The related() function calls set_ancestry() twice in the return statement to test all possibilities of relationship. The return value will be true if either of the calls to set_ancestry() return the value true. A calling program can use the return value from related() to determine whether a pointer has been filled in.

■ **Note** Because you use the library function strcmp() here, you must add an #include directive for <string.h> at the beginning of the program. You'll also need an #include directive for the <stdbool.h> header because you use the bool type and the values true and false.

You now need to add some code to the main() function that you created in Program 11.6 to use the function related() to fill in all the pointers in all the structures where valid addresses can be found. You can insert the following code into main() directly after the loop that inputs all the initial data:

```
current = first;

while(current->next != NULL) /* Check for relation for each person in */
{                            /* the list up to second to last         */
  int parents = 0;        /* Declare parent count local to this block */
  last = current->next;   /* Get the pointer to the next              */

  while(last != NULL)     /* This loop tests current person           */
  {                       /* against all the remainder in the list    */
    if(related(current, last))          /* Found a parent ?           */
      if(++parents == 2)     /* Yes, update count and check it        */
        break;               /* Exit inner loop if both parents found */

    last = last->next;       /* Get the address of the next           */
  }
  current = current->next;   /* Next in the list to check             */
}

/* Now tell them what we know etc. */
/* rest of output code etc.  ...   */
```

This is a relatively self-contained block of code to fill in the parent pointers where possible. Starting with the first structure, a check is made with each of the succeeding structures to see if a parent relationship exists. The checking stops for a given structure if two parents have been found (which would have filled in both pointers) or the end of the list is reached.

Of necessity, some structures will have pointers where the values can't be updated. Because you don't have an infinite list, and barring some very strange family history, there will always be someone whose parent records aren't included. The process will take each of the structures in the list in turn and check it against all the following structures to see if they're related, at least up to the point where the two parents have been discovered.

Of course, you also need to insert prototypes for the functions related() and set_ancestry() at the beginning of the program, immediately after the prototype for the function get_person(). These prototypes would look like this:

```
bool related(struct Family *pmember1, struct Family *pmember2);
bool set_ancestry(struct Family *pmember1, struct Family *pmember2);
```

To show that the pointers have been successfully inserted, you can extend the final output to display information about the parents of each person by adding some additional statements immediately after the last `printf()`. You can also amend the output loop to start from `first` so the output loop will thus be as follows:

```
/* Output Family data in correct order */
current = first;

while (current != NULL)             /* Output Family data in correct order */
{
  printf("\n%s was born %d/%d/%d, and has %s and %s as parents.",
                current->name, current->dob.day, current->dob.month,
             current->dob. year, current->father,  current->mother);
  if(current->p_to_pa != NULL )
    printf("\n\t%s's birth date is %d/%d/%d  ",
               current->father, current->p_to_pa->dob.day,
                                current->p_to_pa->dob.month,
                                current->p_to_pa->dob.year);
  if(current->p_to_ma != NULL)
    printf("and %s's birth date is %d/%d/%d.\n  ",
               current->mother, current->p_to_ma->dob.day,
                                current->p_to_ma->dob.month,
                                current->p_to_ma->dob.year);

  current = current->next;          /* current points to next in list      */
}
```

This should then produce the dates of birth of both parents for each person using the pointers to the parents' structures, but only if the pointers have been set to valid addresses. Note that you don't free the memory in the loop. If you do this, the additional statements to output the parents' dates of birth will produce junk output when the parent structure appears earlier in the list. So finally, you must add a separate loop at the end of `main()` to delete the memory when the output is complete:

```
/* Now free the memory */
current = first;
while(current != NULL)
{
  last = current;     /* Save pointer to enable memory to be freed */
  current = current->next; /* current points to next in list       */
  free(last);         /* Free memory for last                      */
}
```

If you've assembled all the pieces into a new example, you should have a sizeable new program to play with. Here's the sort of output that you should get:

```
Do you want to enter details of a person (Y or N)? y

Enter the name of the person: Jack

Enter Jack's date of birth (day month year); 1 1 65

Who is Jack's father? Bill

Who is Jack's mother? Nell

Do you want to enter details of another person (Y or N)? y

Enter the name of the person: Mary

Enter Mary's date of birth (day month year); 3 3 67

Who is Mary's father? Bert

Who is Mary's mother? Moll

Do you want to enter details of another person (Y or N)? y

Enter the name of the person: Ben

Enter Ben's date of birth (day month year); 2 2 89

Who is Ben's father? Jack

Who is Ben's mother? Mary

Do you want to enter details of another  person (Y or N)? n

Jack was born 1/1/65, and has Bill and Nell as parents.
Mary was born 3/3/67, and has Bert and Moll as parents.
Ben was born 2/2/89, and has Jack and Mary as parents.
        Jack's birth date is 1/1/65  and Mary's birth date is 3/3/67.
```

You could try to modify the program to output everybody in chronological order or possibly work out how many offspring each person has.

Binary Trees

A binary tree is a very useful way of organizing data because you can arrange that the data that you have stored in the tree can be extracted from the tree in an ordered fashion. A binary tree is also a very interesting mechanism because it is basically very simple. Implementing a binary tree can involve recursion as well as dynamic memory allocation. We'll also use pointers to pass structures around.

A binary tree consists of a set of interconnected elements called **nodes**. The starting node is the base of the tree and is called a **root node**, as shown in Figure 11-5.

