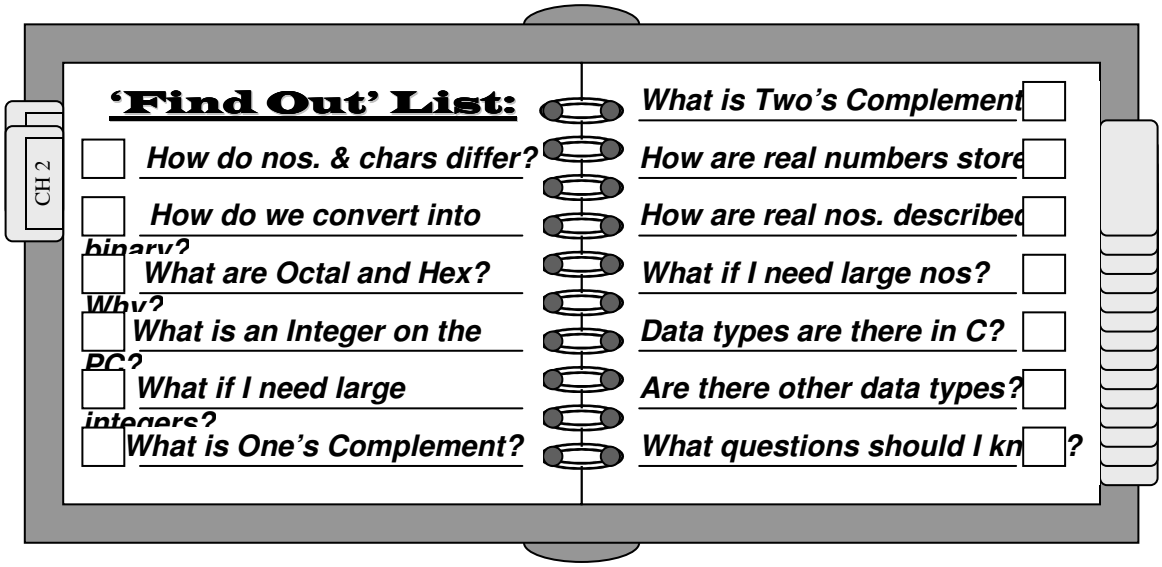
CHAPTER 7:
STRUCTURED DATA OBJECTS

"Simplicity of life, even the barest, is not a misery,
but the very foundation of refinement"
William Morris (1834–96)



Introduction

In this chapter, we ***construct*** our first new data type. We say *construct* because we are actually developing a *new* data type which can be manipulated in the same fashion as can any of our basic data types (e.g., **char**, **int**, **float**). We can associate variable names, or locations in RAM with them. We can construct arrays of **structs**, just as we can construct arrays of data type **char**, or arrays of data type **int**, or arrays of data type **float**. We can use our **sizeof** operator to determine how many bytes of storage we will require. In short, we can consider them as we would any basic data type, except that they may be of variable length.

Students often have trouble with this concept:

> ***How can a structure which we create be considered a data type? After all, aren't data types set in stone by some higher authority, say, Bill Gates?***

No, Virginia, they are not. We can construct our own data types, and they can be just as valid as any of the data types we have discussed to date.

Structured Data Types (Structs)

Recall our definition of an array: *an array is a **fixed** number of **contiguous** storage elements **all of the same data type***. We are about to loosen one of these restrictions, namely that the elements be **all of the same data type**.

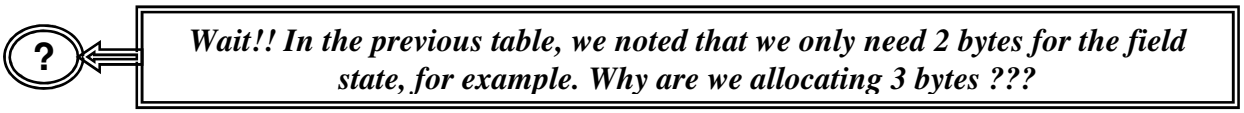
This is a fairly common data structure. We are quite used to the concept of records in a database, given that information about each of us is available in any number of databases. Think about the type of information that is kept on us, however. The registrar at a university might maintain some of the following information:

Table 7.1.

SSN	Name	Street	City	State	Zip	Class	GPA	Credit Hrs	Balance
123-45-6789	Smith, Joe	123 Main St.	Miam	FL	33134	3	3.267	66	2,028.28

Quite a few. Exactly how many different types must be maintained is a subject of debate, and we have a lot of options. For example, should SSN and Zipcodes be stored as characters or numbers? Consider the possibilities given in Table 7.2.

There is no *best* combination. All of the questions asked above really should be answered first. However, let's assume that we agree on the amount of storage necessary for each of the elements as shown in Table 7.3.



Remember, in C, a string requires one additional byte for the NULL character.

We require a total of 128 bytes (10 + 31 + 41 + 26 + 3 + 6 + 1 + 4 + 2 + 4 = 128) of ***contiguous*** storage for each record. Once again, notice the emphasis on *contiguous*. Assume that we wished to retain the following information as shown in Table 7.4

Table 7.2

Data Item	Data type/variable	Example	Comments
SSN (Social Security No.)	**char** *ssn*[11]; **char** *ssn*[9]; **long** *ssn;* **unsigned long** *ssn;*	"123-45-6789" "123456789" 123456789 123456789	If we include hyphens Without hyphens Max Val: 2,147,483,647 Max Val: 4,294,967,295
Name	**char** *name*[???]; **char** *lastname*[??], *firstname*[??];	"Smith, Joe" "Smith" "Joe"	How long is a name? Should we break up name? How long should each be?
Street	**char** *street*[???]; **char** *apt*[???], *street*[???]; **char** *street1*[???], *street2*[???];	"123 Main St." "Apt. 5A" "123 Main St." "Mesa Village" "123 Main St."	How Many Characters? Should we have this? How Many Characters? Should there be two lines? How Many Characters?
City	**char** *city*[???];	"Miami"	How many Characters?
State	**char** *state*[2];	"FL"	This one is easy
Zipcode	**char** *zip*[5]; **char** *zip*[9]; **char** *zip*[10]; **unsigned long** *zip*;	"33134" "331340012" "33134-0012" 331340012	Regular Zipcode Extended Zipcode Extended Zip with hyphen Max Value: 4,294,967,295
Class	**int** class; **char** class; **unsigned char** class; **char** class[???];	3 '3' or 'A' '3' or 'A' "4G"	3 = Junior Max Classes: 128 Max Classes: 256 How many classes ??
Grade Point Average	**float** *gpa*;	3.267	This is pretty easy
Credit Hours	**int** *hrs*;	66	This is also pretty easy
Balance Owed	**float** *balance*;	2028.28	This is also pretty easy

Table 7.3.

