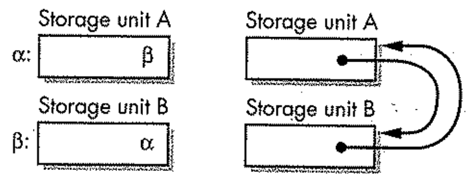


**Figure 2.5** Two Equivalent Notations for Pointers

goes down the window list drawing the contents of each window from top to bottom, making sure to clip the drawing of each window's contents so that it will occur only in its visible region.

### Pointers Defined

When we store the address α of a storage unit A inside another storage unit B, we say that B contains a *pointer* to A, and we say that α is a *pointer*. In Fig. 2.5 we can also say that B *links to* A, or that B contains a *reference* to A. Also, we can say that A is B's "*referent.*" When we follow a pointer to the unit of storage for which the pointer value is the address, we *dereference* the pointer.

A **pointer** is a data value that references a unit of storage.

On the right side of Fig. 2.5, the upward pointing arrow (●⟶) gives an alternative way to represent a pointer to the storage unit A. The tail of the arrow (●— ) represents a stored copy of a data value equal to the address of the storage unit that the tip of the arrow (⟶ ) touches. Thus the left and right sides of Fig. 2.5 depict identical situations.

## 2.3 Pointers in C—The Rudiments

### Learning Objectives

1. To understand how pointers work in C.
2. To be able to declare pointer data types.
3. To know how to allocate and deallocate memory dynamically.
4. To be able to perform operations, such as dereferencing, on pointers.
5. To understand the concepts of aliasing, dangling pointers, and the scope of dynamic storage allocation.

Suppose we want to create some pointers to blocks of storage containing integers. To do this in C, we could first make the following declaration:

```
typedef int *IntegerPointer;
```

pointers to integers

The asterisk (*) in this type definition appears before the type name IntegerPointer to indicate a type consisting of a "pointer to an int." Thus the above declaration says, "define a type consisting of an integer pointer having the name IntegerPointer."

We can now declare two variables A and B to be of a "type" that requires them to contain pointers to integers:

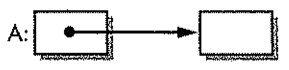
```
IntegerPointer A, B;            /* the declaration, int *A, *B; has the same effect */
```

This means that the C compiler will constrain the values of the variables A and B to be "pointers to integers." There are two aspects of this constraint: (1) A and B have to contain *pointers* to other storage units, and (2) these other storage units, to which the pointer values in A and B refer, are required to contain integers.

dynamic storage allocation

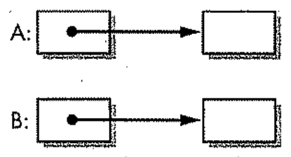
Now, to create some pointers to some empty integer storage units and to make A and B point to them, we use the special C *dynamic storage allocation procedure*, malloc (which is used for dynamic memory allocation).

```
/* Create a new block of storage for an integer, and place a pointer to it in A. */
/* Here (IntegerPointer) type casts the void pointer returned by malloc */
/* into a pointer to a block of storage containing an integer. */

    A = (IntegerPointer) malloc(sizeof(int));
```



A diagram of what is created is given immediately above.

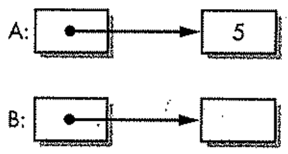
```
/* Then, create another block of storage for an integer, and put a pointer to it in B.*/
/* This time, type cast the void pointer to storage for an int directly with (int *).*/

    B = (int *) malloc(sizeof(int));      /* sizeof(int) == no. of bytes to store an int */
```



A diagram of the newly created storage that results from executing the latter statement after executing the former statement is shown immediately above.

