Table of Contents

Table of Contents — continued

Solutions to Problems and Exercises in Chapter 1

1. If the array A contains no negative integers, A[MaxIndex] will eventually be evaluated in the while-condition of the while-loop. If array index bounds checking is turned on, the erroneous array access A[MaxIndex], in which MaxIndex is out of bounds of the index range 0:MaxIndex–1 of A, will be detected. The error can be fixed by replacing line 5 with

   ```
   5    |   while ( (i < MaxIndex) && (A[i] >= 0) ) {
   ```

   This works because in the "short-circuit and expression" A && B, the subexpression A is evaluated before evaluating the subexpression B, and if A's value is false (i.e., 0), then the value of the entire expression A && B is taken to be false without evaluating B. Hence, when i's value is increased so that i == MaxIndex, the subexpression (i < MaxIndex) evaluates to false and the erroneous out-of-range access of A[MaxIndex] in (A[i] >= 0) is never performed.
2. The statement of the problem on page 16 states that, "In order to avoid doing useless work, the solution should exit as soon as the first negative integer is discovered." The proposed solution does not exit as soon as the first negative integer is discovered. Rather, it examines all array items in descending order and saves the index of the most recent negative integer encountered in descending order. The conditions of the problem statement are therefore not satisfied.
3. Hardware tends to come and go rather rapidly. Vendors release new models of computers, usually with increased performance or lower cost, at frequent intervals in order to keep up with the competition. Programming languages and software tend to ride out the rapid shifts in underlying computers by being rehosted on new platforms — although software versions go through rapid evolution, too, in response to competitive market forces. The fundamental laws of computing seem to have the greatest longevity. For example, the discovery of the $n \log n$ barrier for comparison-based sorting has endured since its discovery. Various human-computer interfaces undergo evolution as the decades pass. The early paper-tape and punched card inputs and electric typewriter or teletype outputs gave way to cathode ray tube screens with lines of characters, and these, in turn, gave way to windows, mice, and the "desktop metaphor." If versatile,

Answers to Review Questions 2.3

1. When we *dereference a pointer*, we follow the pointer to the unit of storage for which the pointer value is the address (in some addressable storage medium in which units of storage have unique addresses that identify them). If A is a pointer variable in C, the notation *A dereferences the pointer value in A.
2. We can say that: B *links* to A, or that A is B's *referent*, or that B *references* A.
3. typedef int *IntegerPointer;
4. Execute the assignment A = (T*) malloc(sizeof(T)), after which A contains a pointer to a new block of storage of type T.
5. *A = 19 assigns 19 to be the value of A's referent. The expression 2*(*A) has a value equal to twice the value of A's referent.
6. It is a type mismatch, since A must take pointer values, and since 5 is not a pointer value, but rather is an integer value.
7. Two different expressions that reference the same unit of storage are called aliases.
8. You execute the function call free(P), where P contains a pointer to the storage to be recycled.
9. Dynamically allocated storage becomes inaccessible when there are no pointers to it that can be reached either directly as values of pointer variables, or indirectly by following links in data structures along a path that can reach it.
10. Garbage is inaccessible dynamically allocated storage that is no longer needed during the execution of a program.
11. A dangling pointer is a pointer to a unit of storage that has been returned to the pool of unallocated dynamic storage.
12. The scope of a variable in C relates to the *lifetime* of its existence and the places it is *visible*. During the time a variable exists and is visible (by virtue of its name not being hidden by another variable of the same name that has a more local scope), the value stored in the storage location associated with the variable can be both assigned and accessed, using the variable's name. The scope of a unit of dynamic storage allocated in C can be thought of as *global* to the scope of named variables in a C program. This means that the lifetime of a unit of dynamically allocated storage lasts from the moment the unit is allocated (using malloc) until the moment the unit's storage is reclaimed (using free(P)). So long as a pointer to a unit of storage exists, and so long as that unit of storage has not

been reclaimed, the value in the unit can be accessed and a new value can be assigned to it.

13. Since units of dynamically allocated storage do not have names in a C program, they are sometimes called *anonymous variables*, meaning variables with no names. Names for variables are created in the text of a C program, but, since dynamically allocated storage is created at the time a program is executed, rather than when the program is written, there is no way to give textual names to units of dynamically allocated storage. However, the pointer to a unit of dynamically allocated storage acts in place of the name of an ordinary variable, since, using the pointer, the value stored in the storage unit can be both assigned and accessed.

