete the framework we need only describe how to discover the next move from any given position. This is performed by the method `advance`. The `advance` method is given, in the argument, the current state. It asks a position to try to find a next position in sequence. It does this by invoking the `nextMove` method we described earlier. The `advance` function returns a true value if advancement is possible from the current position, and a false value if no advancement can be made. An additional responsibility is to test to see if the solution to the problem has been found. If so, then the `done` flag must be set.

```
bool knightsTour::advance(Position * currentPosition)
    // try to advance from a given position
{
    Position * newPosition = currentPosition->nextPosition ();
    if (newPosition) {
        // move forward
        newPosition->moveNumber = currentPosition->moveNumber + 1;
        theStack.push(newPosition);
```

```
        // if we have filled all squares we are done
        if (newPosition->moveNumber ==  boardSize * boardSize)
                done = true;
        // return success
        return true;
        }
    else
        return false;    // can't move forward
}
```

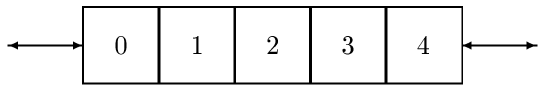
The final method to describe is the one new method added by class `knightsTour` to the framework protocol. This method simply starts the framework running. If success is reported then the stack is popped in order to print the result.

```
void knightsTour::solve ()
{
    // start framework
    if (run ()) {    // print solution
        cout << "solution is:\n";
        while (! theStack.empty ()) {
            cout << * theStack.top () << '\n';
            theStack.pop ();
            }
        }
    else
        cout << "no solution ";
}
```

11.4 An Implementation

Of all the containers in the standard template library, the deque is the one with the least obvious implementation approach. Techniques for implementing vectors, lists, queues, trees and so on are all well known. But there are many possible techniques that can be used to implement a deque, and the language definition does not constrain the software developer of the abstraction in any significant regard. In this section we will describe one possible technique. The approach presented here has the advantage of being relatively simple, but is not the only possibility.

The basic idea in this implementation is to internally represent a deque as a pair of vectors. That is, while we visualize a deque as a linear structure, such as the following:



it actually can be stored internally as two vectors. This allows values to be added at either end. Note, however, that the first vector is "backwards", as the first element is at the top, while the last element is at the bottom, in position 0.

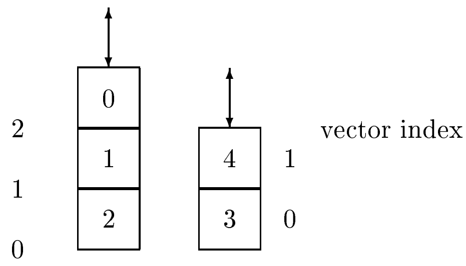


Figure 11.3 gives a class description for our simplified deque implementation, with many of the shorter methods being defined as in-line procedures.

The structure of most of the remaining operations is very similar. All are implemented by performing operations on one or the other of the underlying vectors. The complicating factor is that either vector could potentially be empty, in which case the operation must be performed on the other. The method front, which returns the first element in the collection, is typical:

```
template <class T> T & deque<T>::front ()
    // return first element in deque
{
    if (vecOne.empty ())
        return vecTwo.front ();
    else
        return vecOne.back ();
}
```

If vector one is empty, then the first value in the deque is the first value in vector two. If, on the other hand, vector one is not empty, then the first value in the deque is the *last* value in the first vector. The method back is similar. The methods to remove an element from either the front or the back of the collection are only slightly more complex:

```
template <class T> void deque<T>::pop_front ()
    // remove first element in deque
```

```
//
//     class deque
//         double ended queue

template <class T> class deque {
public:
    typedef dequeIterator<T> iterator;
    typedef T value_type;

        // constructors
    deque () : vecOne(), vecTwo() { }
    deque (unsigned int size, T & initial) : vecOne (size/2, initial),
        vecTwo (size - (size / 2), initial)  { }
    deque (deque<T> & d) : vecOne(d.vecOne), vecTwo(d.vecTwo) { }

        // operations
    T &     operator [ ] (unsigned int);
    T &     front ();
    T &     back ();
    bool    empty () { return vecOne.empty () && vecTwo.empty (); }
    iterator begin () { return iterator(this, 0); }
    iterator end () { return iterator(this, size ()); }
    void    erase (iterator);
    void    erase (iterator, iterator);
    void    insert (iterator, T &);
    int     size () { return vecOne.size () + vecTwo.size (); }
    void    push_front (T & value) { vecOne.push_back(value); }
    void    push_back (T & value) { vecTwo.push_back(value); }
    void    pop_front ();
    void    pop_back ();

protected:
    vector<T> vecOne;
    vector<T> vecTwo;
};
```