Field	SSN	name	street	city	state	zip	class	gpa	hrs	Balance
Datatype	**char**[10]	**char**[31]	**char**[41]	**char**[26]	**char**[3]	**char**[6]	**char**	**float**	**int**	**float**
Bytes	10	31	41	26	3	6	1	4	2	4

Table 7.4.

SSN	name	street	city	state	Zip	class	gpa	hrs	balance
123456789	Clinton, H.	123 Main	Dallas	TX	12345	4	3.145	78	564.89

If we could look in RAM (assuming a base address of 5000) we might see:

Table 7.5

5000	5001	5002	5003	5004	5005	5006	5007	5009	5010	5011	5012	5013
'1'	'2'	'3'	'4'	'5'	'6'	'7'	'9'	'\0'	'C'	'l'	'i'	'n'
5014	5015	5016	5017	5018	5019	5020	5021	5022	5023	5024	5025	5026
't'	'o'	'n'	','	' '	'H'	'.'	'\0'	---	---	---	---	---
5027	5028	5029	5030	5031	5032	5033	5034	5035	5036	5037	5038	5039
---	---	---	---	---	---	---	---	---	---	---	---	---
5040	5041	5042	5043	5044	5045	5046	5047	5048	5049	5050	5051	5052
---	'1'	'2'	'3'	' '	'M'	'a'	'i'	'n'	'\0'	---	---	---
5053	5054	5055	5056	5057	5058	5059	5060	5061	5062	5063	5064	5065
---	---	---	---	---	---	---	---	---	---	---	---	---
5066	5067	5068	5069	5070	5071	5072	5073	5074	5075	5076	5077	5078
---	---	---	---	---	---	---	---	---	---	---	---	---
5079	5080	5081	5082	5083	5084	5085	5086	5087	5088	5089	5090	5091
---	---	---	'D'	'a'	'l'	'l'	'a'	's'	'\0'	---	---	---
5092	5093	5094	5095	5096	5097	5098	5099	5100	5101	5102	5103	5104
---	---	---	---	---	---	---	---	---	---	---	---	---
5105	5106	5107	5108	5109	5110	5111	5112	5113	5114	5115	5116	5117
---	---	---	'T'	'X'	'\0'	'l'	'2'	'3'	'4'	'5'	'\0'	'4'
5118	5119	5120	5121	5122	5123	5124	5125	5126	5127			
3.145				78		564.89						

Or, on a more primitive level, as shown in Table 7.6.

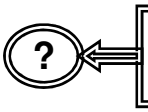
Once again, we need to examine the agreed-upon allotment as given in Table 7.3. Let's look at the fields and how many bytes each field requires, as specified in Table 7.7.

Table 7.6.

5000	5001	5002	5003	5004	5005	5006	5007	5008	5009	5010
00110001	00110010	00110011	00110100	00110101	00110110	00110111	00111000	00111001	00000000	01000011
5011	5012	5013	5014	5015	5016	5017	5018	5019	5020	5021
01101100	01101001	01101110	01110100	01101111	01101110	00101100	00100000	01001000	00101110	00000000
5022	5023	5024	5025	5026	5027	5028	5029	5029	5030	5031
01100000	00010010	10011100	00000000	00111110	10110011	10110010	00000000	00000000	11001110	00100110
5032	5033	5034	5035	5036	5037	5038	5039	5040	5041	5042
01011111	00111110	01011000	00000111	01111110	00000100	10111001	01100101	00000000	00110001	00110010
5043	5044	5045	5046	5047	5048	5049	5050	5051	5052	5053
00110011	00100000	01001101	01100001	01101001	01101110	00100000	01010011	00110100	00011000	00000000
5054	5055	5056	5057	5058	5059	5060	5061	5062	5063	5064
01110100	10111001	00000011	00110110	01100010	10011111	00000000	01011001	00000011	01110011	00011100
5065	5066	5067	5068	5069	5070	5071	5072	5073	5074	5075
00110001	00001110	00000000	11111010	00111101	00110011	00001111	11110010	00011110	10101101	01110011
5076	5077	5078	5079	5080	5081	5082	5083	5084	5085	5086
00000000	01110011	00000110	00000000	00111100	01010100	01000100	01100001	01101100	01101100	01100001
5087	5088	5089	5090	5091	5092	5093	5094	5095	5096	5097
01110011	00000000	01100011	00000111	01111100	11000001	01100000	00000010	00101010	01110100	01010011
5098	5099	5100	5101	5102	5103	5104	5105	5106	5107	5108
01110100	00011100	00000000	01010011	00000000	10011111	00000011	10011111	01010011	01110100	01010100
5109	5110	5111	5112	5113	5114	5115	5116	5117	5118	5119
01011000	00000000	00110001	00110010	00110011	00110100	00110101	00000000	00110100	01000001	00110001
5120	5121	5122	5123	5124	5125	5126	5127			
00000000	00010001	00000000	01001110	01000011	00000000	00000000	01101100			

Table 7.7.

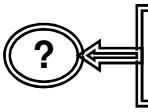
Field	Data Type	Bytes	Starting Address	Ending Address
Ssn	**char**[10]	10	5000	5009
name	**char**[31]	31	5010	5040
street	**char**[41]	41	5041	5081
city	**char**[26]	26	5082	5107
state	**char**[3]	3	5108	5110
zip	**char**[6]	6	5111	5116
class	**char**	1	5117	5117
gpa	**float**	4	5118	5121
hrs	**int**	2	5122	5123
balance	**float**	4	5124	5127



Why are there so many unused/'garbage' bytes (of the 128-bytes in the record above, 71 (or 55%) are unused) ???

That is one of the problems associated with allocating fixed storage spaces for fields in records. Generally speaking, this is not a major problem when storing numeric values. An integer, for example, will require only 2-bytes of storage, regardless of magnitude (unless of course, some integers are outside the range for 2-bit integers, in which case we need to redefine the field as a long). The same is true for floats.

Character strings are more problematic. *How many characters do we really need to adequately store a street name, for example?* Some street addresses may require very little storage (e.g., 5 B St.) while others may require considerably more (e.g., 12367-89A Northwest Mount Saint Helena Boulevard). The problem is further compounded when we consider large databases (e.g., 100 million records), and by the fact that records require contiguous storage. If a record becomes too large (e.g., requiring thousands of bytes of contiguous storage for each record), we may run out of space (even though non-contiguous storage may be available). These are not issues that we will deal with here, but they are topics of concern for database designers.