Now let's store the integer 5 in the block of storage referenced by the pointer in variable A. In what follows, we'll continue to give the diagram that portrays the storage structures created immediately after executing each new statement. So to store 5 in the block of integer storage to which the pointer in A refers, we need to execute a special assignment statement:

```
/ * Store the integer 5 in A's referent. */
    *A = 5;
```



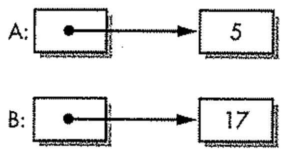


**Figure 2.6** Pointers to Integers

By placing the asterisk (*) before the variable A on the left side of the assignment *A = 5; we designate the storage location *A to which the pointer value in A refers. Similarly, we can store the integer 17 in the block of integer storage to which B's pointer refers:

```
/* Store the integer 17 in B's referent. The result is shown in Fig. 2.6. */
    *B = 17;
```

## Question and Answer Time

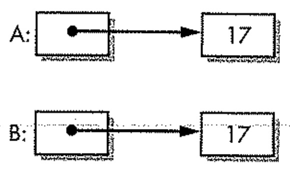
*Question:* Suppose you try to store 17 directly into B by performing the assignment statement B = 17. What happens? *Answer*: The C compiler won't compile the program, since this is a "type mismatch error." The right side of the assignment is a value, 17, of type int, but the left side of the assignment, B, is a variable which is required to contain a pointer to an integer. In C's type system, an int and a pointer to an int are not the same type. Since the types don't match, and since there is no defined automatic conversion to change the type of the right side into a pointer value, C considers B = 17 to be an "illegal pointer arithmetic" in an assignment statement.

type mismatches

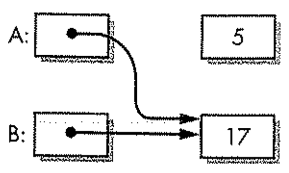
*Question:* Suppose you try to print the value of the pointer stored in B by executing the statement printf("B == 0x%x",B). What happens? *Answer*: In contrast to compilers for some other hard-typed languages, C permits pointer values to be printed. The printf statement just given prints an answer such as B == 0xf6da, which is given in hexadecimal (because the conversion specification %x prints values in hexadecimal).

are pointers invisible?

*Question*: Starting with the situation in Fig. 2.6, which of the following diagrams results if we perform the assignment A = B;—the left diagram or the right diagram?



Left diagram



Right diagram

copying pointers

*Answer*: The right diagram. Why? Because the assignment, A = B, takes a copy of the value of B, which is a *pointer* to the block of storage containing 17, and places this pointer in A, so that A now points to the same block of storage that B points to.

copying integers

*Question*: What assignment statement would we perform if we wanted to create the situation shown in the left diagram, starting with Figure 2.6? *Answer*: The assignment *A = *B; which says to take a copy of the value 17 stored in B's referent, and put it in the location given by A's referent, *A.

## Some Fine Points—Aliases, Recycling, and Dangling Pointers

Now, focus for a moment on the right diagram above, which resulted from performing the assignment A = B on Figure 2.6.

### Aliases

aliases

Note that A and B contain identical pointer values, pointing to the same (shared) storage location, which contains the integer 17. Because A's referent, *A, and B's referent, *B, "name" the same storage location, they are called *aliases*. In general, *aliases* are two different naming expressions that name the same thing. Starting with the right diagram, the effect of performing either the assignment *A = 23; or the assignment, *B = 23; is identical. Namely, the value 23 is put in the location that used to contain 17. Similarly, starting with the right diagram, 19 is the common value of the expressions *A + 2 and *B + 2 because *A and *B are both aliases that designate the same value 17 when used inside expressions.

### Recycling Used Storage

inaccessible storage

Again starting with the right diagram, note that the block of storage containing the integer 5 has no pointer pointing to it. If such a situation should develop, we say that this storage block has become *inaccessible*—meaning that nobody can access it to read its value or to store a new value into it.

It is potentially wasteful to allocate blocks of storage and then abandon them when they become inaccessible, since the pool of storage for allocating new blocks of storage could easily become used up. This could result in a *storage allocation failure*, in which, when we called the function malloc, there would be no more storage to allocate.

storage reclamation

In order to guard against this possibility, we can execute the special C *storage reclamation function*, free(X). This returns the block of storage, to which the pointer in

X points, to the pool of unallocated storage so that it can be used again. (In C, this pool of unallocated storage is often called *dynamic memory*.) Reclaiming storage this way is analogous to recycling used bottles and newspapers.