Figure 11.3: A simplified implementation of the class deque.

```
{
    if (vecOne.empty ())
        vecTwo.erase(vecTwo.begin ());
    else
        vecOne.pop_back ();
}
```

If the first vector is empty, we must erase the first element in the second vector, otherwise we can simply reduce the size of the first vector by one. Note that this approach may not be the best solution, as the erase operation on vector two is almost undoubtedly very costly. Often sequences of pushes and pops occur one after another. This would occur, for example, if the deque were used as a stack or as a queue. Rather than each pop_front causing an erase, a better alternative in this case would be to move some number of elements (for example half) from the second vector back to the first. Subsequent pop_front operations would then encounter the more efficient pop_back alternative, rather than the erase. The development of this possibility will be explored in some of the exercises at the end of the chapter.

The subscript operator changes the index values into a subscript that is appropriate for one of the underlying vectors. Notice that the subscripts must be reversed for the first vector:

```
template <class T> T & deque<T>::operator [ ] (unsigned int index)
    // return given element from deque
{
    int n = vecOne.size ();
    if (index <= n)
        return vecOne [ (n-1) - index ];
    else
        return vecTwo [ index - n ];
}
```

11.4.1 Deque Iterators

An iterator for our deque abstraction is most easily constructed by using the subscript operator to access the underlying element. Such an approach is shown in the class description in Figure 11.4.

As with many of the iterator implementations we present, a major difficulty arises from the need to support both a prefix and a postfix form of the iterator operation. The prefix form is implemented in-line, as it simply changes the state of the current iterator and returns. The postfix form must change the current state, but return an iterator that describes the location prior to the change. This is most easily accomplished by cloning the current iterator, which will preserve the initial state, then updating the value, and finally returning the clone.

```
//
//    class dequeIterator
//        iterator protocol for deque

template <class T> class dequeIterator {
    friend class deque<T>;
    typedef dequeIterator<T> iterator;
public:
        // constructors
    dequeIterator (deque<T> * d, int i) : theDeque(d), index(i) { }
    dequeIterator (deque<T>::iterator & d)
        : theDeque(d.theDeque), index(d.index) { }

        // iterator operations
    T & operator * () { return (*theDeque)[index]; }
    iterator & operator ++ (int) { ++index; return this; }
    iterator operator ++ (); // prefix change
    iterator & operator -- (int) { --index; return this; }
    iterator operator -- (); // postfix change
    bool operator == (iterator & r)
        { return theDeque == r.theDeque && index == r.index; }
    bool operator < (iterator & r)
        { return theDeque == r.theDeque && index < r.index; }
    T & operator [ ] (unsigned int i)
        { return (*theDeque) [index + i]; }
    void operator = (iterator & r)
        { theDeque = r.theDeque; index = r.index; }
    iterator operator + (int i)
        { return iterator(theDeque, index + i); }
    iterator operator - (int i)
        { return iterator(theDeque, index - i); }

protected:
    deque<T> * theDeque;
    int index;
};
```

Figure 11.4: Implementation of a deque iterator.

```
template <class T> deque<T>::iterator dequeIterator<T>::operator ++ (int)
    // postfix form of increment
{
        // clone, update, return clone
    deque<T>::iterator clone(theDeque, index);
    index++;
    return clone;
}
```

Having described the structure of deque iterators, we can now return to the description of those deque methods that make use of iterators. The method `erase` recovers the index position from the iterator, and erases an element from the appropriate vector. It uses the fact that vectors construct random access iterators, and so we can easily create the iterator that corresponds to a given index position within a vector.

```
template <class T> void deque<T>::erase (deque<T>::iterator & itr)
    // erase value from deque
{
    int index = itr.index;
    int n = vecOne.size ();
    if (index < n)
        vecOne.erase (vecOne.begin () + ((n-1) - index));
    else
        vecTwo.erase (vecTwo.begin () + (n - index));
}
```

The `insert` method is similar, and will not be shown. The erase method that removes a range of values is more complicated, since the range may cross the boundary between the two vectors. The implementation of this method therefore divides into three cases, depending upon whether all the elements to be removed are in the first vector, are all in the second vector, or whether some are in the first vector and some are in the second. The development of this code will be left as an exercise.

