Dynamic Memory Management

Introduction

- **We've studied fixed-size data structures such as single-subscripted arrays, double-subscripted arrays and `structs`.**
- **This Lesson introduces dynamic data structures with sizes that grow and shrink at execution time.**
  - **Linked lists are collections of data items "lined up in a row"—insertions and deletions are made *anywhere* in a linked list.**
  - **Stacks are important in compilers and operating systems—insertions and deletions are made *only at one end* of a stack—its top.**

Introduction (Cont.)

- **Queues represent waiting lines; insertions are made *only at the back* (also referred to as the tail) of a queue and deletions are made *only from the front* (also referred to as the head) of a queue.**
- **Binary trees facilitate high-speed searching and sorting of data, efficient elimination of duplicate data items, representing file system directories and compiling expressions into machine language.**

- **Each of these data structures has many other interesting applications.**

Introduction (Cont.)

- **We'll discuss each of the major types of data structures and implement programs that create and manipulate them.**
- **This technique will enable us to build these data structures in a dramatically different manner designed for producing software that's much easier to maintain and reuse.**

Self-Referential Structures

- Recall that a *self-referential structure* contains a pointer member that points to a structure of the *same* structure type.
- For example, the definition

```
struct node {
    int data;
    struct node *nextPtr;
}; // end struct node
```

  defines a type, `struct node`.
- A structure of type `struct node` has two members—integer member `data` and pointer member `nextPtr`.

Self-Referential Structures (Cont.)

- Member `nextPtr` points to a structure of type `struct node`—a structure of the *same* type as the one being declared here, hence the term "*self-referential structure*."
- Member `nextPtr` is referred to as a link—i.e., it can be used to "tie" a structure of type `struct node` to another structure of the same type.
- Self-referential structures can be *linked* together to form useful data structures such as lists, queues, stacks and trees.

Self-Referential Structures (Cont.)

- **Figure 12.1 illustrates two self-referential structure objects linked together to form a list.**
- **A slash—representing a NULL pointer—is placed in the link member of the second self-referential structure to indicate that the link does not point to another structure.**
- **[*Note*: The slash is only for illustration purposes; it does not correspond to the backslash character in C.]**
- **A NULL pointer normally indicates the end of a data structure just as the null character indicates the end of a string.**

Common Programming Error 12.1

Not setting the link in the last node of a list to NULL can lead to runtime errors.

Fig. 12.1 | Self-referential structures linked together.

Dynamic Memory Allocation

- **Creating and maintaining dynamic data structures requires dynamic memory allocation—the ability for a program to *obtain more memory space at execution time* to hold new nodes, and to *release space no longer needed*.**
- **Functions `malloc` and `free`, and operator `sizeof`, are essential to dynamic memory allocation.**

Dynamic Memory Allocation (Cont.)

- Function `malloc` takes as an argument the number of bytes to be allocated and returns a pointer of type `void *` (pointer to void) to the allocated memory.
- As you recall, a `void *` pointer may be assigned to a variable of *any* pointer type.
- Function `malloc` is normally used with the `sizeof` operator.

Dynamic Memory Allocation (Cont.)

- For example, the statement

```
newPtr = malloc( sizeof( struct node ) );
```

  evaluates `sizeof(struct node)` to determine the size in bytes of a structure of type `struct node`, *allocates a new area in memory* of that number of bytes and stores a pointer to the allocated memory in variable `newPtr`.
- The allocated memory is *not* initialized.
- If no memory is available, `malloc` returns `NULL`.

Dynamic Memory Allocation (Cont.)

- Function `free` *deallocates* memory—i.e., the memory is *returned* to the system so that it can be reallocated in the future.
- To *free* memory dynamically allocated by the preceding `malloc` call, use the statement
  - `free( newPtr );`
- C also provides functions `calloc` and `realloc` for creating and modifying *dynamic arrays*.

Portability Tip 12.1

A structure's size is not necessarily the sum of the sizes of its members. This is so because of various machine-dependent boundary alignment requirements (see Chapter 10).

Error-Prevention Tip 12.1

When using `malloc`, test for a `NULL` pointer return value, which indicates that the memory was not allocated.

Common Programming Error 12.2

Not returning dynamically allocated memory when it's no longer needed can cause the system to run out of memory prematurely. This is sometimes called a "memory leak."

Error-Prevention Tip 12.2

When memory that was dynamically allocated is no longer needed, use `free` to return the memory to the system immediately. Then set the pointer to `NULL` to eliminate the possibility that the program could refer to memory that's been reclaimed and which may have already been allocated for another purpose.

Common Programming Error 12.3

Freeing memory not allocated dynamically with `malloc` is an error.

Common Programming Error 12.4

Referring to memory that has been freed is an error that typically results in the program crashing.