Solutions to Exercises 2.3

1. It prints 7.
2. It prints 5.
3. Answer depends on the behavior of your C system and is determined by experiment.
4. Answer depends on the behavior of your C system and is determined by experiment.

Answers to Review Questions 2.4

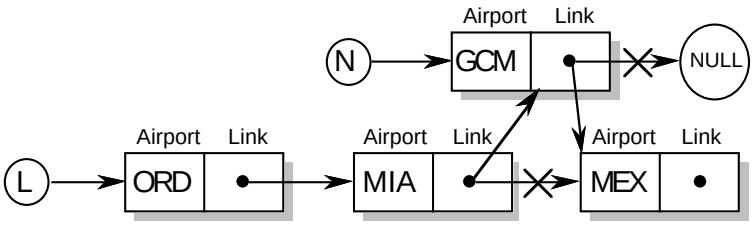
1. The *null address* is a special address that is not the address of any node, and which, by convention, is used to indicate the end of linked lists.
2. By a dot (•) or by a dot connected to an arrow pointing to the value NULL.
3. The value NULL represents the null address in C.
4. By a solid dot (•) in a link field.
5. An empty linked list is a list L having no nodes on it. By convention, it is indicated by the value NULL.
6. Explicit pointer variable notation consists of a box containing the tail of an arrow representing the pointer value, where the box is labeled on the left with the name of the pointer variable followed by a colon. Implicit pointer variable notation consists of an oval containing the pointer variable name in which an arrow connects the boundary of the oval to the referent of the pointer value. The two notations are equivalent. The implicit pointer variable notation is used in most diagrams in the book.

7. By a question mark (?), or, in the case of a pointer to an unknown location, by an arrow pointing to a circle containing a question mark.

Solutions to Exercises 2.4

1. (a) NULL, (b) GCM, (c) MEX, (d) NULL.
2. (a) L–>Link, (b) L–>Link–>Link, (c) L, (d) *L.
3. N–>Link = L–>Link–>Link; L–>Link–>Link = N;

4.



5. strcpy(L–>Link–>Airport, "JFK");
6. N–>Link = L–>Link–>Link; free(L–>Link); L–>Link = N;

Answers to Review Questions 2.5

1. In *top-down programming using stepwise refinement*, one starts with an outline of a program which leaves out specific details, and one progressively fills in more details by a process known as *stepwise refinement*. In *stepwise refinement*, at each step more specific detail is filled in, until finally, a specific executable program has been created, written in an actual programming language.
2. We can define a struct for a NodeType that is tagged with the name NodeTag and we can use this tag to define a Link member of the struct whose type is a pointer to the NodeType struct being defined, as shown in the type definition struct given at the beginning of Exercises 2.5 on page 53.
3. The value NULL belongs to every pointer type in C.

Solutions to Exercises 2.5

1.

```
void InsertNewFirstNode(AirportCode A, NodeType **L)
{
    NodeType *N;

    /* Allocate a new node and let the pointer variable N point to it */
        N = (NodeType *) malloc(sizeof(NodeType));
    /* Set the Airport field of N's referent to A */
        strcpy(N–> Airport, A);
    /* Change the Link field of N's referent to point to the first node of list L */
        N–> Link = *L ;
    /* Change L to point to the node that N points to */
        *L = N;
}
```

2.

```
void  DeleteFirst(NodeType **L)
{
    NodeType *N;

    if (*L != NULL) {
        N =  *L;
        *L = (*L)–>Link;
        free(N);
    }
}
```

3.

```
void  InsertBefore(NodeType *N, NodeType *M)
{
    AirportCode A;

    /* insert node M after node N on list L */
        M–>Link = N–>Link;
        N–>Link = M;
    /* swap airport codes in N and M */
        strcpy(A, N–>Airport);
        strcpy(N–>Airport, M–>Airport);
        strcpy(M–>Airport, A);
}
```