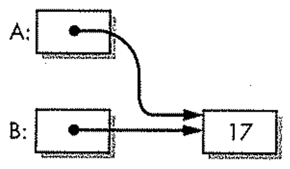
Sometimes, we use the word *garbage* to refer to inaccessible storage. Some programming languages, such as the list-processing language, LISP, have procedures for storage reclamation called garbage collection procedures that are triggered when available unallocated storage is used up. These garbage collection procedures have a way of identifying which blocks of storage are inaccessible and of returning them to the pool of unallocated storage. C does not have such an automatic garbage collection procedure, however, so the recycling of storage must be explicitly managed by the programmer who writes C programs.

garbage collection

If we want to recycle the storage block containing 5 in Fig. 2.6, before destroying the pointer to it in A (by overwriting this pointer with a copy of B's pointer), we could first perform the function call, free(A), as in the following sequence of actions:

disposing of unused storage

```
/* First, dispose of A's referent. Then, replace A with B.*/

      free(A);                  /* First, recycle the storage block to which A refers.*/
      A = B;                    /* Then, put a copy of B's pointer in A.*/
```



Resulting situation

## Dangling Pointers

Once we have *aliases* pointing to the same storage location, a new *programming danger* arises. Starting with the resulting situation immediately above, suppose we called the function free(B). What would happen then, if we tried to evaluate the expression *A + 2? We don't know, but, since *A might no longer contain a valid integer value, the result could be unexpected.

First, the call to free(B) would return the storage location containing 17, which is B's referent, to the pool of available storage. (When it does this, it might erase the value contained in the block, or join the block to another block of storage to make a bigger block—just to name two of several possibilities.) Now, A contains a pointer, called a *dangling pointer*, which points to the same location that B's pointer used to point to. This dangling pointer might now point into the middle of a large block, formed when free(B) caused B's referent block to join another block in the storage reclamation process. The possibility exists that the referent of the dangling pointer doesn't even have a meaningful existence anymore in C. It is reasonable to consider this to be a programming error, even though neither the C compiler nor the running

dangling pointers

C system may have any means of detecting it. Consequently, the creation of dangling pointers is a pitfall that should be avoided.

### The Lifetime of Dynamic Storage

anonymous variables

The blocks of storage that pointers refer to are created dynamically during the running of the program and can be thought of as *anonymous variables*.[2] Such variables have no explicit textual names in the program, and their lifetimes exist beyond the lifetimes of local variables in the program. For example, even though an anonymous variable might be created by a function having its own local variables that vanish after the function has terminated execution, the pointer values and the locations they reference nonetheless have an existence outside the (temporary) existence of the function's local storage. One way to think of this is that dynamic storage allocated for anonymous variables by malloc exists from the moment it is allocated until the moment when its storage is reclaimed, using free(X), or until the program's execution terminates (if its storage is not reclaimed previously, using free(X)).

### Dereferencing

When the asterisk (*) precedes the name of a variable, as in *A, we say that the pointer value of A is *dereferenced*. Thus dereferencing is a fancy name for *pointer following*. When the asterisk (*) precedes a new type name identifier T in a type definition, it defines a *pointer type*. For example,

```
typedef T *TPointer;
```

pointer following

means that the type TPointer is the set of pointers to storage units containing values of type T. Thus, the asterisk precedes pointer variables when they are dereferenced and precedes type name identifiers when they are defined. When a dereferenced variable, *V, occurs on the left side of an assignment, *V = X, it signifies that a copy of the value in X is to be stored in the location to which the pointer value in V points..

## 2.3 Review Questions

1. What does it mean to *dereference a pointer*? If A is a pointer variable in C, what is the notation for dereferencing the pointer value stored in A?

[2] The word *anonymous* means "having no name." For example, anonymous donors to a charity are donors whose names are not revealed. Thus an *anonymous variable* is a variable that has no name but acts like a variable in all other respects. If an ordinary variable is defined to be a named storage location that can contain values that can be varied (i.e., changed) over time, then an *anonymous variable* is an unnamed storage location that can contain values that can be varied over time.