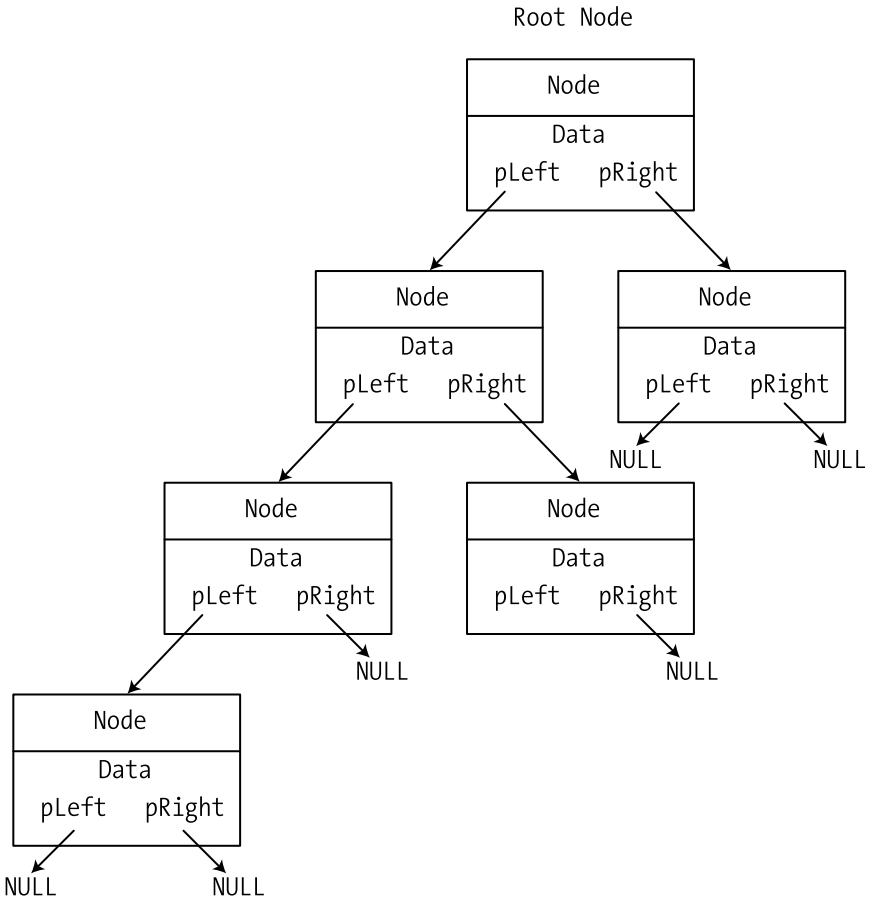

Figure 11-5. *The structure of a binary tree*

Each node typically contains an item of data, plus pointers to two subsidiary nodes, a **left node** and a **right node**. If either subsidiary node does not exist, the corresponding pointer is `NULL`. A node may also include a counter that records when there are duplicates of data items within the tree.

A `struct` makes it very easy to represent a binary tree node. Here's an example of a `struct` that defines nodes that store integers of type `long`:

```
struct Node
{
  long item;                /* The data item           */
  int count;                /* Number of copies of item */
  struct Node *pLeft;       /* Pointer to left node     */
  struct Node *pRight;      /* Pointer to right node    */
};
```

When a data item is to be added to the binary tree and that item already exists somewhere, a new node is not created; the `count` member of the existing node is simply incremented by 1. I'll use the previous definition of a `struct` that represents a node holding an integer when we get to creating a binary tree in practice in Program 11.7, which is the next working example in this chapter.

Ordering Data in a Binary Tree

The way in which you construct the binary tree will determine the order of data items within the tree. Adding a data item to a tree involves comparing the item to be added with the existing items in the tree. Typically items are added so that for every node, the data item stored in the subsidiary left node when it exists is less than the data item in the current node, and the data item stored in the right node when it exists is greater than the item stored in the node. An example of a tree containing an arbitrary sequence of integers is shown in Figure 11-6.

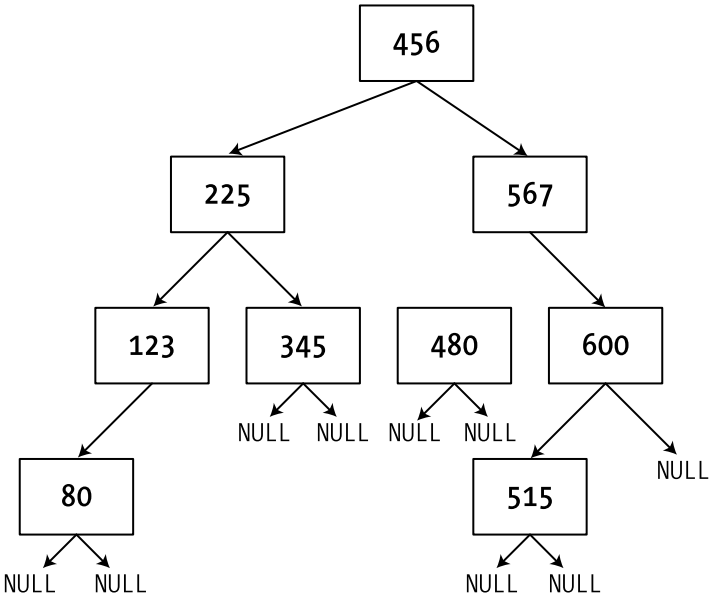

Figure 11-6. *A binary tree storing integers*

The structure of the tree will depend on the sequence in which the data items are added to the tree. Adding a new item involves comparing the data item with the values of the nodes in the tree starting with the root node. If it's less than a given node you then inspect the left subsidiary node, whereas if it's greater you look at the right subsidiary node. This process continues until either you find a node with an equal value, in which case you just update the count for that node, or you arrive at a left or right node pointer that is `NULL`, and that's where the new node is placed.

Constructing a Binary Tree

The starting point is the creation of the root node. All nodes will be created in the same way so the first step is to define a function that creates a node from a data item. I will assume we are creating a tree to store integers, so the `struct` definition you saw earlier will apply. Here's the definition of a function that will create a node:

```
struct Node *createnode(long value)
{
  /* Allocate memory for a new node */
  struct Node *pNode = (struct Node *)malloc(sizeof(struct Node));
```

```
  pNode->item = value;                      /* Set the value          */
  pNode->count = 1;                         /* Set the count          */
  pNode->pLeft = pNode->pRight = NULL;      /* No left or right nodes */
  return pNode;
}
```

The function allocates memory for the new `Node` structure and sets the `item` member to `value`. The `count` member is the number of duplicates of the value in the node so in the first instance this has to be 1. There are no subsidiary nodes at this point so the `pLeft` and `pRight` members are set to `NULL`. The function returns a pointer to the `Node` object that has been created.

To create the root node for a new binary tree you can just use this function, like this:

```
long newvalue;
printf("Enter the node value: ");
scanf(" %ld", newvalue);
struct Node *pRoot = createnode(newvalue);
```

After reading the value to be stored from the keyboard, you call the `createnode()` function to create a new node on the heap. Of course, you must not forget to release the memory for the nodes when you are done.

