see precisely what is happening to the word in memory. Because machines vary on whether bits and bytes are counted from the high-order or low-order end of a word, the use of a utility such as bit_print() is essential. On one machine, our program caused the following to be printed:

```
w.i = 353
00000000 00000000 00000001 01100001
```

whereas on another machine we obtained

```
w.i = 1635778560
01100001 10000000 00000000 00000000
```

Because machines vary with respect to word size and with respect to how bits and bytes are counted, code that uses bit fields may not be portable.

9.10 The ADT Stack

The term *abstract data type* (ADT) is used in computer science to mean a data structure together with its operations, without specifying an implementation. Suppose we wanted a new integer type, one that could hold arbitrarily large values. The new integer type together with its arithmetic operations is an ADT. It is up to each individual system to determine how the values of integer data types are represented and manipulated computationally. Native types such as char, int, and double are implemented by the C compiler.

Programmer-defined types are frequently implemented with structures. In this section, we develop and implement the ADT *stack*, one of the most useful standard data structures. A stack is a data structure that allows insertion and deletion of data to occur only at a single restricted element, the top of the stack. This is the *last-in-first-out* (LIFO) discipline. Conceptually, a stack behaves like a pile of trays that pops up or is pushed down when trays are removed or added. The typical operations that can be used with a stack are *push*, *pop*, *top*, *empty*, *full*, and *reset*. The push operator places a value on the stack. The pop operator retrieves and deletes a value off the stack. The top operator returns the top value from the stack. The empty operator tests if th*e stack is empty. The full operator tests if the stack is full. The reset operator clears the stack, or initializes it. The stack, along with these operations, is a typical ADT.

We will use a fixed-length char array to store the contents of the stack. (Other implementation choices are possible; in Section 10.5, “Stacks,” on page 461, we will implement a stack as a linked list.) The top of the stack will be an integer-valued member named top. The various stack operations will be implemented as functions, each of whose parameter lists includes a parameter of type pointer to stack. By using a pointer, we avoid copying a potentially large stack to perform a simple operation.

In file stack.c

```
/* An implementation of type stack. */

#define   MAX_LEN    1000
#define   EMPTY      –1
#define   FULL       (MAX_LEN – 1)

typedef   enum boolean {false, true}   boolean;

typedef struct stack {
  char   s[MAX_LEN];
  int    top;
} stack;

void reset(stack *stk)
{
  stk –> top = EMPTY;
}

void push(char c, stack *stk)
{
  stk –> top++;
  stk –> s[stk –> top] = c;
}

char pop(stack *stk)
{
  return (stk –> s[stk –> top—]);
}

char top(const stack *stk)
{
  return (stk –> s[stk –> top]);
}

boolean empty(const stack *stk)
{
  return ((boolean) (stk –> top == EMPTY));
}

boolean full(const stack *stk)
{
  return ((boolean) (stk –> top == FULL));
}
```

Dissection of the *stack* Implementation

§ typedef enum boolean {false, true} boolean;

We use this typedef to give a new name, boolean, to the enumeration type, enum boolean. Note that the new name coincides with the tag name. Programmers often do this.

§
```
typedef struct stack {
  char   s[MAX_LEN];
  int    top;
} stack;
```

This code declares the structure type struct stack, and at the same time uses a typedef to give struct stack the new name stack. Here is an equivalent way to write this:

```
struct stack {
  char   s[MAX_LEN];
  int    top;
};

typedef   struct stack   stack;
```

The structure has two members, the array member s and the int member top.

§
```
void reset(stack *stk)
{
  stk –> top = EMPTY;
}
```

The member top in the stack pointed to by stk is assigned the value EMPTY. This conceptually resets the stack, making it empty. In the calling environment, if st is a stack, we can write

```
reset(&st);
```

to reset st or initialize it. At the beginning of a program, we usually start with an empty stack.

```
§   void push(char c, stack *stk)
    {
      stk –> top++;
      stk –> s[stk –> top] = c;
    }

§   char pop(stack* stk)
    {
      return (stk –> s[stk –> top—]);
    }
```

The operation *push* is implemented as a function of two arguments. First, the member top is incremented. Note that

```
stk –> top++    is equivalent to    (stk –> top)++
```

Then the value of c is shoved onto the top of the stack. This function assumes that the stack is not full. The operation *pop* is implemented in like fashion. It assumes the stack is not empty. The value of the expression

```
stk –> top—
```

is the value that is currently stored in the member top. Suppose this value is 7. Then

```
stk –> s[7]
```

gets returned, and the stored value of top in memory gets decremented, making its value 6.

```
§   boolean empty(const stack *stk)
    {
      return ((boolean) (stk –> top == EMPTY));
    }

§   boolean full(const stack *stk)
    {
      return ((boolean) (stk –> top == FULL));
    }
```

Each of these functions tests the stack member top for an appropriate condition and returns a value of type boolean, either true or false. Suppose the expression

```
stk –> top == EMPTY
```

in the body of empty() is true. Then the expression has the int value 1. This value gets cast to the type bool-

ean, making it true, and that is what gets returned.