Why are we allocating 10-bytes for SSN, when a social security number only consists of 9 digits ???

Because we are storing SSN as a character array, we should allocate an extra byte for the null character. As we have seen it makes string manipulation much easier. Of course we don't have to, but if we don't, we have to keep track of the length of each of the strings.

To save space, why don't we save SSN as a numeric data type ???

We could, but once again, there are trade-offs. Firstly, we would have to save it as a long data-type on 4-bytes (a savings of 6-bytes per record) since the longest possible SSN (999-99-9999) requires 9 decimals (the largest number which even an unsigned integer could represent is 65535 (4-decimals); the largest (signed) long number is 2147483647, meaning we could represent all 9-digits. However, what would happen, for example, if someone had the social security number 000-00-100? The numeric value 100 would be stored, and in order to print it out, we would have to do some fancy manipulation (i.e., the leading zeros would have to be added back before printing). Once again, this is an issue for the database designer.

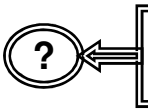
As with other variables, when we declare a record in the c programming language we are requesting that a fixed number of contiguous bytes be reserved in RAM and at what base address they can be found. We also indicating how the bits are to be manipulated by

specifying the component data types found in our structured data object. Consider the instructions given in C/C++ code 7.1.:

C/C++ Code 7.1

```
struct student
{   char ssn[10];
    char name[31];
    char street[41];
    char city[26];
    char state[3];
    char zip[6];
    char class;
    float gpa;
    int hrs;
    float balance;
}
```

The c code above is actually the structure template. Essentially, we are defining a new data type, which we have called **struct student**.



How can this be a new data type??? It is nothing more than a collection of basic data types.

It *is* a collection of basic data types, *BUT* it is also a new and distinct data type. It follows all of the rules we applied to our other data types. It has a set size (128-bytes). It requires contiguous bytes of storage.

As with other data types, we *MUST* associate it with a variable in order to use it.

Using Structured Data Types

Just like all other basic data types, in order to access the data stored in at a location, we *must* associate a variable name with it. It does not make sense to merely issue the commands:

C/C++ Code 7.2

```
int main()
{  int;
   float;
   char;
    o
    o
```

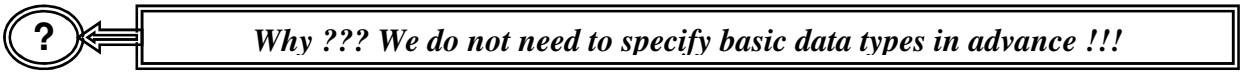
We just as we did, for example, when we made the statements: **int** *i;* **char** *myarray*[10]; **float** *fnumber*; With these declarations, we associated a variable name, or location in RAM with the type of data which was stored there. We must do the same with our new data type, **struct student**.

C/C++ Code 7.3

```
struct student
{    char ssn[10];
     char name[31];
     char street[41];
     char city[26];
     char state[3];
     char zip[6];
     char class;
     float gpa;
     int hrs;
     float balance;
}
int main()
(    struct student studentrecord;
```

In this case, we have associated our data type **struct student** with the variable name (location RAM) *studentrecord*. If we found out that the address *studentrecord* was 5000, and we go to that location, we expect to interpret the data we find there according to the data type **struct student**. On the first 10-bytes (5000-5009) we will find 10 characters (numeric data of type char), on the next 31-bytes (5010-5040), we will also find 31 characters, and so forth. For each component, we know how to interpret the bit patterns we find.

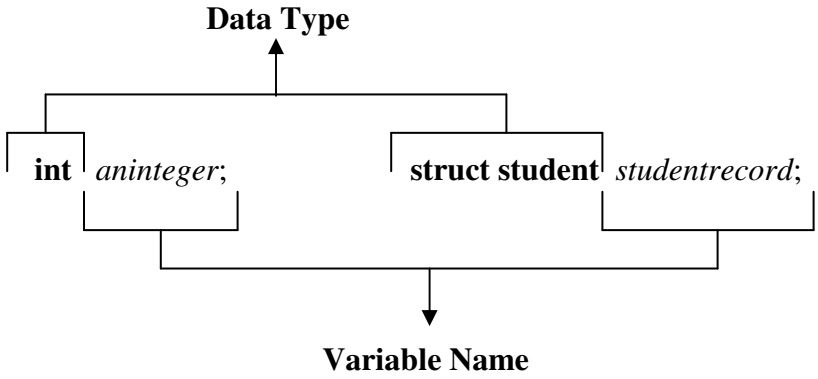
The only difference is that we must define our structure template *BEFORE* the **main** function.



All of the basic data type are *built-in* to the C/C++ programming language (they are, in fact, reserved words). How we are to interpret them is known in advance. Because we are creating a new data type, we must let the compiler know how to interpret the data *in advance*, so that the data will be stored as we wish it to be. We also must let the compiler know this information in advance so that it can check for correct usage *before* creating the compiled program.

For some reason, students initially find this notation confusing. Perhaps it is because the data type **struct student** requires 2 words to represent (although, so does the data type **long**

double). It does follow all of the syntactical rules associated with basic data types. Consider the following comparison:



C/C++ does allow for one additional notation which might (perhaps) make the use of structs a little easier:

C/C++ Code 7.4.

```
struct student
{    char ssn[10];
     char name[31];
     char street[41];
     char city[26];
     char state[3];
     char zip[6];
     char class;
     float gpa;
     int hrs;
     float balance; } studentrecord;
```

Which allows us to combine the definition of the structure template and the variable declaration.

As with any of the other data types, we can also initialize the contents of the structure (in our case, at location *studentrecord*) at the same time we make the variable declaration. Since we have two different ways of introducing a structure, we also have two different ways of initialializing (C/C++ code 7.5 and 7.6):

C/C++ Code 7.5

```
struct student
{    char ssn[10], name[31], street[41],  city[26], state[3], zip[6], class;
     float gpa;
     int hrs;
     float balance; }
int main()
(    struct student studentrecord = {"123456789","Smith, Mary","123 Main Street"
                        "New York","NY","10001",'A',
                        2.78, 98, 1234.50};
```

C/C++ Code 7.6.

```
struct student
{    char ssn[10], name[31], street[41],  city[26], state[3], zip[6], class;
     float gpa;
     int hrs;
     float balance; } studentrecord = {"123456789","Smith, Mary",
                          "123 Main Street", "New York","NY","10001",'A',
                          2.78, 98, 1234.50};
```

Does that mean that we can only use scalar variables with structs???