Working with binary trees is one of the areas where recursion really pays off. The process of inserting a node involves inspecting a succession of nodes in the same way, which is a strong indicator that recursion may be helpful. You can add a node to a tree that already exists with the following function:

```
/* Add a new node to the tree */
struct Node *addnode(long value, struct Node* pNode)
{
  if(pNode == NULL)                         /* If there's no node             */
    return createnode(value);               /* ...create one and return it    */

  if(value ==pNode->item)
  {                                         /* Value equals current node      */
    ++pNode->count;                         /* ...so increment count and      */
    return pNode;                           /* ...return the same node        */
  }

  if(value < pNode->item)                   /* If less than current node value */
  {
    if(pNode->pLeft == NULL)                /* and there's no left node       */
    {
      pNode->pLeft = createnode(value);     /* create a new left node and     */
      return pNode->pLeft;                  /* return it.                     */
    }
    else                                    /* If there is a left node...     */
      return addnode(value, pNode->pLeft); /* add value via the left node    */
  }
  else                                      /* value is greater than current  */
  {
    if(pNode->pRight == NULL)               /* so the same process with       */
    {                                       /* the right node.                */
      pNode-> pRight = createnode(value);
      return pNode-> pRight;
    }
```

```
    else
      return addnode(value, pNode-> pRight);
  }
}
```

The arguments to the `addnode()` function when you call it in the first instance are the value to be stored in the tree and the address of the root node. If you pass `NULL` as the second argument it will create and return a new node so you could also use this function to create the root node. When there is a root node passed as the second argument, there are three situations to deal with:

1. If `value` equals the value in the current node, no new node needs to be created, so you just increment the count in the current node and return it.
2. If `value` is less than the value in the current node, you need to inspect the left subsidiary node. If the left node pointer is `NULL`, you create a new node to hold `value` and make it the left subsidiary node. If the left node exists, you call `addnode()` recursively with the pointer to the left subsidiary node as the second argument.
3. If value is greater than the value in the current node, you proceed with the right node in the same way as with the left node.

Whatever happens within the recursive function calls, the function will return a pointer to the node where value is inserted. This may be a new node or one of the existing nodes if value is already in the tree somewhere.

You could construct a complete binary tree storing an arbitrary number of integers with the following code:

```
long newvalue = 0;
struct Node *pRoot = NULL;
char answer = 'n';
do
{
  printf("Enter the node value: ");
  scanf(" %ld", &newvalue);
  if(pRoot == NULL)
    pRoot = createnode(newvalue);
  else
  addnode(newvalue, pRoot);

  printf("\nDo you want to enter another (y or n)? ");
  scanf(" %c", &answer);
} while(tolower(answer) == 'y');
```

The `do-while` loop constructs the complete tree including the root node. On the first iteration `pRoot` will be `NULL`, so the root node will be created. All subsequent iterations will add nodes to the existing tree.

Traversing a Binary Tree

You can traverse a binary tree to extract the contents in ascending or descending sequence. I'll discuss how you extract the data in ascending sequence and you'll see how you can produce an descending sequence by analogy. At first sight it seems a complicated problem to extract the data from the tree because of its arbitrary structure but using recursion makes it very easy.

I'll start the explanation of the process by stating the obvious: the value of the left subnode is always less than the current node, and the value of the current node is always less than that of the

right subnode. You can conclude from this that the basic process is to extract the values in the sequence: left subnode, followed by current node, followed by right subnode. Of course, where the subnodes have subnodes, the process has to be applied to those too, from left through current to right. It will be easy to see how this works if we look at some code.

Suppose we want to simply list the integer values contained in our binary tree in ascending sequence. The function to do that looks like this:

```
/* List the node values in ascending sequence */
void listnodes(struct Node *pNode)
{
  if(pNode->pLeft != NULL)
    listnodes(pNode->pLeft);           /* List nodes in the left subtree */

  for(int i = 0; i<pNode->count ; i++)
    printf("\n%10ld", pNode->item);    /* Output the current node value  */

  if(pNode->pRight != NULL)
    listnodes(pNode->pRight);          /* List nodes in the right subtree */
}
```

It consists of three simple steps:

1. If the left subnode exists, you call `listnodes()` recursively for that node.
2. Write the value for the current node.
3. If the right subnode exists, call `listnodes()` recursively for that node.

The first step repeats the process for the left subnode of the root if it exists, so the whole subtree to the left will be written before the value of the current node. The value of the current node is repeated count times in the output to reflect the number of duplicate values for that node. The values for the entire right subtree to the root node will be written after the value for the current node. You should be able to see that this happens for every node in the tree, so the values will be written in ascending sequence. All you have to do to output the values in ascending sequence is to call `listnodes()` with the root node pointer as the argument, like this:

```
listnodes(pRoot);          /* Output the contents of the tree */
```

In case you find it hard to believe that such a simple function will output all the values in any binary tree of integers you care to construct, let's see it working.

TRY IT OUT: SORTING USING A BINARY TREE

This example drags together the code fragments you have already seen:

```
/* Program 11.7 Sorting integers using a binary tree */
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

/* Function prototypes */
struct Node *createnode(long value);                  /* Create a tree node */
struct Node *addnode(long value, struct Node* pNode); /* Insert a new node  */
void listnodes(struct Node *pNode);                   /* List all nodes     */
void freenodes(struct Node *pNode);                   /* Release memory     */
```

```
/* Defines a node in a binary tree sotring integers */
struct Node
{
  long item;                /* The data item            */
  int count;                /* Number of copies of item */
  struct Node *pLeft;       /* Pointer to left node     */
  struct Node *pRight;      /* Pointer to right node    */
};

/* Function main - execution starts here */
int main(void)
{
  long newvalue = 0;
  struct Node *pRoot = NULL;
  char answer = 'n';
  do
  {
    printf("Enter the node value: ");
    scanf(" %ld", &newvalue);
    if(pRoot == NULL)
      pRoot = createnode(newvalue);
    else
    addnode(newvalue, pRoot);
    printf("\nDo you want to enter another (y or n)? ");
    scanf(" %c", &answer);
  } while(tolower(answer) == 'y');

  printf("The values in ascending sequence are: ");
  listnodes(pRoot);          /* Output the contents of the tree */
  freenodes(pRoot);          /* Release the heap memory         */

  return 0;
}

struct Node *createnode(long value)
{
  struct Node *pNode = (struct Node *)malloc(sizeof(struct Node));
  pNode->item = value;                      /* Set the value          */
  pNode->count = 1;                         /* Set the count          */
  pNode->pLeft = pNode->pRight = NULL;      /* No left or right nodes */
  return pNode;
}

/* Add a new node to the tree */
struct Node *addnode(long value, struct Node* pNode)
{
  if(pNode == NULL)                         /* If there's no node             */
    return createnode(value);               /* ...create one and return it    */

  if(value ==pNode->item)
  {                                         /* Value equals current node      */
    ++pNode->count;                         /* ...so increment count and      */
    return pNode;                           /* ...return the same node        */
  }
```

```
  if(value < pNode->item)                 /* If less than current node value */
  {
    if(pNode->pLeft == NULL)              /* and there's no left node        */
    {
      pNode->pLeft = createnode(value);   /* create a new left node and      */
      return pNode->pLeft;                /* return it.                      */
    }
    else                                  /* If there is a left node...      */
      return addnode(value, pNode->pLeft); /* add value via the left node     */
  }
  else                                    /* value is greater than current   */
  {
    if(pNode->pRight == NULL)             /* so the same process with        */
    {                                     /* the right node.                 */
      pNode-> pRight = createnode(value);
      return pNode-> pRight;
    }
    else
      return addnode(value, pNode-> pRight);
  }
}

/* List the node values in ascending sequence */
void listnodes(struct Node *pNode)
{
  if(pNode->pLeft != NULL)
    listnodes(pNode->pLeft);

  for(int i = 0; i<pNode->count ; i++)
    printf("\n%10ld", pNode->item);

  if(pNode->pRight != NULL)
    listnodes(pNode->pRight);
}

/* Release memory allocated to nodes */
void freenodes(struct Node * pNode)
{
  if(pNode == NULL)                   /* If there's no node...        */
    return;                           /* we are done.                 */

  if(pNode->pLeft != NULL)            /* If there's a left sub-tree   */
    freenodes(pNode->pLeft);          /* free memory for those nodes. */

  if(pNode->pRight != NULL)           /* If there's a right sub-tree  */
    freenodes(pNode->pRight);         /* free memory for those nodes. */

  free(pNode);                        /* Free current node memory     */
}
```