2. What are some other ways of describing the situation in which a unit of storage B contains a pointer to a unit of storage A?
3. How would you declare a data type in C called **IntegerPointer** to be a type consisting of pointers to integers? Give the C syntax for the type definition needed to accomplish this.
4. Suppose A is a variable declared in C to be a pointer variable containing pointers to storage of type T. How do you dynamically allocate a new block of storage of type T and obtain a pointer to it in C?
5. Suppose A has been declared to be an **IntegerPointer** variable. How would you write an assignment statement to assign the integer 19 to be the value of A's referent? How would you write an expression whose value is twice the value of A's referent?
6. Suppose A has been declared to be an **IntegerPointer** variable. What is wrong with the assignment A = 5; for assigning 5 to be the value of A's referent?
7. What are aliases?
8. How do you recycle dynamically allocated storage when it is no longer needed so that its space can be reused to meet further storage allocation needs?
9. How can dynamically allocated storage become inaccessible in C?
10. What is garbage?
11. What is a dangling pointer?
12. What is the lifetime of dynamically allocated storage?
13. What is an anonymous variable?

## 2.3 Exercises

For the following problems, assume that the following types and variables have been declared in C, and are available for use:

```
typedef int *IntegerPointer;

IntegerPointer A , B;
```

1. What does the following procedure print when it is executed?

```
void P(void)
{
     A = (IntegerPointer)malloc(sizeof(int));
     B = (IntegerPointer)malloc(sizeof(int))
     *A = 19;
     *B = 5;
     A = B;
     *B = 7;
     printf("%d\n",*A);
}
```

2. What does the following procedure print when it is executed?

```
void P(void)
{
     A = (int *)malloc(sizeof(int));
     B = (int *)malloc(sizeof(int))
     *A = 19;
     *B = 5;
     *A = *B;
     *B = 7;
     printf("%d\n",*A);
}
```

3. How does storage allocation failure manifest itself in your C system? If your instructor will authorize you to do so, try running the following program. What does it tell you if it terminates and prints its final message properly?

```
void AllocationFailureTest(void)
{
     typedef int BigArray[1000];
     BigArray *A;

     while ((A = (BigArray *)malloc(sizeof(BigArray))) != NULL)
          ;

     printf("storage allocation failure occurred");
}
```

*Caution:* Don't try executing this procedure in your C system unless you know in advance how to interrupt a running program, how to terminate its execution, and how to regain control of events. It is possible that the above program will result in an endless loop, if allocation failure does not result in setting A to NULL, when malloc(sizeof(BigArray)) is called with insufficient storage available to satisfy the request. As a precaution, you can rewrite the program above to count the number of times malloc(sizeof(BigArray)) is called, and to pause and ask you if you want to continue, say, every 5000 times it is called.

4. What happens in your C system if you try to dereference the null pointer, as in the following C fragment?

```
A = NULL;          /* set A to NULL */
A = *A;            /* dereference the pointer NULL */
```

*Caution:* Don't try to execute this on your computer unless your instructor authorizes you to do so, and you know how to recover from crashes.

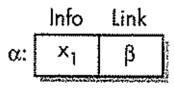
## 2.4 Pointer Diagramming Notation

**Learning Objectives**

1. To understand how to interpret the pointer diagrams used in this book.
2. To learn a notation for picturing linked representations that is helpful when reasoning about them.

Diagrams illustrating pointers in this book follow a few simple conventions. The boxes represent different units of addressable storage. We will refer to these boxes as *nodes*. By convention in these diagrams, we assume that different nodes represent storage locations with different addresses. The three rows in Fig. 2.7 represent the same linked list. When addresses are represented explicitly, they are signified by Greek letters, such as α, β, and γ, as shown in the topmost linked list of Fig. 2.7. For instance, α is the address of the following node:

nodes



It is written to the left of the node, followed by a colon. β signifies an address stored as a pointer value in the Link field of this node. If a node is divided into fields (representing a C structure with members), the names of the fields are attached to the border of the box representing the node. Thus the node above has two fields: an Info field, containing the value $x_1$, and a Link field containing the pointer β.