No. A struct is a data type, just like all of the other basic data types. As with the other data types, we can use them as building blocks to construct additional structures, for example, arrays.

Arrays of Structured Data Types

An array of structs is basically no different than an array of characters, integers, or real numbers. As with these types of arrays, the main advantage is that we can readily calculate the address of any element in the array quickly (although we need one additional step to calculate the address of any field within the record. The main disadvantage, as with all arrays, is that we require *contiguous* bytes of storage.

Consider the code given in 7.6.:

C/C++ Code 7.6.

```
#include <stdio.h>
#define CLASSSIZE 100          // Assume this is the maximum number of students
struct student                 // The student template
{   char   ssn[10];
    char   name[31];
    char   street[41];
    char   city [26];
    char   state[3];
    char   zip[6];
    char   class;
    float  gpa;
    int    hrs;
    float  balance;      };        // for a total of 128-bytes
int main()
struct student class[CLASSSIZE];   // an array of 100 students
```

We have established an *array of records* or, in relational data base terms, a *table*. We have also requested a total of 100 * 128 = 12,800 bytes of RAM starting at location *class*. Each record in the table will require 128 bytes of storage. Assuming we set values into the table, it might appear as:

Table 7.8.

Rec	*ssn*	*name*	*street*	*city*	*state*	*zip*	*Class*	*gpa*	*hrs*	*balance*
0	123456789	Clinton, H.	123 Main	Dallas	TX	12345	4	3.145	78	4762.89
1	987654321	Marino, D	4523 Ocean	Miami	FL	33100	3	2.206	65	0.00
2	234567890	Tolstoy, L.	750 Basil St.	Moscow	ID	66066	3	4.000	120	9878.99
3	876543210	Jordan, M.	89 So.Shore	Chicago	IL	11111	2	2.512	45	-8900.00
•	•	•	•	•	•	•	•	•	•	•
96	345678901	Bach, J.	13 Melody	Los Angeles	CA	79901	4	3.889	112	7685.89
97	765432109	Dali, S.	45 Surreal	New York	NY	10221	1	1.457	36	750.67
98	456789012	Turner, T.	2 Peachtreee	Atlanta	GA	43012	4	2.001	89	0.00
99	654321098	Gump,. F.	6 Fantasy	Birmingham	AL	23456	2	3.876	82	-500.00

Because each record contains the same number of bytes, we can readily determine the base address of each record, and, since we know how many bytes of storage each field in the record requires, we can likewise determine the base address of each field in the record. Again, assuming the base address of our data type **struct student** at location *class* is 5000, the associated base addresses for each field would be:

Table 7.9.

Rec	***ssn***	***name***	***street***	***city***	***state***	***zip***	***class***	***gpa***	***hrs***	***Balance***
0	5000	5010	5041	5082	5108	5111	5117	5118	5122	5124
1	5128	5138	5169	5210	5236	5239	5245	5246	5250	5252
2	5256	5266	5297	5338	5364	5367	5373	5374	5378	5380
3	5384	5394	5425	5466	5492	5495	5501	5502	5506	5508

• • • • • • • • • • • • • • • • • • • •

96	17288	17298	17329	17370	17396	17399	17405	17406	17410	17412
97	17416	17426	17457	17498	17524	17527	17533	17534	17538	17540
98	17544	17554	17785	17826	17852	17855	17861	17862	17866	17868
99	17672	17682	17713	17754	17780	17783	17789	17790	17794	17796

Which makes sense since we know that the array of structures takes up 12,800 bytes, meaning that it takes up locations 5,000 + (12,800 – 1) = 17,799 (notice that the field *balance*, in the last record, starts at location 17,796 and is of type **float**, meaning it takes up locations 17,796 through 17,799).

Calculating the base address of any record is (almost) as easy as calculating the address of any element in an array since we are dealing with an array (a 'superarray', if you will). The only difference is that since we are mixing the data types used in the record, we need to multiply our offset (here listed as record number, but in fact, since we are dealing with an array, the offset from the base address of the array) by the number of bytes required for each record (e.g., the base address of record 97 would be 5000 + 97 * 128 = 5000 + 12416 = 17416). Once we have determined the base address of any record, we can readily determine the base address of any field within that record, as we did previously.

The general formula applied, to calculate the base address of any field, is:

base address
+ offset * number of bytes per record
+ number bytes preceding the field

The c programming language provides us with some convenient notation to access records and fields within a record. For example, if we wished to find the zipcode for record (offset) 2 (actually the third record), we would use the notation:

class*[2].*zip

Once again, since *class* is an array of type **struct student** we can use an index (offset) to determine the base address of the record (5000 + 2 * 128 = 5256). Using the dot operator (.) after the record allows us to determine the base address of the individual field, based on the specifications we established in the structure template (e.g., the field *zip* can be found 10 + 31 + 41 + 26 + 3 = 111 bytes from the base address for the record, or in our example, at address 5256 + 111 = 5367).

Because zip is a character array, we could also have used the notation: **class[2].zip[1]**

In this case we would be pointing to the address 5367 + 1 = 5368, which contains the character '2' (or more precisely, the numeric value 50).

Let's take a look at how we might actually use a **struct**. In the following example (C code 7.7.), we will create a (simple) table which will hold three records, each with the fields *account_no* (an integer), *name* (a character array of length 15), and *balance* (a float). To demonstrate different methods for entering data into the struct, we will initialize the first two records when we declare, and input the third from the keyboard. Once we have all the data, we will search the **struct** for *balances* which exceed $200, and print out the information we find.

For this program, for each record (**struct**) we need :

```
struct account
   {  int account_no;                    2-bytes
      char name[15];                     15-bytes
      float balance; };                  4-bytes
                                         ---------
                                         21-bytes per record
```

And since we are declaring an array of three records (**struct account** *customer*[3]) at location customer, we are requesting a total of 3 * 21 = 63 contiguous bytes of storage. If the base address of customer were 7500, the addresses associated with each field would be:

Table 7.10.