Here is some typical output from this example:

```
Enter the node value: 56

Do you want to enter another (y or n)? y
Enter the node value: 33

Do you want to enter another (y or n)? y
Enter the node value: 77

Do you want to enter another (y or n)? y
Enter the node value: -10

Do you want to enter another (y or n)? y
Enter the node value: 100

Do you want to enter another (y or n)? y
Enter the node value: -5

Do you want to enter another (y or n)? y
Enter the node value: 200

Do you want to enter another (y or n)? n
The values in ascending sequence are:
       -10
        -5
        33
        56
        77
       100
       200
```

How It Works

The `do-while` loop in `main()` constructs the binary tree from the values that are entered in the way I discussed earlier. The loop continues as long as you enter `'y'` or `'Y'` when prompted. Calling `listnodes()` with the address of the root node as the argument outputs all the values in the tree in ascending sequence. You then call the `freenodes()` function to release the memory that is allocated for the nodes for the tree.

The `freenodes()` function is the only new code in the example. This is another recursive function that works in a similar way to the `listnodes()` function. It is essential to delete the memory for the subsidiary nodes of each node before freeing the memory for the node itself, because once you have freed a memory block, it could be used immediately by some other program that is executing concurrently. This means that the addresses for the subsidiary nodes are effectively unavailable once the memory has been released. The function therefore always calls `freenodes()` for the subsidiary node pointers if they are not `NULL` before releasing the memory for the current node.

You can construct binary trees to store any kind of data including `struct` objects and strings. If you want to organize strings in a binary tree for example, you could use a pointer in each node to refer to the string rather than making copies of the strings within the tree.

Sharing Memory

You've already seen how you can save memory through the use of bit-fields, which are typically applied to logical variables. C has a further capability that allows you to place several variables in the

same memory area. This can be applied somewhat more widely than bit-fields when memory is short, because circumstances frequently arise in practice in which you're working with several variables, but only one of them holds a valid value at any given moment.

Another situation in which you can share memory between a number of variables to some advantage is when your program processes a number of different kinds of data record, but only one kind at a time, and the kind to be processed is determined at execution time. A third possibility is that you want to access the same data at different times, and assume it's of a different type on different occasions. You might have a group of variables of numeric data types, for instance, that you want to treat as simply an array of type `char` so that you can move them about as a single chunk of data.

Unions

The facility in C that allows the same memory area to be shared by a number of different variables is called a **union**. The syntax for declaring a union is similar to that used for structures, and a union is usually given a tag name in the same way. You use the keyword `union` to define a union. For example, the following statement declares a union to be shared by three variables:

```
union u_example
{
  float decval;
  int *pnum;
  double my_value;
} U1;
```

This statement declares a union with the tag name `u_example`, which shares memory between a floating-point value `decval`, a pointer to an integer `pnum`, and a double precision floating-point variable `my_value`. The statement also defines one instance of the union with a variable name of `U1`. You can declare further instances of this union with a statement such as this:

```
union u_example U2, U3;
```

Members of a union are accessed in exactly the same way as members of a structure. For example, to assign values to members of `U1` and `U2`, you can write this:

```
U1.decval = 2.5;
U2.decval = 3.5 * U1.decval;
```

TRY IT OUT: USING UNIONS

Here's a simple example that makes use of a union:

```
/* Program 11.8 The operation of a union */
#include <stdio.h>

int main(void)
{
  union u_example
  {
    float decval;
    int pnum;
    double my_value;
  } U1;
```

```
  U1.my_value = 125.5;
  U1.pnum = 10;
  U1.decval = 1000.5f;
  printf("\ndecval = %f   pnum = %d   my_value = %lf",
                     U1.decval, U1.pnum, U1.my_value );

  printf("\nU1 size = %d\ndecval size = %d   pnum size = %d   my_value"
                              " size = %d",sizeof U1, sizeof U1.decval,
                                  sizeof U1.pnum, sizeof U1.my_value);
  return 0;
}
```

How It Works

This example demonstrates the structure and basic operation of a union. You declare the union `U1` as follows:

```
union u_example
{
  float decval;
  int pnum;
  double my_value;
} U1;
```

The three members of the union are of different types and they each require a different amount of storage (assuming your compiler assigns 2 bytes to variables of type `int`).

With the assignment statements, you assign a value to each of the members of the union instance `U1` in turn:

```
  U1.my_value = 125.5;
  U1.pnum = 10;
  U1.decval = 1000.5f;
```

Notice that you reference each member of the union in the same way as you do members of a structure.

The next two statements output each of the three member values, the size of the union `U1`, and the size of each of its members. You get this output (or something close if your machine assigns 4 bytes to variables of type `int`):

```
decval = 1000.500000   pnum = 8192   my_value = 125.50016
U1 size = 8
decval size = 4   pnum size = 2   my_value size = 8
```

The first thing to note is that the last variable that was assigned a value is correct, and the other two have been corrupted. This is to be expected, because they all share the same memory space. The second thing to notice is how little the member `my_value` has been corrupted. This is because only the least significant part of `my_value` is being modified. In a practical situation, such a small error could easily be overlooked, but the ultimate consequences could be dire. You need to take great care when using unions that you aren't using invalid data.

■ **Note** You can see from the output of the sizes of the union and its members that the size of the union is the same as the size of the largest member.

Pointers to Unions

You can also define a pointer to a union with a statement such as this:

```
union u_example *pU;
```

Once the pointer has been defined, you can modify members of the union, via the pointer, with these statements:

```
pU = &U2;
U1.decval = pU->decval;
```

The expression on the right of the second assignment is equivalent to `U2.decval`.

Initializing Unions

If you wish to initialize an instance of a union when you declare it, you can initialize it only with a constant of the same type as the first variable in the union. The union just declared, `u_example`, can be initialized only with a `float` constant, as in the following:

```
union u_example U4 = 3.14f;
```

You can always rearrange the sequence of members in a definition of a union so that the member that you want to initialize occurs first. The sequence of members has no other significance, because all members overlap in the same memory area.