4.

```
NodeType *Copy(NodeType *L)
{
    NodeType *N, *M, *L2;

    if (L == NULL) {

        return NULL;

    } else {

        /* initialization and copying of first node */

            M = (NodeType *) malloc(sizeof(NodeType));
            L2 = M;
            N = L;
            strcpy(M–>Airport, N–>Airport);

        /* L2 points to the copy of the list L being constructed */
        /* N is a pointer that steps along the nodes of L to copy */
        /* M is a pointer that steps along the nodes of L2 that are copies */
```

```
        /* of the corresponding nodes in L that N points to */

            while (N–>Link != NULL) {
                M–>Link = (NodeType *) malloc(sizeof(NodeType));
                N = N–>Link;              /* create new last node on copy list */
                M = M–>Link;              /* advance pointers on both lists */
                strcpy(M–>Airport, N–>Airport);          /* copy airport code */
            }

        /* mark last node of copy as the end of the list L2 */
            M–>Link = NULL;

        /* return pointer to first node of L2 as the function result */
            return L2;

    }
}
```

5.

```
void Reverse(NodeType **L)
{
    NodeType *R, *N;

    R = NULL;              /* initialize R, the reversed list, to be the empty list */

    while (*L != NULL) {
        N =  *L;                                    /* let N point to L's first node */
        *L = (*L)–>Link;              /* now, let L point to the remainder of L */
        N–>Link = R;                                /* link N to the rest of R */
        R = N;                        /* and make R point to its new first node N */
    }

    *L = R;               /* finally, replace L by a pointer to the reversed list R */

}
```

6. If L is the null list (meaning L = NULL), then, when the condition in the while-loop is evaluated, the attempt to find the value of (strcmp(L–>Airport, A) != 0) will try to dereference the null pointer, NULL. This error can be fixed by reversing the order of the operands of the short-circuit && operator in the while-condition on line 4 giving: while ((L != NULL) && (strcmp(L–>Airport, A) != 0)) {

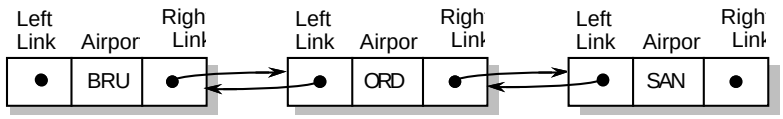
7.

```
NodeType *Concat(NodeType *L1, NodeType *L2)
{
    NodeType *N;

    if  (L1 == NULL) {
        return L2;
    } else {
        N = L1;                                 /* let N point to the first node of L1 */
        while (N–>Link != NULL) N = N–>Link;     /* find the last node of L1 */
        N–>Link = L2;                  /* set the link of the last node of L1 to L2 */
        return L1;                  /* return the pointer to the concatenated lists */
    }

}
```

8. If L points to a list consisting of just one node, then the function call LastNode(L) results in executing a statement that tries to dereference NULL. This is another example of a bug found by checking a boundary case.
9. Suppose L is a list containing only one node. That is, suppose L contains a pointer to a node whose link is NULL. Then the special case on lines 9:12 of Program 2.15 applies, and we free the node *L points to, after which we need to store a NULL pointer in the variable L external to the function call. To do this, we need to use L's external address, &L, as an actual parameter when the function is called and we need to use *L = NULL inside the function in order to store NULL in the location to which the actual parameter &L points. If the function prototype had been void DeleteLastNode(NodeType *L), there would be no way to set the contents of the external variable L to NULL in the case of a list consisting of only one node to be deleted, since the actual parameter passed for the value of L at the time of the function call would be the address of the first node on the list, leaving no way to change the contents of the external variable L after the deletion from a pointer to the first node of the list to a NULL pointer.

Answers to Review Questions 2.6

1. Nodes having two separate pointer fields can be linked into two-way lists, binary trees, and two-way rings.
2. In symmetrically linked lists, nodes point both to their predecessors and successors in the list — except for the first node in the list whose predecessor is NULL and the last node of the list whose successor is NULL.