Record/Offset	Acccount_no	name	balance
0	7500	7502	7517
1	7521	7523	7538
2	7542	7544	7559

C/C++ Code 7.3.

```
#include <stdio.h>                          // For input/output functions
#include <stdlib.h>                         // for atoi and atof

struct account                              // the structure template
{   int account_no;                         // customer account number
    char name[15];                          // customer name
    float balance; };                       // customer balance

int main()
{  struct account customer[3] =             // our array of records, the first 2 here
    {{2340, "Jones, Mary", 432.23}{1234, “Smith, John”, 25.00}};
    int index;                              // the array index/offset
    char entry[20];                         // a temporary string

/*  get the third record from the keyboard */
     printf("Enter customer id: ");         // Prompt for ID
    gets(entry);                            // assume we enter '2010'
    customer[2].account_no = atoi(entry);   // convert to integer
    printf("Enter customer name: ");        // Prompt for Name
    gets(customer[1].name);                 // assume we enter "Rodriguez, J"
    printf("Enter Balance: ");              // Prompt for balance
    gets(entry);                            // assume we enter "245.34"
    customer[1].balance = atof(entry);      // convert to float

    for (index = 0; index < 3; index++)     // Now search the array
       if (customer[index].balance > 200)   // and print if balance > 200
          printf("Name: %16s Account: %7d Balance: %10.2f\n",
                  customer[index].name, customer[index].account_no,
                  customer[index].balance);

/* The above print statement will produce the output:

                Name: Rodriguez, J.    Account: 2010  Balance:    245.34          */

 }
```

Again, Assuming a base address of 7500, the relevant portion of RAM might appear as:

Table 7.11.

7500	7501	7502	7503	7504	7505	7506	7507	7508	7509	7510	7511	7512	7513
2340		**'J'**	**'o'**	**'n'**	**'e'**	**'s'**	**','**	**' '**	**'M'**	**'a'**	**'r'**	**'y'**	**'\0'**
7514	7515	7516	7517	7518	7519	7520	7521	7522	7523	7524	7525	7526	7527
---	**---**	**---**	**432.23**				**1234**		**'S'**	**'m'**	**'i'**	**'t'**	**'h'**
7528	7529	7530	7531	7532	7533	7534	7535	7536	7537	7538	7539	7540	7541
','	**' '**	**'J'**	**'o'**	**'h'**	**'n'**	**'\0'**	**---**	**---**	**---**	**25.00**			
7542	7543	7544	7545	7546	7547	7548	7549	7550	7551	7552	7553	7554	7555
2010		**'R'**	**'o'**	**'d'**	**'r'**	**'i'**	**'g'**	**'u'**	**'e'**	**'z'**	**','**	**' '**	**'J'**
7556	7557	7558	7559	7560	7561	7562							
'\0'	**---**	**---**	**245.34**										

Or, once again, on a more primitive level:

Table 7.12.

7500	7501	7502	7503	7504	7505	7506	7507	7508	7509	7510	7511	7512	7513
00001001	**00100100**	**01001010**	**01101111**	**01101110**	**01100101**	**01110011**	**00101100**	**00100000**	**01001101**	**01100001**	**01110010**	**01111001**	**00000000**
7514	7515	7516	7517	7518	7519	7520	7521	7522	7523	7524	7525	7526	7527
01101111	**00100000**	**00100100**	**00111101**	**11010111**	**10101001**	**11010111**	**00000100**	**11010010**	**01010011**	**01101101**	**01101001**	**01110100**	**01101000**
7528	7529	7530	7531	7532	7533	7534	7535	7536	7537	7538	7539	7540	7541
00101100	**00100000**	**01001010**	**01101111**	**01101001**	**01101110**	**00000000**	**01100101**	**01101111**	**01111010**	**01100101**	**01110011**	**01110010**	**01111001**
7542	7543	7544	7545	7546	7547	7548	7549	7550	7551	7552	7553	7554	7555
00000111	**11011010**	**01010010**	**01101111**	**01100100**	**01110010**	**01101001**	**01100111**	**01110101**	**01100101**	**1111010**	**00101100**	**00100000**	**01001010**
7556	7557	7558	7559	7560	7561	7562							
00000000	**01101111**	**00000100**	**00111101**	**11010111**	**01100101**	**01100101**							

Pointers and Structured Data Types

As with other data types and structures, we can access elements in a **struct** through the use of pointers. As we shall see in later chapters, associating a pointer with a structured data object is not only useful, it is necessary. Without pointers, we would not be able to construct any of the data structures we will discuss in the following sections.

As with pointers to basic data types, we need to know only a few details:

1. The base address of the data type
2. The type of data which we would find at that location.

Consider the table we established using C Code 7.7. The structured data object we created was:

```
struct account
   {   int account_no;          2-bytes
       char name[15];           15-bytes
       float balance; };        4-bytes
                                ---------
                                21-bytes per record
```

and we associated the data type **struct account** with the location *customer*, which was an array of 3 elements. The code applied constructed the table:

Table 7.13.

Offset	*account_no*	*name*	*balance*
0	2340	Jones, Mary	432.23
1	1234	Smith, John	25.00
2	2019	Rodriguez, J.	245.34

We know that if we wished to print the table, we would apply the code:

C/C++ Code 7.8.

```
for (index = 0; index < 3; index++)
    printf("Name: %16s Account: %7d Balance: %10.2f\n",
            customer[index].name, customer[index].account_no,  customer[index].balance);
```

In function main, we could have added the pointer: **struct account** *cust; This declaration request 4-bytes of contiguous storage at location *cust*. Further, it tells us that if we go to location *cust*, we will find the data type **struct account**, on 21-contiguous bytes, laid out such that:

The first 2-bytes will contain a (signed) integer
The next 15-bytes will contain a string
The remaining 4-bytes will contain a string

We could therefore print out the table using the alternative code:

C Code 7.9.

```
cust = customer;
while (cust <= &customer[2])
{    printf("Name: %16s Account: %7d Balance: %10.2f\n",
              cust ->name, cust->account_no,  cust->balance);
    cust++;
}
```

How does this work???

Let's consider the program line-by-line. We already know how the data table will be stored (see Tables 7.11. and 7.12.). When we make the declaration:

struct account **cust*;

we are asking for 4-bytes of RAM at location *cust*. Let's assume that *cust* will be assigned location 7572. Given the base address of 7500 for our array of **struct account** *customer*, after the command:

cust = *customer*;

Table 7.14.

7572	7573	7574	7575
7500			

The address 7500 will be stored at location *cust* (address 7572). Our conditional check:

while (*cust* <= *&customer*[2])

Checks to see if the address stored at location *cust* is less than or equal to:

7500 + 2 * 21 = 7500 + 42 = 7542

Which is the base address of our third (*customer*[2]) record. Since it is (in this case), we will print out the data requested:

printf("Name: %16s Account: %7d Balance: %10.2f\n", *cust ->name*, *cust->account_no*, *cust->balance*);

There is only one notation in this expression which might need some explanation. The symbol **->** used, for example in the statement *cust***->***name*, is actually a redirect operator (which we have seen previously). It instructs the program to go to the address stored at location *cust* (in this case, 7500) and to print out the contents of field *name* as a string on field of 16 places.