Structures As Union Members

Structures and arrays can be members of a union. It's also possible for a union to be a member of a structure. To illustrate this, you could write the following:

```
struct my_structure
{
  int num1;
  float num2;
  union
  {
    int *pnum;
    float *pfnum;
  } my_U
} samples[5];
```

Here you declare a structure type, `my_structure`, which contains a union without a tag name, so instances of the union can exist only within instances of the structure. This is often described as an **anonymous union**. You also define an array of five instances of the structure, referenced by the variable name `samples`. The union within the structure shares memory between two pointers. To reference members of the union, you use the same notation that you used for nested structures. For example, to access the pointer to `int` in the third element of the structure array, you use the expression appearing on the left in the following statement:

```
samples[2].my_U.pnum = &my_num;
```

You're assuming here that the variable `my_num` has been declared as type `int`.

It's important to realize that when you're using a value stored in a union, you always retrieve the last value assigned. This may seem obvious, but in practice it's all too easy to use a value as `float` that has most recently been stored as an integer, and sometimes the error can be quite subtle, as

shown by the curious output of my_value in Program 11.7. Naturally, you'll usually end up with garbage if you do this. One technique that is often adopted is to embed a union in a struct that also has a member that specifies what type of value is currently stored in the union. For example

```
/* Type code for data in union */
#define TYPE_LONG   1
#define TYPE_FLOAT  2
#define TYPE_CHAR   3

struct Item
{
  int u_type;
  union
  {
    long integer;
    float floating;
    char ch;
  } u;
} var;
```

This defines the Item structure type that contains two members: a value u_type of type int, and an instance u of an anonymous union. The union can store a value of type long, type float or type char, and the u_type member is used to record the type that is currently stored in u.

You could set a value for var like this:

```
var.u.floating = 2.5f;
var.u_type = TYPE_FLOAT;
```

When you are processing var, you need to check what kind of value is stored. Here's an example of how you might do that:

```
switch(var.u_type)
{
  case TYPE_FLOAT:
  printf("\nValue of var is %10f", var.u.floating);
  break;
  case TYPE_LONG:
  printf("\nValue of var is %10ld", var.u.integer);
  break;
  case TYPE_CHAR:
  printf("\nValue of var is %10c", var.u.ch);
  break;
  default:
  printf("\nInvalid union type code.");
  break;
}
```

When working with unions is this way it is usually convenient to put code such as this in a function.

Defining Your Own Data Types

With structures you've come pretty close to defining your own data types. It doesn't look quite right because you must use the keyword struct in your declarations of structure variables. Declaration of a variable for a built-in type is simpler. However, there's a feature of the C language that permits you

to get over this and make the declaration of variables of structure types you've defined follow exactly the same syntax as for the built-in types. You can apply this feature to simplify types derived from the built-in types, but here, with structures, it really comes into its own.

Structures and the typedef Facility

Suppose you have a structure for geometric points with three coordinates, x, y, and z, that you define with the following statement:

```
struct pts
{
  int x;
  int y;
  int z;
};
```

You can now define an alternative name for declaring such structures using the keyword `typedef`. The following statement shows how you might do this:

```
typedef struct pts Point;
```

This statement specifies that the name `Point` is a synonym for `struct pts`.

When you want to declare some instances of the structure `pts`, you can use a statement such as this:

```
Point start_pt;
Point end_pt;
```

Here, you declare the two structure variables `start_pt` and `end_pt`. The `struct` keyword isn't necessary, and you have a very natural way of declaring structure variables. The appearance of the statement is exactly the same form as a declaration for a `float` or an `int`.

You could combine the `typedef` and the structure declaration as follows:

```
typedef struct pts
{
  int x;
  int y;
  int z;
} Point;
```

Don't confuse this with a basic `struct` declaration. Here, `Point` isn't a structure variable name—this is a type name you're defining. When you need to declare structure variables, as you've just seen, you can use a statement such as this:

```
Point my_pt;
```

There's nothing to prevent you from having several types defined that pertain to a single structure type, or any other type for that matter, although this can be confusing in some situations. One application of this that can help to make your program more understandable is where you're using a basic type for a specific kind of value and you would like to use a type name to reflect the kind of variable you're creating. For example, suppose your application involves weights of different kinds, such as weights of components and weights of assembly. You might find it useful to define a type name, `weight`, as a synonym for type `double`. You could do this with the following statement:

```
typedef double weight;
```

You can now declare variables of type `weight`:

```
weight piston = 6.5;
weight valve = 0.35;
```

Of course, these variables are all of type `double` because `weight` is just a synonym for `double`, and you can still declare variables of type `double` in the usual way.

Simplifying Code Using typedef

Another useful application of `typedef` is to simplify complicated types that can arise. Suppose you have occasion to frequently define pointers to the structure `pts`. You could define a type to do this for you with this statement:

```
typedef struct pts *pPoint;
```

Now, when you want to declare some pointers, you can just write this:

```
pPoint pfirst;
pPoint plast;
```

The two variables declared here are both pointers to structures of type `pts`. The declarations are less error-prone to write and they're also easier to understand.

In Chapter 9 I discussed pointers to functions, which are declared with an even more complicated notation. One of the examples has the pointer declaration:

```
int(*pfun)(int, int);                  /* Function pointer declaration */
```

If you are expecting to use several pointers to functions of this kind in a program, you can use `typedef` to declare a generic type for such declarations with this statement:

```
typedef int (*function_pointer)(int, int);    /* Function pointer type */
```

This doesn't declare a variable of type "pointer to a function." This declares `function_pointer` as a type name that you can use to declare a "pointer to function", so you can replace the original declaration of `pfun` with this statement:

```
function_pointer pfun;
```

This is evidently much simpler than what you started with. The benefit in simplicity is even more marked if you have several such pointers to declare, because you can declare three pointers to functions with the following statements:

```
function_pointer pfun1;
function_pointer pfun2;
function_pointer pfun3;
```

Of course, you can also initialize them, so if you assume you have the functions `sum()`, `product()`, and `difference()`, you can declare and initialize your pointers with the following:

```
function_pointer pfun1 = sum;
function_pointer pfun2 = difference;
function_pointer pfun3 = product;
```

The type name that you've defined naturally only applies to "pointers to functions" with the arguments and return type that you specified in the `typedef` statement. If you want something different, you can simply define another type.

Designing a Program

You've reached the end of another long chapter, and it's time to see how you can put what you've learned into practice in the context of a more substantial example.

The Problem

Numerical data is almost always easier and faster to understand when it's presented graphically. The problem that you're going tackle is to write a program that produces a vertical bar chart from a set of data values. This is an interesting problem for a couple of reasons. It will enable you to make use of structures in a practical context. The problem also involves working out how to place and present the bar chart within the space available, which is the kind of messy manipulation that comes up quite often in real-world applications.

