DATA STRUCTURE THROUGH C LANGUAGE

Typedef

In most C projects, the typedef statement is used to create equivalence names for other C types, particularly for structures and pointers, but potentially for any type. Using typedef equivalence names is a good way to hide implementation details. It also makes your code more readable, and improves the overall portability of your product.

Typedef and Portability

Typedef is frequently used to improve code portability. To cite a simple example, suppose you had a need to declare a data structure that was guaranteed to occupy the same amount of memory on every platform. If this structure had integer fields you might be tempted to declare them to be either short or long, believing that these types would always translate to two- or four-byte integers, respectively. Unfortunately ANSI C makes no such guarantee. Specifically, if you declare a field to be type long, it will break when you try to port your code to an Alpha running Digital UNIX, where an int is four bytes, and a long is eight bytes. To avoid this problem you can use typedef in conjunction with conditional compilation directives to declare integer equivalence types; on most platforms the equivalence type for a four-byte field will be long, but under Alpha/Digital UNIX it will be int:

```
 #if defined( ALPHA ) && defined ( DIGITAL_UNIX )

typedef int BYTE4_t;

#else typedef long BYTE4_t;

#endif
```

For the record, a variable or field that needed to be exactly four bytes will then be declared like this:

```
BYTE4_t field_name;
```

Typedef and Structures

To create a structure variable suitable for describing a street address, and to pass the variable to a routine that could print it, you might use code like that shown in Figure 1-1, which explicitly declares all structure components using the struct keyword. Normally, however, you would declare equivalence names for the address structure using typedef, and then declare structure variables and parameters using the equivalence names, as shown in Figure 1-2. Note that, in this example, one typedef statement was used to create two equivalence names: ADDRESS_t, which is equivalent to struct address_s, and ADDRESS_p_t, which is equivalent to struct address_s*. This is a very common technique in modern C usage.

```
struct address_s
```

```
{ char *street;
char *city;
char *region;
char *country;
char *postal_code;
};
static void print_address(
struct address_s *address_info
 );
static void print_an_address( void )
{
struct address_s address;
address.street = “1823 23rd Ave NE”;
address.city = “Seattle”;
address.region = “WA”;
address.postal_code = “98023”;
print_address( &address );
}
```

Typedef and Functions

Recall that the type of a function is a combination of the function’s return type, and the number and type of each of the function’s parameters. Typedef can be used to declare an equivalence name for a function type, and, subsequently, the equivalence name can be used to declare a prototype for a function of that type. One use of this is to reduce repetition when declaring many functions of the same type. For example, if you are implementing a standard sort algorithm, you will need to define a prototype for a compare function; specifically, a function used to determine whether one object is greater than another. A prototype for such a function would traditionally be declared like this:

```
typedef struct address_s
{
char *street;
char *city;

char *region;

char *country;

char *postal_code;

} ADDRESS_t, *ADDRESS_p_t;

static void print_address(

ADDRESS_p_t address_info );

static void print_an_address( void )

{

ADDRESS_t address;

address.street = “1823 23rd Ave NE”;

address.city = “Seattle”;

address.region = “WA”;

address.postal_code = “98023”;

print_address( &address );

}
```

Pointers and Arrays

In C, pointers and arrays are very closely related. With only two exceptions, the name of an array is equivalent to a pointer to the first element of the array. The first parameter of the sort_dates function is declared to be pointer to date structure; the actual call to sort_dates passes the name of an array of date_structures as the corresponding argument, and the C compiler obligingly converts the name to a pointer. Since a C subroutine can’t determine the length, or cardinality, of an array passed as an argument, the second argument to sort_dates specifies the length of the array. The two exceptions are that an array name cannot be an lvalue (it cannot appear on the left side of the equal sign in an assignment); and that C does not treat the name as a pointer when used as the argument of the size of operator. If you use the name of an array as an argument of size of, the size of the entire array is computed. This allows us to create a very convenient macro which I have called CARD (short for cardinality); this takes the size of the array divided by the size of the first element of the array, which yields the total number of elements in the array.