The next statement: *cust++;*

Increments the contents of location *cust* by 21 such that location cust would appear as:

Table 7.15.

7572	7573	7574	7575
7521			

Which is the base address of our second record, *customer*[1].

Why are the contents of location cust incremented by 21-bytes???

Because our data type **struct account** contains 21-bytes. It is no different that incrementing a pointer to data type int by 2, or a pointer to data type float by 4. We have created a new data type which just happens to contain 21-bytes. If we were to issue the command:

printf("The data type struct account contains %d bytes\n", (**sizeof**) *customer*[1]);

We would see the output:

The data type struct account contains 21 bytes

The next time we increment the contents of location *cust* (*cust*++;) it will become 7542, which is the base address of *customer*[2], and we will print out the requested information. After that, the contents of location *cust* will be incremented by 21-bytes, meaning that it will contain the address 7563. In this case, when we make our conditional check (**while** (*cust* <= *&customer*[2]), where *&customer*[2] is 7542), the result will be false, and we stop printing.

Basically, that is true. Let's take a more relevant usage of pointers. Consider the following table:

Table 7.16.

Offset	*employee_name*	*SSN*	*title*	*department*	*depart_rm*	*department_head*
0	Christie, Agatha	012-34-5678	Copy Editor	Management	319	Dostoyevsky
1	Hesse, Herman	123-45-6789	Programmer	Info. Systems	205	Shakespeare
2	Eliot, T.S.	234-56-7890	Accountant	Accounting	456	Goethe
3	Carroll, Lewis	345-67-8901	Systems Analyst	Info. Systems	205	Shakespeare
4	Beckett, Samuel	456-78-9012	Coordinator	Management	319	Dostoyevsky

This table violates even the most simplistic rules: It is NOT in 1st normal form (i.e., it has repeating groups). All the information about an employee's department is repetitious: the fields *department*, *depart_location*, and *department_head*, are best placed in a separate table and a foreign key placed in the employee's table.

We would be better off with the following two tables:

Table 7.17.

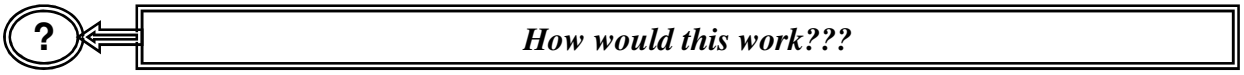
Offset	*employee_name*	*SSN*	*title*	*department*
0	Christie, Agatha	012-34-5678	Copy Editor	7800
1	Hesse, Herman	123-45-6789	Programmer	7765
2	Eliot, T.S.	234-56-7890	Accountant	7730
3	Carroll, Lewis	345-67-8901	Analyst	7765
4	Beckett, Samuel	456-78-9012	Coordinator	7800

Where *department* contains the RAM address where we will find the following information:

Table 7.18.

Offset	*depart_name*	*depart_ rm*	*depart_head*
0	Accounting	456	Goethe
1	Info. Systems	205	Shakespeare
2	Management	319	Dostoyevsky

Base address
7730
7765
7800



First of all, let's look at our *two* structured objects (call them *emp_record* and *dept_record*):

```
struct emp_record
{     char employee_name[18];                          18-bytes
      char ssn[10];                                    10-bytes
      char title[12];                                  12-bytes
      struct dept_record * department; };               4-bytes
                                                       ---------
                                                       44-bytes
struct dept_record
{     char dept_name[15];                              15-bytes
      int dept_room;                                    2-bytes
      char dept_head[18];   };                         18-bytes
                                                       ---------
                                                       35-bytes
```

The major point to note is that in our structured object *emp_record* we include a pointer field (department, of data type **struct dept_record**) which will contain an address which, if we were to go to, we would find all of the information needed about the employee's department. In function **main**, we might make the declaration:

C/C++ Code 7.10..

```
void main()
{   struct emp_record customer[5];
    struct dept_record department[3];
```

Let' assume that 44 * 5 = 220 contiguous bytes for our array *customer* are assigned starting at the base address 7500. Let's also assume that 35 * 3 = 105 contiguous bytes for our array *department* are assigned starting at the base address 7730. Notice that in this case, we have initialized the arrays, merely reserved a total of 220 + 105 = 325 bytes of storage.

Yes, in much the same manner as we did earlier. Consider the following C code, which would set in the data values as given in tables 7.17. and 7.18 (except for the pointer field in data structure **struct emp_record**):

C/C++ Code 7.11.

```
void main()
{    struct emp_record customer[5] =
     {{"Christie, Agatha", "012345678", "Copy Editor",,},{"Hesse, Herman", "123456789",
        "Programmer",,},{"Eliot, T.S.","234567890","Accountant",,}, {"Carroll, Lewis",
        "345678901","Analyst",,},{"Beckett, Samuel","456789012","Coordinator",,}};
     struct dept_record department[3]=
     {{"Accounting", 456, "Goethe"},{"Info. Systems", 205, "Shakespeare"},
      {"Management", 319, "Dostoyevsky"}};
```

Table 7.19.

