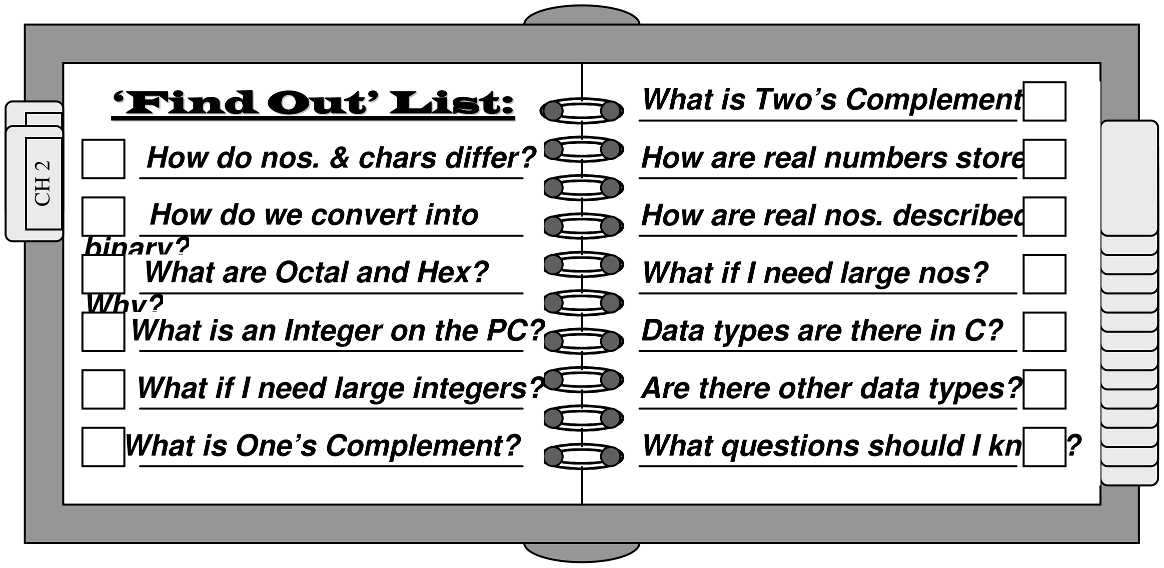
CHAPTER 11: DYNAMIC MEMORY ALLOCATION

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


'Find Out' List:
CH 2
How do nos. & chars differ?
How do we convert into binary?
What are Octal and Hex? Why?
What is an Integer on the PC?
What if I need large integers?
What is One's Complement?
What is Two's Complement
How are real numbers store
How are real nos. describec
What if I need large nos?
Data types are there in C?
Are there other data types?
What questions should I kn ?

Introduction

All of the abstract data types we have seen so far have been bridled by a major constraint: we needed a ***fixed*** number of ***contiguous*** blocks of RAM. We have been requesting RAM on a *static* basis. In other words, once we request a specified amount of storage, that space is reserved even before we actually save any data to it, and it can not be used by any other part of the program[1]. What we need to be able to do now is to request additional RAM on a *dynamic* basis, or on-the-run, as we require it.

Def

Allocating memory dynamically generally requires the data structures and techniques we have been discussing to date. Without the use of structured data objects and pointers, we would not be able to allocate memory dynamically. The idea of linked lists, and how to use linked lists to reduce search times contributes to efficiency of dynamic memory allocation. Used in conjunction, these structures and approaches allow us flexibility far beyond that possible with many of the older 3rd generation languages, such as COBOL and FORTRAN.

[1] Note that this is not meant to contradict our previous discussion of Static vs. External vs. Automatic variables. Any variable declared within a function, assuming it is automatic, for example, is still available only to that function (i.e., the variable name can only be used by that function). External variables can still be referred to from any function.

Dynamically Allocated Linked Lists

One of the main functions of the operating system is to keep track of available RAM. We have put an additional constraint on the operating system in that when we declare a data type (whether a basic data type, or a **struct** we have constructed), we are requesting that a contiguous number of bytes be reserved for each of the data types. That is, if we need to store an integer, we need 2-bytes of contiguous storage; to store a float, we require 4-bytes of contiguous storage; if we want to store a **struct**, we must specify, in advance (through our template), the total number of contiguous bytes we need.