11.5 Chapter Summary

Key Concepts

- deque
- Depth first search

- Breadth first search
- Backtracking
- Framework
- Inheritance
- virtual member functions

The deque, or double-ended-queue, is a data structure that provides a combination of features from both the vector and list data types. Like a vector, a deque is a randomly accessible and indexed data structure. Like a list, elements can be efficiently inserted at either the front or the end of the structure. Thus a deque can be used in either a stack-like or a queue-like fashion.

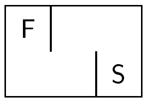
We have illustrated the use of the deque by developing a program that can find a path through a simple maze. By storing the intermediate steps in a stack, the search technique will explore one path entirely to completion before examining alternatives. This is known as *depth first* search. By changing the use of the deque to a queue-like form, all paths are explored in parallel. This is known as *breadth first* search.

We have introduced *backtracking* as a general problem solving technique, applicable whenever the "state" of the task at hand can be captured and stored at the point where one of many alternative possibilities must be selected.

Finally, we introduced the idea of a software *framework*. A framework describes the general structure of a solution to a problem, without defining any specific details. These details can then be filled in, typically using inheritance, to specialize the framework for the solution of a given problem. A framework thus provides reuse not only for code but also for the reuse of an idea or approach to solving a class of similar problems.

Study Questions

1. Give a short characterization of the deque data type.
2. What vector operations are not supported by the class deque? What list operations?
3. Give the integer encoding of the following simple six-cell maze:



4. Show the state of the deque in the maze example (Section 11.2) after five moves have been performed.

5. How does breadth first search differ from depth first search?

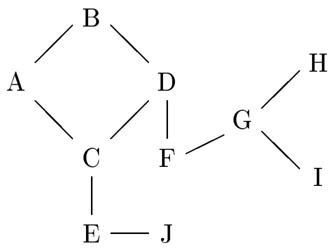
6. What requirements must be satisfied in order to use backtracking in the solution of a problem?

7. What is a framework?

8. What is the underlying representation used to hold the values in the implementation of the `deque` described in this chapter?

Exercises

1. Consider the following graph. Starting from node A, list the vertices as they might be visited in a breadth first search, and as they might be visited using a depth first search. Note that there are many different sequences for both forms of search.



2. The maze solving program exploits a redundancy in the maze encoding. This redundancy come from the assumption that if one can move east from one cell to the next, one can also move west from the second cell back to the first. The constructor `maze::maze` makes use of this redundancy, by only processing openings to the north and west.

   However, this redundancy is not intrinsic to the numeric encoding of the maze. A "one-way" opening could easily be described as a cell with value 3, for example, being next to a call with value 7. From the 7 cell one could move to the 3 cell, but from the 3 cell one could not move back to the 7 cell.

   To change our maze solving program it is only necessary to change the constructor that translates the external encoding into the internal encoding. Show the modifications that must be provided to permit this form of maze.

3. A difficulty occurs when the first element is removed from a `deque`, but the first vector in the internal representation is empty. The `pop` then causes the first element to be removed from the second vector. This is potentially costly if a number of pops are

performed in sequence. A better alternative is to first move half the elements from the second vector into the first vector, so that subsequent pops are implemented as operations on the first vector. Write implementations for pop_front and pop_back that use this idea.

4. The implementation of the erase method that takes a range of values as argument must recognize three different cases:

   (a) Both beginning and end of range are found in the first vector.

   (b) Both beginning and end of range are found in the second vector.

   (c) Beginning of range is found in first vector, end of range in second vector.

   Implement the code for the erase method that handles each of these cases.