The Analysis

You won't be making any assumptions about the size of the "page" that you're going to output to, or the number of columns, or even the scale of the chart. Instead, you'll just write a function that accepts a dimension for the output page and then makes the set of bars fit the page, if possible. This will make the function useful in virtually any situation. You'll store the values in a sequence of structures in a linked list. In this way, you'll just need to pass the first structure to the function, and the function will be able to get at them all. You'll keep the structure very simple, but you can embellish it later with other information of your own design.

Assume that the order in which the bars are to appear in the chart is going to be the same as the order in which the data values are entered, so you won't need to sort them. There will be two functions in your program: a function that generates the bar chart, and a function `main()` that exercises the bar chart generation process.

These are the steps required:

1. Write the bar-chart function.
2. Write a `main()` function to test the bar-chart function once you've written it.

The Solution

This section outlines the steps you'll take to solve the problem.

Step 1

Obviously, you're going to use a structure in this program because that's what this chapter is about. The first stage is to design the structure that you'll use throughout the program. You'll use a `typedef` so that you don't have to keep reusing the keyword `struct`.

```
/* Program 11.9 Generating a bar chart */
#include <stdio.h>

typedef struct barTAG
{
  double value;
  struct barTAG *pnextbar;
}bar;
```

```
int main(void)
{
  /* Code for main */
}

/* Definition of the bar-chart function */
```

The barTAG structure will define a bar simply by its value. Notice how you define the pointer in the structure to the next structure. This will enable you to store the bars as a linked list, which has the merit that you can allocate memory as you go so none will be wasted. This suits this situation because you'll only ever want to step through the bars from the first to the last. You'll create them in sequence from the input values and append each new bar to the tail of the previous one. You'll then create the visual representation of the bar chart by stepping through the structures in the linked list. You may have thought that the typedef statement would mean that you could use the bar type name that you're defining here. However, you have to use struct barTAG here because at this point the compiler hasn't finished processing the typedef yet, so bar isn't defined. In other words, the barTAG structure is analyzed first by the compiler, after which the typedef can be expedited to define the meaning of bar.

Now you can specify the function prototype for the bar-chart function and put the skeleton of the definition for the function. It will need to have parameters for a pointer to the first bar in the linked list, the page height and width, and the title for the chart to be produced:

```
/* Program 11.9 Generating a bar chart */
#include <stdio.h>

#define PAGE_HEIGHT  20
#define PAGE_WIDTH   40
typedef struct barTAG
{
  double value;
  struct barTAG *pnextbar;
}bar;

typedef unsigned int uint;     /* Type definition*/

/* Function prototype */
int bar_chart(bar *pfirstbar, uint page_width, uint page_height, char *title);

int main(void)
{
   /* Code for main */
}

int bar_chart(bar *pfirstbar, uint page_width, uint page_height,
                                                    char *title)
{
  /* Code for function...  */
  return 0;
}
```

You've added a typedef to define uint as an alternative to unsigned int. This will shorten statements that declare variables of type unsigned int.

Next, you can add some declarations and code for the basic data that you need for the bar chart. You'll need the maximum and minimum values for the bars and the vertical height of the chart, which

will be determined by the difference between the maximum and minimum values. You also need to calculate the width of a bar, given the page width and the number of bars, and you must adjust the height to accommodate a horizontal axis and the title:

```
/* Program 11.9 Generating a bar chart */
#include <stdio.h>

#define PAGE_HEIGHT  20
#define PAGE_WIDTH   40

typedef struct barTAG
{
  double value;
  struct barTAG *pnextbar;
}bar;

typedef unsigned int uint;     /* Type definition */

/* Function prototype */
int bar_chart(bar *pfirstbar, uint page_width, uint page_height,
                                                    char *title);

int main(void)
{
   /* Code for main */
}

int bar_chart(bar *pfirstbar, uint page_width, uint page_height,
                                                    char *title)
{
  bar *plastbar = pfirstbar;  /* Pointer to previous bar         */
  double max = 0.0;           /* Maximum bar value               */
  double min = 0.0;           /* Minimum bar value               */
  double vert_scale = 0.0;    /* Unit step in vertical direction */
  uint bar_count = 1;         /* Number of bars - at least 1     */
  uint barwidth = 0;          /* Width of a bar                  */
  uint space = 2;             /* spaces between bars             */

   /* Find maximum and minimum of all bar values */

  /* Set max and min to first bar value */
  max = min = plastbar->value;

  while((plastbar = plastbar->pnextbar) != NULL)
  {
    bar_count++;              /* Increment bar count */
    max = (max < plastbar->value)? plastbar->value : max;
    min = (min > plastbar->value)? plastbar->value : min;
  }
  vert_scale = (max - min)/page_height; /* Calculate step length */
```

```
  /* Check bar width */
  if((barwidth = page_width/bar_count - space) < 1)
  {
    printf("\nPage width too narrow.\n");
    return -1;
  }

  /* Code for rest of the function...  */
  return 0;
}
```

The space variable stores the number of spaces separating one bar from the next, and you arbitrarily assign the value 2 for this.