These basic constraints (e.g., 2-bytes for the data type **int**, 4-bytes for a pointer) **can not** be eliminated or eased since basic data (whether defined by the c programming language or by us) can not be broken down.

We can, however, modify the manner in which we store arrays (the basic data structure we have been dealing with up to this point). Arrays are wonderful data structures because we can do so much with them (i.e., determine an element address quickly, sort them, perform very quick searches on them, link them, and so forth). Up to this point in time, we have relied on them to store data, and have considered techniques which utilize their basic structure. But we know that there is a basic problem with them: ***we must reserve all of the contiguous memory we need in advance.***

Suppose that we wished to set-up a database which contained 1000 simple records, say, of the structure :

C/C++ Code 11.1

```
struct custrec
     {  char   ssn[10];
        char   name[31];  }
```

Notice that the structure requires only 41-bytes of contiguous storage; if we were to associate this structure with an array (e.g., using the declaration **struct custrecrec** *customer*[1000]) we would require a total of 1000 * 41 = 41,000 bytes (about 40 KB) of contiguous storage. It is possible that we might get the runtime message:

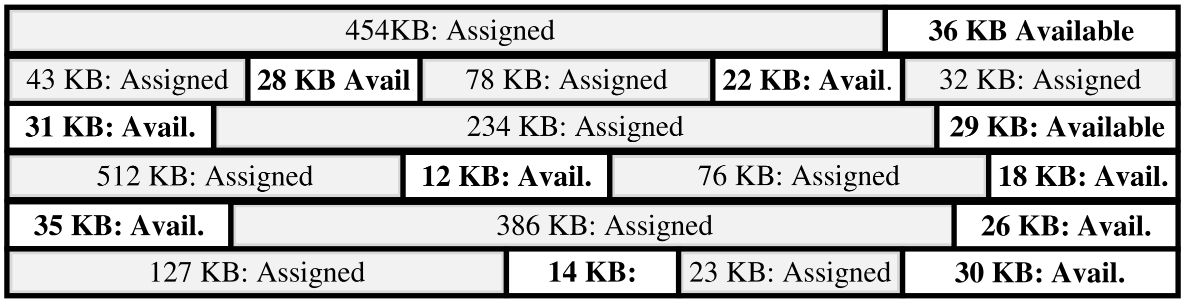
Not enough memory (or something similar).

However, if we were to check the amount of RAM available (we won't get into how to do that here), we might find that we actually have about 288,000 bytes (about 281 KB) of RAM available.

Why do we get the error message then?

While we might have enough RAM to store our database, we do not have enough ***contiguous*** bytes of RAM. If we could look inside our RAM, we might see:

Figure 11.1.



We do indeed have a total of 36 + 28 + 22 + 31 + 29 + 12 + 18 + 35 + 26 + 14 + 30 = 281KB of RAM available, **BUT** the largest contiguous block of memory available is only 36KB, insufficient for our array. We could, theoretically have 2 arrays (perhaps *customer1* and *customer2*), each containing 500 records. In this case, the largest contiguous block needed would be 41 * 500 = 20,500 bytes (about 20KB). However, if we split up our array we give up many of the advantages associated with arrays (e.g., a single variable name, sorting and search advantages).

We need to find a way to store the records in a non-contiguous fashion AND still be able to access them in an orderly fashion. Our discussion of linked lists gave us some ideas on how we could do this: *Through the use of pointers*.

In the previous chapter, when we linked our lists in some ordered fashion (e.g., alphabetically, by name), we used pointers which did not necessarily point to some adjacent record. Depending on the number of records in our array, the number of bytes required for each record, and the manner in which we established the list, we might have one record pointing to a record which was thousands of bytes from it.

Let’s rewrite the record structure to include a pointer field:

C/C++ Code 11.2.

```
struct custrec
{     char ssn[10], name[31];
      struct custrec * next; };
```

Nothing new here. As we did previously, we will use the pointer field to store the address of the next record. Each record will require 45 bytes of contiguous storage. The main difference is in our declaration:

C/C++ Code 11.3.

```
int main()
{     struct custrec * customer,  * first, * previous;
```

Notice that we are *NOT* associating the data type **struct custrec** with an array, but with three pointer variables. As could be anticipated, pointer *first* will hold the address of the first record on the linked list, and pointer customer will hold the address of the current record (the one we are examining). Pointer variable *previous* will hold the address of the last record we were examining (this will make more sense when we actually go through the procedure of setting up the linked list).