Solution to Exercise 2.6

1.

```
void Delete(NodeType *L)
{

    /* Make L's predecessor point to its successor */
        if (L–>LeftLink != NULL) {
            L–>LeftLink–>RightLink= L–>RightLink;
        }
```

```
    /* Make L's successor point to its predecessor */
        if (L–>RightLink != NULL) {
            L–>RightLink–>LeftLink= L–>LeftLink;
        }

    /* Dispose of the storage for node L */
        free(L);
}
```

Answers to Review Questions 3.2

1. The base case occurs when a recursive program gives a direct solution to a subproblem without using any recursive calls to compute it.
2. Four decomposition methods: (i) *First & Rest*, where First = *m* and Rest = *m*+1:*n*; (ii) *Last & All but Last*, where Last = *n* and All but Last = *m*:*n*–1, (iii) *Split in Halves*, where LeftHalf = *m*:middle, RightHalf = middle+1:*n*, and middle = (*m* + *n*) / 2; and (iv) *Edges & Center*, where Edges = (*m* & *n*) and Center = *m*+1:*n*–1.
3. A *call tree* is a tree with a recursive call at its root, in which the descendants of each node show the recursive calls made by the call at that node. Calls resulting in base cases have no descendants, and are the so-called leaves of the call tree.
4. A trace of a recursive function call is a sequence of successive lines on which the calling expressions for successive recursive calls are given amidst the values waiting to be combined with the results returned by those recursive calls. When the base cases are reached and values are returned directly from them, the trace shows how the values combine to produce the value returned by the original function call.
5. The number of different combinations of *n* things taken *k* at a time is given by the formula: $\binom{n}{k} = \frac{n!}{k!(n-k)!}$.
6. Decompose a non-empty list into its Head (which is its first node) and its Tail (consisting of the rest of the nodes on the list after the first one).

Solutions to Selected Exercises in Exercises 3.2

1.

```
double Power(double x, int n)
{
    if  (n == 0)  {
        return 1.0;                          /* base case, x^0 == 1.0 */
    } else {
        return x * Power(x, n – 1);                  /* recursion step */
    }
}
```

2.

```
double Power(double x, int n)
{
    double p;

    if  (n == 0)  {

        return 1.0;                          /* base case, x**0 = 1.0 */
```

```
    } else {

        p = Power(x, n / 2);                    /* let p == x to the half(n) power */

        if  ( (n % 2) == 0 ) {
            return p * p;                       /* if n was even, return Square(p) */
        } else {
            return x * p * p;                   /* if n was odd, return x*Square(p) */
        }

    }

}
```

3.

```
int Mult(int m, int n)
{
    if (m == 1) {
        return n;
    } else {
        return n + Mult(m – 1, n);
    }
}
```

4. Recursive Euclid's Algorithm:

```
int  gcd(int m, int n)
{
    if   (n == 0) {
        return m;              /* base case, if remainder is 0, then result is m */
    } else {
        return gcd(n, m % n) ;                                  /* recursion step */
    }
}
```

5. First, to prove that gcd(*m*,*n*) terminates, we know that when *m* is divided by the divisor *n* to obtain a quotient *q* and a remainder *r* = *m* % *n*, the quantities obtained obey the relation: *m* = *q***n* + *r*, where 0 ≤ *r* < *n*. So, the remainder *r* is either 0 or a positive integer less than the divisor *n*. Thus, there can only be a finite number (less than *n*) of successive non-zero remainders, *r*, which decrease by at least 1 before a zero remainder is produced. Once a zero remainder is obtained, the base case of the recursion (on line 4 of gcd(*m*,*n*)) is reached, and the function call terminates. To prove that gcd(*m*,*n*) returns the greatest common divisor, we note that on successive calls, if *n* = 0, then gcd(*m*,*n*) = *m*, and otherwise, gcd(*m*,*n*) = gcd(*n*,*r*). In the latter case, *r* is related to *m* and *n* by the equation *r* = *m* – *q***n*, so any common divisor of the pair (*m*,*n*) is also a common divisor of the pair (*n*,*r*) — including the greatest common divisor. Consequently, the gcd of successive remainder pairs is preserved. Finally, when *n* = 0, gcd(*m*,*n*) = *m*, so

m is the last non-zero remainder in the sequence of remainder pairs, all of which were divisible by the gcd. This implies that *m* itself is the gcd.