You will, of necessity, be outputting the chart a row at a time. Therefore, you'll need a string that corresponds to a section across a bar that you can use to draw that bar row by row, and a string of the same length, containing spaces to use when there's no bar at a particular position across the page. Let's add the code to create these:

```
/* Program 11.9 Generating a bar chart */
#include <stdio.h>
#include <stdlib.h>

#define PAGE_HEIGHT  20
#define PAGE_WIDTH   40

typedef struct barTAG
{
  double value;
  struct barTAG *pnextbar;
}bar;

typedef unsigned int uint;     /* Type definition */

/* Function prototype */
int bar_chart(bar *pfirstbar, uint page_width, uint page_height,
                                                    char *title);

int main(void)
{
  /* Code for main */
}

int bar_chart(bar *pfirstbar, uint page_width, uint page_height,
                                                    char *title)
{
  bar *plastbar = pfirstbar;  /* Pointer to previous bar         */
  double max = 0.0;           /* Maximum bar value               */
  double min = 0.0;           /* Minimum bar value               */
  double vert_scale = 0.0;    /* Unit step in vertical direction */
  uint bar_count = 1;         /* Number of bars - at least 1     */
  uint barwidth = 0;          /* Width of a bar                  */
  uint space = 2;             /* spaces between bars             */
  uint i = 0;                 /* Loop counter                    */
  char *column = NULL;        /* Pointer to bar column section   */
  char *blank = NULL;         /* Blank string for bar+space      */
```

```
  /* Find maximum and minimum of all bar values */

  /* Set max and min to first bar value */
  max = min = plastbar->value;

  while((plastbar = plastbar->pnextbar) != NULL)
  {
    bar_count++;              /* Increment bar count */
    max = (max < plastbar->value)? plastbar->value : max;
    min = (min > plastbar->value)? plastbar->value : min;
  }
  vert_scale = (max - min)/page_height; /* Calculate step length */

  /* Check bar width */
  if((barwidth = page_width/bar_count - space) < 1)
  {
    printf("\nPage width too narrow.\n");
    return -1;
  }

  /* Set up a string that will be used to build the columns */

  /* Get the memory */
  if((column = malloc(barwidth + space + 1)) == NULL)
  {
    printf("\nFailed to allocate memory in barchart()"
                          " - terminating program.\n");
    exit(1);
  }
  for(i = 0 ; i<space ; i++)
    *(column+i)=' ';          /* Blank the space between bars */
  for( ; i<space+barwidth ; i++)
    *(column+i)='#';          /* Enter the bar characters     */
  *(column+i) = '\0';         /* Add string terminator        */

  /* Set up a string that will be used as a blank column */

  /* Get the memory */
  if((blank = malloc(barwidth + space + 1)) == NULL)
  {
    printf("\nFailed to allocate memory in barchart()"
                          " - terminating program.\n");
    exit(1);
  }

  for(i = 0 ; i<space+barwidth ; i++)
    *(blank+i) = ' ';         /* Blank total width of bar+space */
  *(blank+i) = '\0';          /* Add string terminator          */

  /* Code for rest of the function...  */
  free(blank);                /* Free memory for blank string   */
  free(column);               /* Free memory for column string  */
  return 0;
}
```

You'll draw a bar using '#' characters. When you draw a bar, you'll write a string containing space spaces and barwidth '#' characters. You allocate the memory for this dynamically using the library function malloc(), so you must add an #include directive for the header file stdlib.h. The string that you'll use to draw a bar is column, and blank is a string of the same length containing spaces. After the bar chart has been drawn and just before you exit, you free the memory occupied by column and blank.

Next, you can add the final piece of code that draws the chart:

```
/* Program 11.9 Generating a bar chart */
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

#define PAGE_HEIGHT  20
#define PAGE_WIDTH   40

typedef struct barTAG
{
  double value;
  struct barTAG *pnextbar;
}bar;

typedef unsigned int uint;     /* Type definition */

/* Function prototype */
int bar_chart(bar *pfirstbar, uint page_width, uint page_height,
                                                  char *title);
int main(void)
{
   /* Code for main */
}

int bar_chart(bar *pfirstbar, uint page_width, uint page_height,
                                                  char *title)
{
  bar *plastbar = pfirstbar;  /* Pointer to previous bar            */
  double max = 0.0;           /* Maximum bar value                  */
  double min = 0.0;           /* Minimum bar value                  */
  double vert_scale = 0.0;    /* Unit step in vertical direction    */
  double position = 0.0;      /* Current vertical position on chart */
  uint bar_count = 1;         /* Number of bars - at least 1        */
  uint barwidth = 0;          /* Width of a bar                     */
  uint space = 2;             /* spaces between bars                */
  uint i = 0;                 /* Loop counter                       */
  uint bars = 0;              /* Loop counter through bars          */
  char *column = NULL;        /* Pointer to bar column section      */
  char *blank = NULL;         /* Blank string for bar+space         */
  bool axis = false;          /* Indicates axis drawn               */

  /* Find maximum and minimum of all bar values */

  /* Set max and min to first bar value */
  max = min = plastbar->value;
```

```
while((plastbar = plastbar->pnextbar) != NULL)
{
  bar_count++;              /* Increment bar count */
  max = (max < plastbar->value)? plastbar->value : max;
  min = (min > plastbar->value)? plastbar->value : min;
}
vert_scale = (max - min)/page_height; /* Calculate step length */

/* Check bar width */
if((barwidth = page_width/bar_count - space) < 1)
{
  printf("\nPage width too narrow.\n");
  return -1;
}

/* Set up a string that will be used to build the columns */

/* Get the memory */
if((column = malloc(barwidth + space + 1)) == NULL)
{
  printf("\nFailed to allocate memory in barchart()"
                        " - terminating program.\n");
  exit(1);
}
for(i = 0 ; i<space ; i++)
  *(column+i)=' ';         /* Blank the space between bars */
for( ; i < space+barwidth ; i++)
  *(column+i)='#';         /* Enter the bar characters     */
*(column+i) = '\0';        /* Add string terminator        */

/* Set up a string that will be used as a blank column */

/* Get the memory */
if((blank = malloc(barwidth + space + 1)) == NULL)
{
  printf("\nFailed to allocate memory in barchart()"
                        " - terminating program.\n");
  exit(1);
}

for(i = 0 ; i<space+barwidth ; i++)
  *(blank+i) = ' ';        /* Blank total width of bar+space */
*(blank+i) = '\0';         /* Add string terminator          */

printf("^ %s\n", title);   /* Output the chart title      */
```

```
  /* Draw the bar chart */
  position = max;
  for(i = 0 ; i <= page_height ; i++)
  {
    /* Check if we need to output the horizontal axis */
    if(position <= 0.0 && !axis)
    {
      printf("+");           /* Start of horizontal axis    */
      for(bars = 0; bars < bar_count*(barwidth+space); bars++)
        printf("-");         /* Output horizontal axis      */
      printf(">\n");
      axis = true;           /* Axis was drawn              */
      position -= vert_scale;/* Decrement position           */
      continue;
    }
    printf("|");             /* Output vertical axis        */
    plastbar = pfirstbar;    /* start with the first bar    */

    /* For each bar... */
    for(bars = 1; bars <= bar_count; bars++)
    {
      /* If position is between axis and value, output column */
      /* otherwise output blank                               */
      printf("%s", position <= plastbar->value &&
                   plastbar->value >= 0.0 && position > 0.0 ||
                   position >= plastbar->value &&
                   plastbar->value <= 0.0 &&
                   position <= 0.0 ? column: blank);
      plastbar = plastbar->pnextbar;
    }
    printf("\n");            /* End the line of output      */
    position -= vert_scale;  /* Decrement position          */
  }
  if(!axis)            /* Have we output the horizontal axis? */
  {                    /* No, so do it now                    */
    printf("+");
    for(bars = 0; bars < bar_count*(barwidth+space); bars++)
      printf("-");
    printf(">\n");
  }

  free(blank);               /* Free memory for blank string   */
  free(column);              /* Free memory for column string  */
  return 0;
}
```

The `for` loop outputs `page_height` lines of characters. Each line will represent a distance of `vert_scale` on the vertical axis. You get this value by dividing `page_height` by the difference between the maximum and minimum values. Therefore, the first line of output corresponds to `position` having the value `max`, and it's decremented by `vert_scale` on each iteration until it reaches `min`.