7500	7501	7502	7503	7504	7505	7506	7507	7508	7509	7510	7511	7512	7513	7514	7515	7516
'C'	**'h'**	**'r'**	**'i'**	**'s'**	**'t'**	**'i'**	**'e'**	**','**	**' '**	**'A'**	**'g'**	**'a'**	**'t'**	**'h'**	**'a'**	**'\0'**
7517	7518	7519	7520	7521	7522	7523	7524	7525	7526	7527	7528	7529	7530	7531	7532	7533
---	**'0'**	**'1'**	**'2'**	**'3'**	**'4'**	**'5'**	**'6'**	**'7'**	**'8'**	**'\0'**	**'C'**	**'o'**	**'p'**	**'y'**	**' '**	**'E'**
7534	7535	7536	7537	7538	7539	7540	7541	7542	7543	7544	7545	7546	7547	7548	7549	7550
'd'	**'i'**	**'t'**	**'o'**	**'r'**	**'/0'**					**'H'**	**'e'**	**'s'**	**'s'**	**'e'**	**','**	**' '**
7551	7552	7553	7554	7555	7556	7557	7558	7559	7560	7561	7562	7563	7564	7565	7566	7567
'H'	**'e'**	**'r'**	**'m'**	**'a'**	**'n'**	**'\0'**	**---**	**---**	**---**	**---**	**'1'**	**'2'**	**'3'**	**'4'**	**'5'**	**'6'**
7568	7569	7570	7571	7572	7573	7574	7575	7576	7577	7578	7579	7580	7581	7582	7583	7584
'7'	**'8'**	**'9'**	**'\0'**	**'P'**	**'r'**	**'o'**	**'g'**	**'r'**	**'a'**	**'m'**	**'m'**	**'e'**	**'r'**	**'\0'**	**---**	
7585	7586	7587	•	•	•	•	•	•	•	•	•	7676	7677	7678	7679	7680
			•	•	•	•	•	•	•	•	•	**'B'**	**'e'**	**'c'**	**'k'**	**'e'**
7681	7682	7683	7684	7685	7686	7687	7688	7689	7690	7691	7692	7693	7694	7695	7696	7697
't'	**'t'**	**','**	**' '**	**'S'**	**'a'**	**'m'**	**'u'**	**'e'**	**'l'**	**'\0'**	**---**	**---**	**'4'**	**'5'**	**'6'**	**'7'**
7698	7699	7700	7701	7702	7703	7704	7705	7706	7707	7708	7709	7710	7711	7712	7713	7714
'8'	**'9'**	**'0'**	**'1'**	**'2'**	**'\0'**	**'C'**	**'o'**	**'o'**	**'d'**	**'i'**	**'n'**	**'a'**	**'t'**	**'o'**	**'r'**	**'\0'**
7715	7716	7717	7718	7719												
---	**'9'**	**'0'**	**'1'**	**'2'**												

For our array *department* are (with a base address of 7730), RAM might appear as:

Table 7.20.

7730	7731	7732	7733	7734	7735	7736	7737	7738	7739	7740	7741	7742	7743	7744	7745
'A'	**'c'**	**'c'**	**'o'**	**'u'**	**'n'**	**'t'**	**'i'**	**'n'**	**'g'**	**'\0'**	**---**	**---**	**---**	**---**	**4**
7746	7747	7748	7749	7750	7751	7752	7753	7754	7755	7756	7757	7758	7759	7760	7761
56	**'G'**	**'o'**	**'e'**	**'t'**	**'h'**	**'e'**	**'\0'**	**---**	**---**	**---**	**---**	**---**	**---**	**---**	**---**
7762	7763	7764	7765	7766	7767	7768	7769	7770	7771	7772	7773	7774	7775	7776	7777
---	**---**	**---**	**'I'**	**'n'**	**'f'**	**'o'**	**'.'**	**' '**	**'S'**	**'y'**	**'s'**	**'t'**	**'e'**	**'m'**	**'s'**
7778	7779	7780	7781	7782	7783	7784	7785	7786	7787	7788	7789	7790	7791	7792	7793
'\0'	**---**	**205**		**'S'**	**'h'**	**'a'**	**'k'**	**'e'**	**'s'**	**'p'**	**'e'**	**'a'**	**'r'**	**'e'**	**'\0'**
7794	7795	7796	7797	7798	7799	7800	7801	7802	7803	7804	7805	7806	7807	7808	7809
---	**---**	**---**	**---**	**---**	**---**	**'M'**	**'a'**	**'n'**	**'a'**	**'g'**	**'e'**	**'m'**	**'e'**	**'n'**	**'t'**
7810	7811	7812	7813	7814	7815	7816	7817	7818	7819	7820	7821	7822	7823	7824	7825
'0\'	**---**	**---**	**---**	**---**	**319**		**'D'**	**'o'**	**'s'**	**'t'**	**'o'**	**'y'**	**'e'**	**'v'**	**'s'**
7826	7827	7828	7829	7830	7831	7832	7833	7834							
'k'	**'y'**	**'\0'**	**---**	**---**	**---**	**---**	**---**	**---**							

In terms of space required and readability, yes. When we tried to put all of the information in the one table (Table 7.16) we needed a total of:

char *employee_name*[18];	18-bytes
char *ssn*[10];	10-bytes
char *title*[12];	12-bytes
char *dept_name*[15];	15-bytes
int *dept_room*;	2-bytes
char *dept_head*[18];	18-bytes
	75-bytes per record

or 5 * 75 = 375 bytes of contiguous storage. When we used two tables, we needed a total of 220 + 105 or 325 bytes of storage, and two blocks of 220 and 105 contiguous bytes. Further, this is a simplified example. Suppose we had 800 employees and a total of 10 departments. If we were to try and store all of the information in one table, we would require a total of 800 * 75 = 60,000 contiguous bytes of storage. Using two tables, we would require 800 * 44 = 35,200 contiguous bytes (for table employee), and 10 * 35 = 350 contiguous bytes (for table department), or a total of 35,550 bytes (59% of what we need for one table).

Manipulation of the two tables is also relatively simple. If, for example, we wished to print out the two tables (so that it would appear as it does in Table 7.16.) we could use the following program:

C/C++ Code 7.12.

```
void main()
{    struct emp_record customer[5];
     struct dept_record department[3];      // assume that we have already inputted all the data
     int i;
     for (i = 0; i < 5; i++)
        printf("%18s  %10s  %12s  %15s  %5d  %18s\n", employee[i].employee_name,
                 employee[i].ssn, employee[i].title, employee[i].department -> dept_name,
                 employee[i].department -> dept_room, employee[i].department -> dept_chair);
}
```

Once again, the only notation which might appear initially confusing is the redirection using pointers. Let's follow the pointers and how they are manipulated in each pass:

Table 7.21.

i	*employee[i].department*	*employee[i].department -> dept_name*	*employee[i].department -> dept_room*	*employee[i].department -> dept_chair*
0	7800	Management	319	Dostoyevsky
1	7765	Info. Systems	205	Shakespeare
2	7330	Accounting	456	Goethe
3	7765	Info. Systems	205	Shakespeare
4	7800	Management	319	Dostoyevsky

Are there any other uses of pointers in structured data objects???

Yes, many. In the next section, after a brief discussion on searching and sorting (which will in part also use pointers) we will see pointers in linked lists. In fact, from that point onward, we will find that without the use of pointers, we could not possibly be able to construct any abstract data types we will discuss.

Summary

In chapters 4 and 5, both of which dealt with arrays, we were forced to operate under some restrictions: the data structures all had a ***fixed*** number of ***contiguous*** storage elements ***all of the same data type***. In this chapter, we were able to remove the constraint that the data object ***all be of the same data type***.