To set up the linked list, we will:

Establishing a Dynamically Linked List

1. Ask the operating system to find us 45 bytes of contiguous storage and return the base address, which we will store in location *customer*.
2. Store the data into the fields (*ssn* & *name*) we established for our data type **struct custrec**.
3. If this is the first record on the list, we will also store the address returned by the operating system in location *first* (*first* = *customer*).
4. If this is not the first record on the list, have the previous record's *next* field point to the address of the current record (*previous* -> *next* = *customer*).
5. If there are no more records to add to the list (i.e., the current record is the last record), set the current record's *next* field to NULL (*customer* -> *next* = NULL). Stop, we are done.
6. If there are more records to add to the list, then before getting the next one, store the address of the current record into location previous (*previous* = *customer*). Go to step 1.

It might look confusing, but it should make more sense when we go through the procedure step-by-step.

Establishing Dynamically Allocated Linked Lists

Let's assume that we wished to store information about the customers given in Table 11.1. Notice that the list is not sorted in any order (either by ssn or name). Right now, we will not concern ourselves with ordering the list by any field. Let's just set up the list so that "Grieg" will be the first record and will point to "Shumann", and so forth, until "Vivaldi" point to "Liszt", who point to no one (NULL).

Table 11.1.

ssn	*name*
551928772	Grieg, Eduard
732010233	Shumann, Robert
321100678	Beethoven, Ludwig
697467721	Vivaldi, Antonio
678946790	Liszt, Franz

Now, let's follow the procedure we outlined previously:

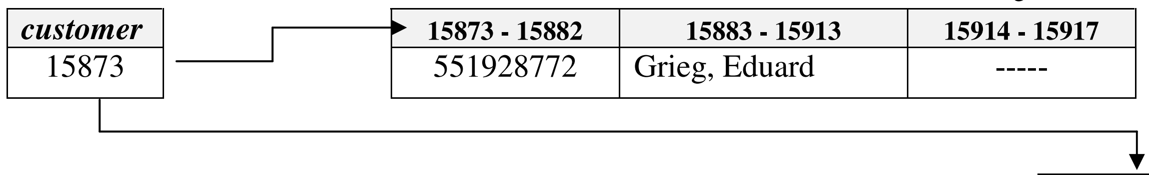
1. **Ask the operating system to find us 45 bytes of contiguous storage and return the base address, which we will store in location *customer*.**

Right now, let’s not worry about the commands necessary to do this (we will see those soon). Let’s just assume that we have done so, and the operating system returns the address 15873 (meaning addresses 15873 through 15917 are available; remember we need 45 contiguous bytes for our structure). This address will be stored in pointer variable *customer*.

2. **Store the data into the fields (*ssn* & *name*) we established for our data type struct custrec**

After we set in our first record, we would see the following We would now see the following layout:

Figure 11.2.



first
15873

3. **If this is the first record on the list, we will also store the address returned by the operating system in location *first* (*first* = *customer*).**

We can skip steps 4 and 5 right now, since they don’t apply.

previous
15873

6. **If there are more records to add to the list, then before getting the next one, store the address of the current record into location previous (*previous* = *customer*). Go to step 1.**

Variable *previous* now contains the address 15873.

1. **Ask the operating system to find us 45 bytes of contiguous storage and return the base address, which we will store in location *customer*.**