On each line, you must decide first if you need to output the horizontal axis. This will be necessary when `position` is less than or equal to 0 and you haven't already displayed the axis.

On lines other than the horizontal axis, you must decide what to display for each bar position. This is done in the inner `for` loop that repeats for each bar. The conditional operator in the `printf()` call outputs either `column` or `blank`. You output `column` if `position` is between the value of the bar and 0, and you output `blank` otherwise. Having output a complete row of bar segments, you output `'\n'` to end the line and decrement the value of `position`.

It's possible that all the bars could be positive, in which case you need to make sure that the horizontal axis is output after the loop is complete, because it won't be output from within the loop.

Step 2

Now you just need to implement `main()` to exercise the `bar_chart()` function:

```
/* Program 11.9 Generating a bar chart */
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stdbool.h>

#define PAGE_HEIGHT  20
#define PAGE_WIDTH   40

typedef struct barTAG         /* Bar structure       */
{
  double value;              /* Value of bar        */
  struct barTAG *pnextbar;   /* Pointer to next bar */
}bar;                         /* Type for a bar      */

typedef unsigned int uint;    /* Type definition     */

/* Function prototype */
int bar_chart(bar *pfirstbar, uint page_width, uint page_height, char *title);

int main()
{
  bar firstbar;             /* First bar structure */
  bar *plastbar = NULL;     /* Pointer to last bar */
  char value[80];           /* Input buffer        */
  char title[80];           /* Chart title         */

  printf("\nEnter the chart title: ");
  gets(title);              /* Read chart title    */

  for( ;; )                 /* Loop for bar input  */
  {
    printf("Enter the value of the bar, or use quit to end: ");
    gets(value);
```

```
    if(strcmp(value, "quit") == 0)   /* quit entered?       */
     break;                          /* then input finished */

    /* Store in next bar */
    if(!plastbar)                   /* First time?         */
    {
      firstbar.pnextbar = NULL;  /* Initialize next pointer */
      plastbar = &firstbar;      /* Use the first           */
    }
    else
    {
      /* Get memory */
      if(!(plastbar-> = malloc(sizeof(bar))))
      {
        printf("Oops! Couldn't allocate memory\n");
        return -1;
      }
      plastbar = plastbar->pnextbar;    /* Old next is new bar  */
      plastbar->pnextbar = NULL;        /* New bar next is NULL */
    }
    plastbar->value = atof(value);      /* Store the value      */
  }

  /* Create bar-chart */
  bar_chart(&firstbar, PAGE_WIDTH, PAGE_HEIGHT, title);

  /* We are done, so release all the memory we allocated */
  while(firstbar.pnextbar)
  {
    plastbar = firstbar.pnextbar;           /* Save pointer to next */
    firstbar.pnextbar = plastbar->pnextbar; /* Get one after next   */
    free(plastbar);                         /* Free next memory     */
  }
  return 0;
}

int bar_chart(bar *pfirstbar, uint page_width, uint page_height, char *title)
{
  /* Implementation of function as before... */
}
```

After reading the chart title using `gets()`, you read successive values in the `for` loop. For each value other than the first, you allocate the memory for a new bar structure before storing the value. Of course, you keep track of the first structure, `firstbar`, because this is the link to all the others, and you track the pointer to the last structure that you added so that you can update its `pnextbar` pointer when you add another. Once you have all the values, you call `bar_chart()` to produce the chart. Finally, you delete the memory for the bars. Note that you need to take care to not delete `firstbar`, as you didn't allocate the memory for this dynamically. You need an `#include` directive for `string.h` because you use the `gets()` function.

All you do then is add a line to `main()` that actually prints the chart from the values typed in. Typical output from the example is shown here:

```
Enter the chart title: Trial Bar Chart
Enter the value of the bar, or use quit to end: 6
Enter the value of the bar, or use quit to end: 3
Enter the value of the bar, or use quit to end: -5
Enter the value of the bar, or use quit to end: -7
Enter the value of the bar, or use quit to end: 9
Enter the value of the bar, or use quit to end: 4
Enter the value of the bar, or use quit to end: quit

^ Trial Bar Chart
|                          ####
|                          ####
|                          ####
|                          ####
|  ####                    ####
|  ####                    ####
|  ####                    ####
|  ####                    ####  ####
|  ####  ####              ####  ####
|  ####  ####              ####  ####
|  ####  ####              ####  ####
|  ####  ####              ####  ####
+----------------------------------->
|              ####  ####
|              ####  ####
|              ####  ####
|              ####  ####
|              ####  ####
|                    ####
|                    ####
|                    ####
```

Summary

This has been something of a marathon chapter, but the topic is extremely important. Having a good grasp of structures rates alongside understanding pointers and functions in importance, if you want to use C effectively.

Most real-world applications deal with things such as people, cars, or materials, which require several different values to represent them. Structures in C provide a ready tool for dealing with these sorts of complex objects. Although some of the operations may seem a little complicated, remember that you're dealing with complicated entities, so the complexity isn't implicit in the programming capability; rather, it's built into the problem you're tackling.

In the next chapter, you'll look at how you can store data in external files. This will, of course, include the ability to store structures.

Exercises

The following exercises enable you to try out what you've learned in this chapter. If you get stuck, look back over the chapter for help. If you're still stuck, you can download the solutions from the Source Code/Download area of the Apress web site (`http://www.apress.com`), but that really should be a last resort.

Exercise 11-1. Define a `struct` type with the name `Length` that represents a length in yards, feet, and inches. Define an `add()` function that will add two `Length` arguments and return the sum as type `Length`. Define a second function, `show()`, that will display the value of its `Length` argument. Write a program that will use the `Length` type and the `add()` and `show()` functions to sum an arbitrary number of lengths in yards, feet, and inches that are entered from the keyboard and output the total length.

Exercise 11-2. Define a `struct` type that contains a person's name consisting of a first name and a second name, plus the person's phone number. Use this `struct` in a program that will allow one or more names and corresponding numbers to be entered and will store the entries in an array of structures. The program should allow a second name to be entered and output all the numbers corresponding to the name, and optionally output all the names with their corresponding numbers.

Exercise 11-3. Modify or reimplement the program from the previous exercise to store the structures in a linked list in ascending alphabetical order of the names.

Exercise 11-4. Write a program to use a struct to count the number of occurrences of each different word in a paragraph of text that's entered from the keyboard.

Exercise 11-5. Write a program that reads an arbitrary number of names consisting of a first name followed by a last name. The program should use a binary tree to output the names in ascending alphabetical sequence ordered by first name within second name (i.e., second name takes precedence in the ordering so Ann Choosy comes after Bill Champ and before Arthur Choosy).