fields

There exists a special *null* address, which by convention is not the address of any node. It is depicted by a solid dot (•) that represents an arrow with no tip that points

the null address

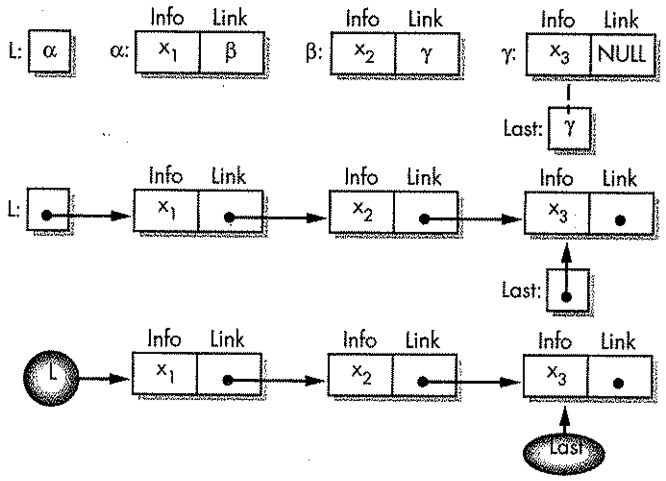
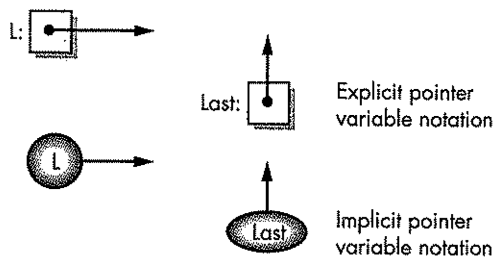


**Figure 2.7** Three Equivalent Diagrams in Pointer Diagramming Notation

nowhere. In C, the null pointer is represented by the special pointer value NULL. This special null address is used to show where a linked list ends, and it is used to represent an empty structure (such as a linked list having no nodes).

Finally, Fig. 2.7 shows some examples of the diagramming notation for the pointer variables L and Last. In the diagram below, the two notations for L and Last are equivalent.


L:
Last:
Explicit pointer variable notation
L
Last
Implicit pointer variable notation

pointer variable notation

Pointer variables can be written two ways: either (1) *explicitly* as a box containing a pointer value labeled on the left side with the name of the variable followed by a colon, or (2) *implicitly* as an oval (or circle) containing the variable name and pointing to the node the pointer value references.

We can use a pointer to NULL to represent the null pointer in diagrams where we want to show explicitly how the null pointer is replaced by a non-null pointer value. For instance, Fig. 2.8 shows how we can insert a new second node into a linked list by changing the link of the first node to point to it, and by replacing its null link with a pointer to the previous second node. The arrows that are "crossed out" represent pointer values that were replaced with new pointer values during the operation shown in the diagram.

using implicit pointer variable notation

We will use the implicit notation for pointer variables (as ovals containing variable names) in diagrams with many nodes in which we use pointer variables to contain pointers to point at places in the overall linked data structure (1) where change is about to occur or (2) to keep track of a location we will need to use in the future. The implicit notation tends to be more compact and less cluttered than the explicit notation in such circumstances, but we should always remember that it is equivalent to the explicit notation in which the variable name, followed by a colon, is to the left of a box containing a pointer.

unknown values

Finally, when the value of a variable or the value of a field in a node is unknown (as happens sometimes at the outset of an algorithm before initialization), we may symbolize the unknown value by a question mark (?) for nonpointer values, or a

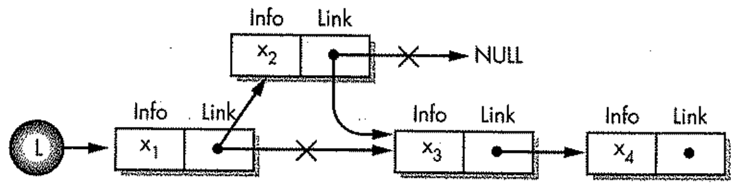


**Figure 2.8** Inserting a New Second Node into a Linked List

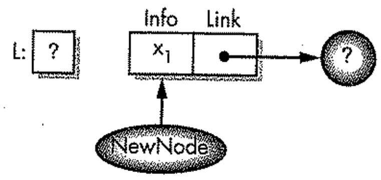


**Figure 2.9** Unknown Values and Pointers

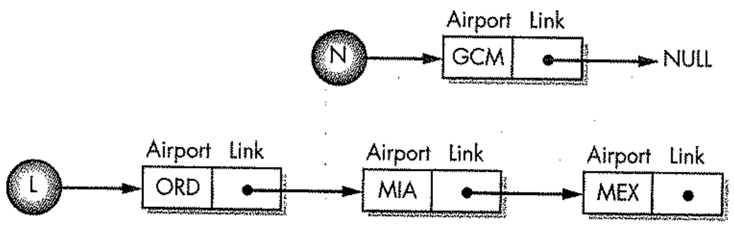
pointer to a circle containing a question mark, for an unknown pointer value, as in Fig. 2.9. We will use these explicit denotations of unknown values in situations where we want to emphasize that a newly allocated structure needs to be initialized with known values, and where we need something to "cross out" to indicate the replacement of an old value with a new value.