6.

```
int Product(int m, int n)
{
    if   (m < n) {
         return m * Product(m+1,n);
    } else {
         return n;
    }
}
```

7. Despite what some conceive to be the elegance and simplicity of the Reverse function given in the solution below, the overall solution is a poor one on two counts: (1) The overall running time is quadratic instead of linear because of repeated scanning and character copying of the successively smaller tails that are reversed, and (2) The scratch storage used to accommodate concatentations and tails is taken from the dynamic storage pool using malloc, and is never freed after use.

```
/****
*
*  Solution to Exercise 3.2.7 in Chapter 3, page 81.
*
****/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char *EmptyString = "";

/* ------------------------------------------------------------------------- */

int Empty(char *S)
{
    return (S[0] == '\0');
}

/* ------------------------------------------------------------ */

char *Head(char *S)
{
    char *t;

    t = (char *) malloc(2*sizeof(char));
    t[0] = S[0];
    t[1] = '\0';

    return t;
}

/* ------------------------------------------------------------ */
```

```
char *Tail(char *S)
{
    char *t, *temp;
    int n;

    n = strlen(S);

    if (n <= 1) {
        return EmptyString;
    }else {
        t = (char *) malloc(n*sizeof(char));
        temp = t;
        S++;
        while ((*t++ = *S++) != '\0')
            ;
    }

    return temp;
}

/* ---------------------------------------------------------------- */

char *Concat(char *S, char *T)    /* see Program 8.17 on page 315 */
{
    char *P;
    char *temp;

    P = (char *) malloc((1+strlen(S)+strlen(T))*sizeof(char));

    temp = P;

    while ((*P++ = *S++) != '\0')
        ;

    P--;

    while ((*P++ = *T++) != '\0')
        ;

    return temp;
}

/* ---------------------------------------------------------------- */

char *Reverse(char *S)                          /* to Reverse a String S */
{
    char *temp;

    if (Empty(S)) {                         /* the empty string is returned */
        return EmptyString;                         /* for the base case */
    } else  {
        return Concat(Reverse(Tail(S)), Head(S));
    }
}


/* ---------------------------------------------------------------------------- */

int main(void)
{
    char *S = "abcdefg", *S1;
```

```
    printf("the string S == %s\n",S);
    S1 = Reverse(S);
    printf("the reverse of S == %s\n",S1);

}

/* ------------------------------------------------------------------------- */
```

8. The answer to exercise 8 is not given.

9.

```
int Length(NodeType *L)                    /* to compute the number of  */
{                                                /* nodes in a linked list L */
    if (L == NULL) {
        return 0;                          /* since the empty list has no nodes in it */
    } else {
        return 1 + Length(L–>Link);        /* length = 1 + length of L's tail */
    }
}
```

10.

```
int Min2(int A[ ], int m, int n)           /* first define an auxiliary function Min2 */
{
    int MinOfRest;

    if (m == n) {
        return A[m];
    } else {
        MinOfRest = Min2(A,m+1,n);
        if  (A[m] < MinOfRest) {
            return A[m];
        } else {
            return MinOfRest;
        }
    }
}

int  Min(int A[ ])      /* to find the smallest integer in an integer array A[0:n–1] */
{
    return Min2(A,0,n–1);
}
```

11.

```
int Ack(int m, int n)                      /* assume m ≥ 0 and n ≥ 0 */
{
    if (m == 0) {
        return (n+1);
    } else if (n == 0) {
        return Ack(m–1,1);
    } else {
        return Ack(m–1,Ack(m,n–1));
    }
}
```

12. The answer to exercise 12 depends on the behavior of your C system.

13. The procedure, P(*n*), writes the digits of the non-negative integer *n*.
14. The procedure, R(*n*), writes the digits of the integer *n* in reverse order.
15. Using the auxiliary function NewtonSqrt(x,epsilon,a), call Sqrt(x,epsilon) in what follows (making sure to #include <math.h> beforehand):

```
double NewtonSqrt(double x, double epsilon, double a)
{
    if (fabs(a*a – x) <= epsilon) {
        return a;
    } else {
        return NewtonSqrt(x, epsilon, (a + x/a)/2.0);
    }
}

double Sqrt(double x, double epsilon)
{
    return NewtonSqrt(x,epsilon,x/2.0);
}
```