To test our stack implementation, we can put the preceding code in a *.c* file and add the following code at the bottom. Our function main() enters the characters of a string onto a stack and then pops them, printing each character out in turn. The effect is to print in reverse order the characters that were pushed onto the stack.

In file stack.c

```
/* Test the stack implementation by reversing a string. */

#include <stdio.h>

int main(void)
{
  char    str[] = "My name is Laura Pohl!";
  int     i;
  stack   s;
  reset(&s);                    /* initialize the stack */
  printf(" In the string: %s\n", str);
  for (i = 0; str[i] != '\0'; ++i)
    if (!full(&s))
     push(str[i], &s);          /* push a char on the stack */
  printf("From the stack: ");
  while (!empty(&s))
    putchar(pop(&s));           /* pop a char off the stack */
  putchar('\n');
  return 0;
}
```

The output from this test program is

```
 In the string: My name is Laura Pohl!
From the stack: !lhoP aruaL si eman yM
```

Note that the expression &s, the address of the stack variable s, is used as an argument whenever we call a stack function. Because each of these functions expects a pointer of type stack *, the expression &s is appropriate.

Summary

1 Structures and unions are principal methods by which the programmer can define new types.

2 The typedef facility can be used to give new names to types. Programmers routinely use typedef to give a new name to a structure or union type.

3 A structure is an aggregation of components that can be treated as a single variable. The components of the structure are called members.

4 Members of structures can be accessed by using the member access operator ".". If s is a structure variable with a member named m, then the expression s.m refers to the value of the member m within the structure s.

5 Members of structures can also be accessed by the member access operator –>. If p is a pointer that has been assigned the value &s, then the expression p –> m also refers to s.m. Both "." and –> have highest precedence among C operators.

6 In ANSI C, if a and b are two variables of the same structure type, then the assignment expression a = b is valid. It causes each member of a to be assigned the value of the corresponding member of b. Also, a structure expression can be passed as an argument to a function and returned from a function. (Many traditional C compilers also have these capabilities, but not all of them.)

7 When a structure variable is passed as an argument to a function, it is passed "call-by-value." If the structure has many members, or members that are large arrays, this may be an inefficient way of getting the job done. If we redesign the function definition so that a pointer to the structure instead of the structure itself is used, then a local copy of the structure will not be created.

8 A union is like a structure, except that the members of a union share the same space in memory. Unions are used principally to conserve memory. The space allocated for a union variable can be used to hold a variety of types, specified by the members of the union. It is the programmer's responsibility to know which representation is currently stored in a union variable.

9 The members of structures and unions can be arrays or other structures and unions. Considerable complexity is possible when nesting arrays, structures, and unions within each other. Care must be taken that the proper variables are being accessed.

10 A bit field is an int or unsigned member of a structure or union that has a specified number of bits. The number of bits is given as an integral constant expression following a colon. This number is

called the width of the bit field. The width is limited to the number of bits in a machine word. Consecutive bit fields in a structure are stored typically as contiguous bits in a machine word, provided that they fit.

11 Bit fields can be unnamed, in which case they are used for padding or word alignment purposes. The unnamed bit field of width 0 is special. It causes immediate alignment on the next word.

12 How bit fields are implemented is system-dependent. Hence, their use need not be portable. Nonetheless, bit fields have important uses.

13 The stack is an abstract data type (ADT) that has many uses, especially in computer science. An ADT can be implemented many different ways.

Exercises

1 In some situations a typedef can be replaced by a #define. Here is an example:

```
typedef   float   DOLLARS;

int main(void)
{
  DOLLARS   amount = 100.0, interest = 0.07 * amount;

  printf("DOLLARS = %.2f\n", amount + interest);
  return 0;
}
```

Execute this program so you understand its effects, then replace the typedef by

```
#define   DOLLARS   float
```

When you recompile the program and execute it, does it behave as it did before?