## 2.4 Review Questions

1. What is the null address?
2. How is null address depicted in pointer diagramming notation?
3. What value represents the null address in C?
4. How is the end of a linked list indicated in pointer diagramming notation?
5. What is an empty linked list, and what value is used to indicate it?
6. Explain the difference between explicit pointer variable notation and implicit pointer variable notation.
7. How is an unknown value signified in pointer diagramming notation?

## 2.4 Exercises

[*Note:* The exercises below introduce an important new –> notation.] For these exercises assume that the following diagram specifies the values of the variables N and L:



(*Note:* Here, the three-letter codes ORD, MIA, GCM, and MEX are the codes airlines use on baggage tags: ORD = Chicago's O'Hare airport; MIA = Miami, Florida; MEX = Mexico City, Mexico; and GCM = Grand Cayman in the Cayman Islands.)

1. In the diagram above, if you assume L's referent, *L, is a node containing a struct with an Airport field containing ORD, then you could write: (*L).Airport == ORD. Also, given that L–>Airport is a preferred equivalent notation for (*L).Airport, you could conclude that L–>Link–>Airport == MIA. Using this line of reasoning, what

are the values of the following four struct field selection expressions: (a) N–>Link, (b) N–>Airport, (c) L–>Link–>Link–>Airport, and (d) L–>Link–>Link–>Link?

2. Write C expressions possibly using: (a) the variables L and N, (b) the struct member names Airport and Link, (c) the dereferencing operator (*), the struct member selection operator (.) or the operator (–>), having the following values: (i) a pointer to the node containing MIA, (ii) a pointer to the node containing MEX, (iii) a pointer to the node containing ORD, and (iv) the node containing ORD.
3. Write two C assignment statements that (when executed) will insert N's referent as the new third node of the list L. Before these assignments are executed, the airport codes of the nodes in the list referenced by L are ORD, MIA, MEX. After the assignments are executed, the airport codes should be in the sequence ORD, MIA, GCM, MEX.
4. Draw the picture of the result of executing the two assignment statements that solve problem 3, starting with the diagram for the values of L and N above.
5. In C, executing the string copy function strcpy(L–>Airport,"JFK") starting with the diagram above, replaces the airport code ORD by JFK. Write a function call using strcpy that replaces MIA by JFK. (*Note:* JFK is the airport code for John F. Kennedy International Airport in New York City.)
6. Write statements in C that replace the node containing MIA with the node containing GCM. After execution, L should point to nodes with airport codes given in the sequence ORD, GCM, MEX. Remember to free the storage for the node containing MIA.

## 2.5 Linear Linked Lists

**Learning Objectives**

1. To understand how to represent and manipulate linked lists using C's pointer data types.
2. To develop skill in designing and implementing linked list algorithms.
3. To learn a set of useful basic linked list operations.

linear linked lists

A *linear linked list* (or a *linked list*, for short) is a sequence of nodes in which each node, except the last, links to a successor node.

The link field of the last node contains the null pointer value, NULL, which signifies the end of the list. Figure 2.10 represents a linear linked list containing the three-letter airport codes for the stops on American Airlines flight number 89. This flight starts in Düsseldorf, Germany (DÜS), it stops at Chicago's O'Hare Airport (ORD), and it continues on to its final destination at San Diego (SAN).

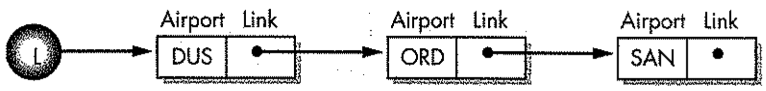


**Figure 2.10** A Linear Linked List