16.

```
int C(int n, int k)                    /* where n and k are non-negative integers */
{
    if ((k == 0) | | (n == k)) {
        return 1;
    }else {
        return C(n–1,k) + C(n–1,k–1);
    }
}
```

17. The answer to exercise 17 is not given.

Answers to Review Questions 3.3

1. An *infinite regress* occurs when a recursive program calls itself endlessly.
2. Two programming errors that can cause infinite regresses are: (1) a recursive program with no base case, or (2) a recursive program with a base case that never gets called.
3. An infinite regress causes an unending sequence of recursive calls. To evaluate each such call, the C run-time system allocates a new call frame which it allocates in a run-time call-frame storage region. Thus, the call-frame region will become exhausted when the run-time system attempts endlessly to allocate new call frames.
4. If single precision integers were used as the integer representation of the parameter, *n*, in Program 3.12, then the call Factorial(0) would result in the recursive calls Factorial(0),

Factorial(–1), Factorial(–2), and so on, being made. If, for example, single precision integers are represented by 16-bit, two's complement integers, then after descending to the value –32,768, the values cycle back to 32,767 and descend towards 0 again, eventually reaching the value 1. When Factorial(1) is called, the function terminates with the value 1. Then, all the values in the range –32768:32768 are multiplied together, giving the value 0. This scenario can happen only if there is enough memory space to hold all 65536 calls in the call-frame storage region. But many contemporary computers will have enough memory to do this.

Solution to Exercise 3.3

1. The function call F(2) causes an infinite regress.

Answers to Review Questions 3.4

1. The complexity class that characterizes the recursive solution of the Towers of Hanoi puzzle is called the exponential *complexity class.*

2. The principal disadvantage of the exponential complexity class is that the running time of the algorithms that fall in this class require very large running times for all but very small arguments. Such running times are so large that we cannot expect an answer from these algorithms given a reasonably sized input for a long, long time. Thus, we generally try to avoid using such exponential algorithms.

Solution to Exercise 3.4

1. Since there are $3.1536*10^7$ seconds in one year, and the length of the instruction sequence is $L(n) = 2^n - 1$, we need to find the largest n such that

$$2^n - 1 \leq 3.1536*10^7.$$

The largest such n is 24 (which can be determined by finding the largest n such that $n \leq \log_2(3.1536*10^7 + 1)$, where $\log_2(3.1536*10^7 + 1) = 24.91049639$). Hence, a tower of at most 24 disks can be moved in a year's time.

Answers to Review Questions 4.2

1. A *C module* M is a program consisting of two separate coordinated text files, M.h, its *interface file* (or header file) and M.c, its *implementation file* (or source file), such that M provides a collection of related entities all of which work together to offer a set of capabilities or to provide a set of services or components that can be used to help solve some class of problems.
2. The *interface file* M.h of a module M is a text which declares entities that are visible to (and hence usable by) external users. These entities can include declarations of constants, typedefs, variables, and functions. Only the prototypes of functions are given (and in them, only the argument types, and not the argument names, are given).
3. The *implementation file* M.c of a module M is a program text which declares local program entities that are private to the module (and cannot be seen or used by external users), and which also gives the full declarations of functions whose prototypes are given in the interface header file, M.h, and which are visible to external users.
4. To use the services provided by a module M, you place an include directive which includes the header file M.h at the beginning of your program, using the syntax #include "M.h". The effect is as if the declarations inside M.h had been substituted in your program at the place the include directive is given. The module M is usually compiled separately. The extern declarations in M.h tell the linker how to link in externally defined functions from other modules compiled separately.

Answers to Review Questions 4.3

1. A *priority queue* is a collection of prioritized items in which items can be inserted in any order of ranking of their priorities, but are removed in highest-to-lowest order of their priorities.
2. A priority queue's items could be stored in arrays (or linked lists, or trees, or many other kinds of data structures acting as containers). Considering only arrays as containers, the items could be stored either in sorted or unsorted order (where sorting is done with respect to the order implied by the items' priorities). If an unsorted array is used, you add a new item at the end of the array, and to remove an item, you scan the array to locate the

item of highest priority, remove it, and then move the last item into the hole created by the removal of the highest priority item. For sorted arrays, to insert an item, you move all items of priority greater than the item one space to the right, and insert the item into the hole created. To remove an item you simply remove the last item in the array.