Dynamic Memory Allocation

Dynamic memory allocation in C is performed using one of the standard library functions malloc, calloc or realloc (or a cover for one of these routines). Dynamically allocated memory must eventually be freed by calling free. If allocated memory is not properly freed a memory leak results, which could result in a program malfunction.

```
typedef struct date_s

{

short year;

char month;

char day;

} DATE_t, *DATE_p_t;

static void sort_dates( DATE_p_t dates, int num_dates ); DATE_t
dates[4] = { {1066, 3, 27},

                    {1941, 12, 1},

                    {1492, 10, 12},

                    {1815, 10, 14}

                    };

. . .

sort_dates( dates, 4 );
```

The ability to dynamically allocate memory accounts for much of the power of C. It also accounts for much of the complexity of many C programs, and, subsequently, is the source of many of their problems. One of the biggest problems associated with dynamically allocated memory comes from trying to deal with allocation failure. Many organizations effectively short-circuit this problem by writing cover routines for the dynamic memory allocation routines that do not return when an error occurs; instead, they abort the program.

```
#define CARD( arr ) (sizeof((arr))/sizeof(*(arr)))

. . .

sort_dates( dates, CARD( dates ) );
```

Doubly Linked Lists

In this section we will examine one of the most common and versatile data structures in data processing: the doubly linked list. In designing the VAX, Digital Equipment Corporation engineers felt that this type of list was so important that they designed machine-level instructions for manipulating them.

In addition to learning about doubly linked lists, in this lesson you will begin to learn how to formally define data structures, and to encapsulate data structure functionality in modules.

Methods

In this class we will define the following operations that may be performed on a doubly linked list, and the elements that belong to a doubly linked list: •Create a new doubly linked list

•Create a new enqueuable item

•Test whether an item is enqueued

•Test whether a list is empty

•Add an item to the head of a list

•Add an item to the tail of a list

•Add an item after a previously enqueued item

•Add an item before a previously enqueued item

•Dequeue an item from a list

•Dequeue the item at the head of a list

•Dequeue the item at the tail of a list

•Get the item at the head of a list (without dequeing it)

•Get the item at the tail of a list

•Given an item, get the following item

•Given an item, get the previous item

•Get the name of a list

•Get the name of an item

•Destroy an item

•Empty a list

•Destroy a list

Sorting

In this section we will discuss sorting algorithms in general, and three sorting algorithms in detail: selection sort, bubble sort and merge sort.

Objectives

At the conclusion of this section, and with the successful completion of your third project, you will have demonstrated the ability to:

•Define the differences between the selection sort, bubble sort and merge sort sorting algorithms; and

•Implement a traditional merge sort algorithm.

Bubble Sort

The bubble sort algorithm moves sequentially through a list of data structures dividing it into a sorted part, and an unsorted part. A bubble sort starts at end of the list, compares adjacent elements, and swaps them if the right-hand element (the element later in the list) is less than the left-hand element (the element with earlier in the list). With successive passes through the list, each member percolates or bubbles to its ordered position at the beginning of the list. The pseudo code looks like this:

```
numElements = number of structures to be sorted

for ( inx = 0 ; inx < numElements - 1 ; ++inx )

     for ( jnx = numElements - 1 ; jnx != inx ; --jnx )

          if ( element( jnx ) < element( jnx - 1 ) )

               swap( element( jnx ), element( jnx - 1 ) )
```

Select Sort

A selection sort looks a lot like a bubble sort. It moves iteratively through a list of data structures, dividing the list into a sorted and an unsorted part. The pseudocode looks like this:

```
numElements = number of structures to be sorted

for ( inx = 0 ; inx < numElements - 1 ; ++inx ) least = inx

     for ( jnx = inx + 1 ; jnx < numElements ; ++jnx )

          if ( element( least ) > element( jnx ) ) least = jnx

               swap( element( least ), element( inx ) )
```

The big difference between a select sort and a bubble sort is the efficiency it introduces by reducing the number of swaps that must be performed on each pass through the list. As you can see, instead of performing a swap each time it determines that an element is out of order, it merely keeps track of the position of the smallest element it has found so far; then, at the end of each iteration, it performs exactly one swap, installing the smallest element in the correct position in the list.

Mergesort

The main idea behind the merge sort algorithm is to recursively divide a data structure in half, sort each half independently, and then merge the results. Since the data structure needs to be split, this algorithm works best with well-organized, contiguous structures such as arrays. To mergesort an array, divide the array in half, and independently sort the two halves. When each half has been sorted, merge the results into a temporary buffer, then copy the contents of the buffer back to the original array. Here is the pseudocode for sorting an array:

```
mergesort( array, numElements )
if ( numElements > 1 )
lowHalf = numElements / 2
highHalf = numElements - lowHalf
array2 = array + lowHalf
mergesort( array, lowHalf )
mergesort( array2, highHalf )
inx = jnx = knx = 0
     while ( inx < lowHalf && jnx < highHalf )
          if ( array[inx] < array2[jnx] ) tempArray[knx++] = array[inx++]
          else tempArray[knx++] = array2[jnx++]
```