Structured data objects (**structs**) are an extremely useful abstract data type. Without the concepts advanced, there would be no such things as databases (at least not without extreme measures applied to manipulate the data contained in each of the records). Further, **structs** will be used in each of the following data types which we will examine in the rest of the text. This is especially true for linked lists and hierarchical data types (trees).

This chapter also went into more details about the use of pointers. Combining pointers in a structured data object results in a powerful combination. Placing pointers as a field in a structured data object allows us to use dynamic memory allocation, which will later allow us to construct a variety of tree.

Chapter Terminology: be able to fully describe these terms

& (before a variable name)
* (before a variable name)
->
address offset
arrays of structs
base address
calculation of field address
calculation of record address
contiguous storage
dot notation
field
pointer
pointer field
redirection
sizeof operator
struct
structure template
structured data object
table

Review Questions

1. What advantages do an array of data type **struct** have over a regular array?? Explain

2. It was stated that by declaring a structured data object, say **struct newobject** { …} we are creating a new data type. Explain.

3. How many bytes of contiguous storage would be required for the array *mydata* given the following structure template and declaration:

```
struct mystuff
{   char classification, title[13];
    int number1, number2, numbers[42];
    float fnumb1, fnumb2[7];
    double doub1[3];
    struct mystuff * mypointer;  };
int main()
{   struct mystuff mydata[12];
```

4. Given:
```
struct datatemplate
{   char group[3];
    int groupclass;
    struct datatemplate *next;  };
char main()
{   struct datatemplate data[3] =  {  {"AB",76,6741},{"X4",322,6732},
                                      {"6T",0,6750}};
```

   If we issue the statement: *printf*("%lu", *data*)
   And we receive the output: **6732**

   A. Show how the data would be laid out in RAM (in Decimal or in Roman Characters)
   B What would be printed by the statement: *printf*("%d", (**sizeof**) *data*);
   C. What would be printed by the statement: *printf*("%lu", &*data*[1]);
   D. What would be printed by the statement: *printf*("%lu", &*data*[2].*next*);
   E. What would be printed by the statement: *printf*("%d", *data*[1].*groupclass*);
   F. What would be printed by the statement: *printf*("%s", *data*[2].*group*);
   G. What would be printed by the statement: *printf*("%d", *data*[0].*next*->*groupclass*);
   H. What would be printed by the statement: *printf*("%d", *data*[0].*next*->next->*group*);

Review Question Answers (NOTE: checking the answers before you have tried to answer the questions doesn't help you at all)

1. What advantages do an array of data type **struct** have over a regular array?? Explain

   The main advantage is that the data elements in a structured data object (struct) need not all be of the same data type.

2. It was stated that by declaring a structured data object, say **struct newobject** { …} we are creating a new data type. Explain.

   The basic adage *Give me an address and tell me what type of data is stored there, and I will tell you the value of that data type* still applies. After we have determined an address, say for struct newobject { …}, we are telling the compiler that if we go to that address, we will be able to interpret the data found and how we will interpret that data.

3. How many bytes of contiguous storage would be required for the array *mydata* given the following structure template and declaration:

```
struct mystuff
{   char classification, title[13];
    int number1, number2, numbers[42];
    float fnumb1, fnumb2[7];
    double doub1[3];
    struct mystuff * mypointer;  };
int main()
{   struct mystuff mydata[12];
```

Given:	**char**	***classification*,**	**//**	**1-byte**
		***title*[13];**	**//**	**13-bytes**
	int	***number1*,**	**//**	**2-bytes**
		***number2*,**	**//**	**2-bytes**
		***numbers*[42];**	**//**	**84-bytes**
	float	***fnumb1*,**	**//**	**4-bytes**
		***fnumb2*[7]**	**//**	**28-bytes**
	double	***doubl*[3];**	**//**	**24-bytes**
	struct mystuff	****mypointer*;**	**//**	**4-bytes**
				162-bytes

Then: ***mydata*[12] => 12 * 162 = 1,944 contiguous bytes of storage**

4. Given:
```
struct datatemplate
{   char group[3];
    int groupclass;
    struct datatemplate *next;  };
char main()
{   struct datatemplate data[3] =  {  {"AB",76,6741},{"X4",322,6732},
                                      {"6T",0,6750}};
```

If we issue the statement: *printf*("%lu", *data*)
And we receive the output: **6732**

A. Show how the data would be laid out in RAM (in Decimal or in Roman Characters)

Each record requires:	**char**	**group[3];**	**// 3-bytes**
	int	**groupclass;**	**// 2-bytes**
	struct datatemplate	***next;**	**// 4-bytes**
			9-bytes

If the base address of data is 6732, then each record and field has the address:

Offset	Group	groupclass	Next
0	6732	6735	6737
1	6741	6744	6746
2	6750	6753	6755

If we were to go to that address, we would find the values:

Offset	Group	groupclass	Next
0	"AB"	76	6741
1	"X4"	322	6732
2	"6T"	0	6750

Therefore, in RAM we would see:

6732	6733	6734	6735	6736	6737	6738	6739	6740	6741	6742	6743	6744	6745
'A'	**'B'**	**'\0'**	**76**		**6741**				**'X'**	**'4'**	**'\0'**	**322**	
6746	6747	6748	6749	6750	6751	6752	6753	6754	6755	6756	6757	6758	
67 32				**'6'**	**'T'**	**'\0'**	**0**		**6750**				

B What would be printed by the statement: *printf*("%d", (**sizeof**) *data*);

Each record requires 9-bytes of storage. Since there are 3 records, (sizeof) *data* would be 3 * 9 = 27-bytes.

C. What would be printed by the statement: *printf*("%lu", &*data*[1]);

From the above table, we can see that &*data*[1] is 6741

D. What would be printed by the statement: *printf*("%lu", &*data*[2].*next*);

From the above table, we can see that &*data*[2].*next* is 6750

E. What would be printed by the statement: *printf*("%d", *data*[1].*groupclass*);

From the above table, we can see that *data*[1].*groupclass* is 322

F. What would be printed by the statement: *printf*("%s", *data*[2].*group*);

From the above table, we can see that *data*[2].*group* is 6T

G. What would be printed by the statement: *printf*("%d", *data*[0].*next*->*groupclass*);

From the above table, we know that *data*[0].*next* is 6741. At that base address, we would find the second (offset 1) record. The contents of field *groupclass* for the second record is 322.

H. What would be printed by the statement: *printf*("%d", *data*[0].*next*->next->*group*);

From the above table, we know that *data*[0].*next* is 6741, which is record 2 (offset 1). At that base address (record 2, or offset 1), we find that the value of next is 6732, which is record 1 (offset 0). The value of group, for record 1, is "AB".