3. Given the sketch in the answer to the previous question, the unsorted array representation has a more efficient insertion operation while the sorted array has a more efficient removal operation. The reason is that these two efficient operations operate on only one item, whereas insertion into a sorted array and removal from an unsorted array potentially require all items in the array either to be scanned or to be moved.

Answers to Exercises 4.3

1.

```
    |  PQItem Remove(PriorityQueue *PQ)
    |  {
80  |      PQItem temp;
    |      PQListNode *NodeToFree;
    |
    |      if ( ! Empty(PQ)) {                    /* result undefined if PQ empty */
    |          NodeToFree = PQ–>ItemList;          /* otherwise, remove the */
85  |          temp = NodeToFree–>NodeItem;        /* highest priority item */
    |          PQ–>ItemList = NodeToFree–>Link;   /* from the front of the list */
    |          PQ–>Count – –;                     /* decrease the item count */
    |          free(NodeToFree);                  /* and free the space for the */
    |          return (temp);                     /* node that was removed */
90  |      }
    |  }
```

2. The items in the ItemArray are stored in positions 0:Count – 1. Given an ItemArray that currently contains Count items, its last item will therefore be found in the Count – 1 position. Thus, to remove the last item, we first decrement the Count member of the PriorityQueue struct, using PQ–>Count – –, and then we move the last item at PQ–>ItemArray[PQ–>Count] into the hole opened up by removal of the maximum priority item at the position PQ–>ItemArray[MaxIndex].

Answers to Review Questions 4.4

1. A *program shell* is a top-level program with “holes” that invokes and uses plug-in modules and organizes the operations and services these modules provide to define the topmost level of operation of the overall program.

2. Using modules can help organize the work in a software project by providing a clean way to break the work of the project into subprojects associated with implementation of the separate modules.
3. Using modules can help structure a software system design during the design phase, by first defining only the interfaces (or header files) of the modules (and not their detailed implementations) so that the modules fit together cleanly in the overall design. The overall design will be clean, in general, if it is composed from a few modules having simple, well-defined interactions.

Answers to Exercises 4.4

1. *Tic-Tac-Toe Program Shell:*

```
/* the following is the text of a TicTacToe Program Shell */

#include <stdio.h>
#include  "TicTacToeUserInterface.h"
#include "MoveCalculationModuleInterface.h"

Move theMove;                   /* theMove contains current player's move*/
Board theBoard;          /* theBoard gives the configuration of the game */

int main (void)
{

     InitializeAndDisplayTicTacToeBoard( );

     do {

          GetAndProcessOneEvent( );  /* Gets and displays user's move */
                                              /* if user clicks in a square */
                                                        /* on the board. */
                                         /* Otherwise, lets user select X or O */
                                        /* to play, and let's user choose who */
                                    /* plays first — the user or the machine */

          if (ItIsMachinesTurnToMove( ) {
               theMove = CalculateMove(theBoard);              /* done by */
                                                    /* MoveCalculationModule*/
               Display(theMove);                                  /* done by */
                                          /* TicTacToeUserInterfaceModule */
          }

          if (GameIsOver( ) ) {
               DisplayResults( );                  /* X wins, O wins, or Draw */
               AskIfUserWantsToPlayAgain( );
          }

     } while ( ! UserWantsToQuit( ) );

     Shutdown( );

}
```

Interface file of a module to handle the Tic-Tac-Toe user interface:

```
/* the text of the file  "TicTacToeUserInterface.h" */

#include <stdio.h>

typedef       enum { false, true } Boolean ;
typedef       enum {X, O, Blank} Token;
typedef       Token Board[3][3];
typedef       struct {
                    int   row;
                    int   column;
              } Move;

extern Boolean ItIsMachinesTurnToMove(void);
extern Boolean UserWantsToQuit(void);
extern void InitializeAndDisplayTicTacToeBoard(void);
extern void GetAndProcessOneEvent(void);
extern void Display(Move);
extern Boolean GameIsOver(void);      /* true if there was a win or draw */
extern void DisplayResults(void);
extern void AskIfUserWantsToPlayAgain(void);
extern void Shutdown(void);

/* end of file "TicTacToeUserInterface.h" */
```

Interface file of a module to calculate the machine's move:

```
/*  the text for the file "MoveCalculationModuleInterface.h" */

#include <stdio.h>
#include "TicTacToeUserInterface.h"    /* included in order to import the */
                                         /* typedefs for Move and Board */
                                                          /* needed below */
extern Move CalculateMove(Board);

/*  end of  file "MoveCalculationModuleInterface.h" */
```

Answers to Review Questions 4.5

1. An *available space list* is a linked list of unused list nodes that is organized as a pool of available storage to allocate during the operation of a list-processing program. It is generally linked together during the initialization of the program. Freshly allocated nodes are removed from it, and freed nodes are returned to it during the execution of the program. It thus provides the basis for a storage allocation and management policy for user-defined linked lists.
2. *Information hiding* occurs when program entities are declared and used locally within a function or module, preventing them from being seen and/or used outside the function or module.

3. *Information hiding* can promote ease of program modification by confining the region of program text in which changes have to be made to a local region of the total program text. When entities are defined locally within a function, or privately inside a module in its implementation file, then changes in the local or private entities can be made without interfering with entities outside.
4. A *representation-independent notation* is a notation which does not have to be changed when its underlying data representation is changed. For example, `GetLink(N)` is a representation-independent notation for getting the `Link` of node `N` in a linked-list, whereas the three expressions `N–>Link`, `ListMemory[N].Link`, and `Link[N]` are representation-dependent notations (from which, by looking at the notations, one can discover what the underlying data representations are).
5. If one does not use a representation-independent notation, then the users of the module will have to use notations that depend on the data representations chosen, in which case the data representation isn't hidden properly (since it can be discovered by studying the notation of use outside the module in which the representation is supposed to be hidden).
6. Efficiency trades off against generality whenever making a program more general makes it more costly to execute, or whenever making it more efficient entails making it less general. Using a representation-dependent notation makes a program more efficient but less general than using a representation-independent notation that hides a specific representation-dependent data access notation behind a general function call notation. The latter, though more general, is less efficient because it incurs the expense of making extra function calls (i.e., those providing the representation-independent "wrapper," as it is sometimes called, which hides the representation-dependent notation inside).
7. The representation-independent notation for operating on nodes of a linked list is less efficient than the particular kinds of representation-dependent notations because using it incurs the extra cost of transmitting procedure parameters and making procedure calls and returns — which are extra expenses not incurred by using the representation-dependent notations.

Answers to Selected Exercises in Exercises 4.5

6. Nothing is wrong with the program. It will work as it is. However, note that it redeclares the variable, `Avail`, to be a string variable local to the function `main( )`, when an identically named variable is

used in the implementation section of the module, ParallelArrayLists. This is acceptable, since the variable, Avail, used inside the implementation file "ListImplementation.c" for the Parallel Array Representation module of Program 4.23 does not interfere with any identically named locally defined variables inside the main( ) procedure.

Answers to Review Questions 4.6

1. Modules are also called *units* or *packages* in other modern languages such as Pascal and Ada.
2. *Procedural abstraction* occurs when one creates a procedure, P(a_1, a_2, ..., a_n), as a named unit of action. Then, one can use the action represented by P, knowing only *what* P does and not *how* P is implemented. Later, one can change how P is implemented without changing every instance of P's use (which could not have happened if the instructions giving P's action had been repeated every time P was used). The use of procedural abstraction therefore promotes ease of use and change.
3. *Data abstraction* occurs when one hides the representation of a data structure and the implementation details of the operations on it using a representation-independent notation. This enables the abstract data to be used knowing only *what* it does, not *how* its details are implemented. As with procedural abstraction, so also with data abstraction, by separating the *what* from the *how*, both ease of use and ease of modification are promoted.
4. *Encapsulation* consists of hiding program entities (or certain features of externally visible program entities) inside a protective wall, called the capsule boundary, so that they can neither be seen nor used outside the capsule.
5. In summary, using C modules properly can provide for: (a) separate compilation (which can reduce compilation times since the whole program doesn't need to be compiled each time), (b) ease of modification, (c) ease of use of the services provided by the module, (d) help with software system design, and (e) help in organizing the work of a software project.