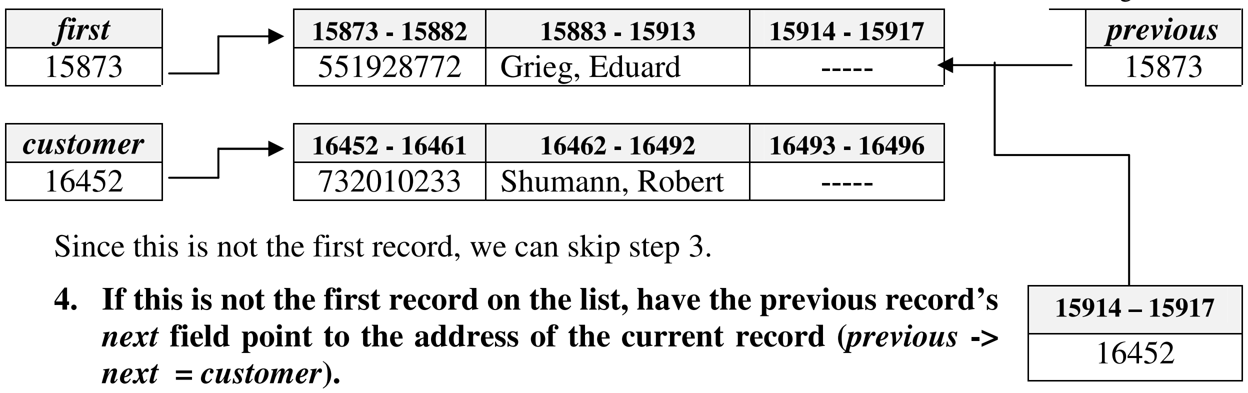
customer
16452

Assume that the address 16452 is returned.

2. **Store the data into the fields (*ssn* & *name*) we established for our data type struct custrec**

Our layout would now appear as (Figure 11.3):

Figure 11.3.

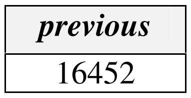


Since this is not the first record, we can skip step 3.

4. **If this is not the first record on the list, have the previous record's *next* field point to the address of the current record (*previous* -> *next* = *customer*).**

Skipping step 5, since there are more records to be added:

6. **If there are more records to add to the list, then before getting the next one, store the address of the current record into location previous (*previous* = *customer*). Go to step 1.**

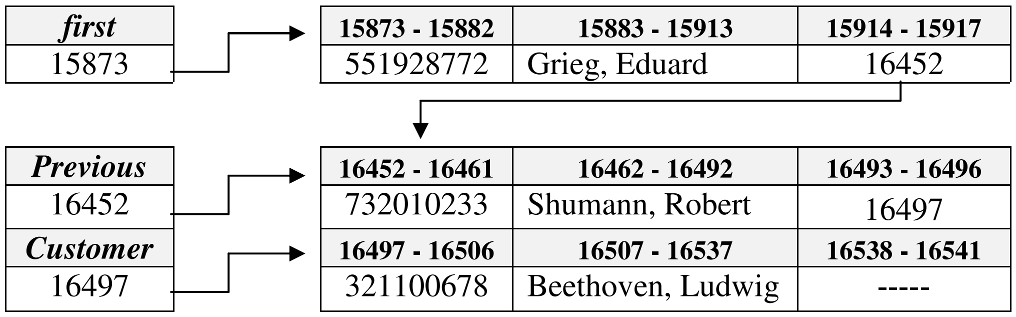


Variable *previous* now contains the address 16452.

Let's take a look at how RAM layout changes after each variable is added, without going through the individual steps.

<u>After Record #3 is Added:</u>

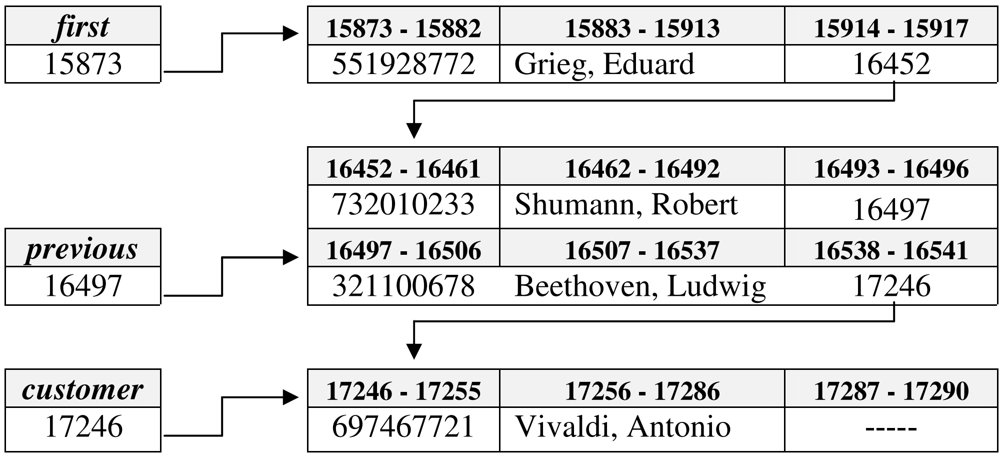
Figure 11.4.



Notice that if contiguous memory is available, it will be used.

After Record #4 is Added:

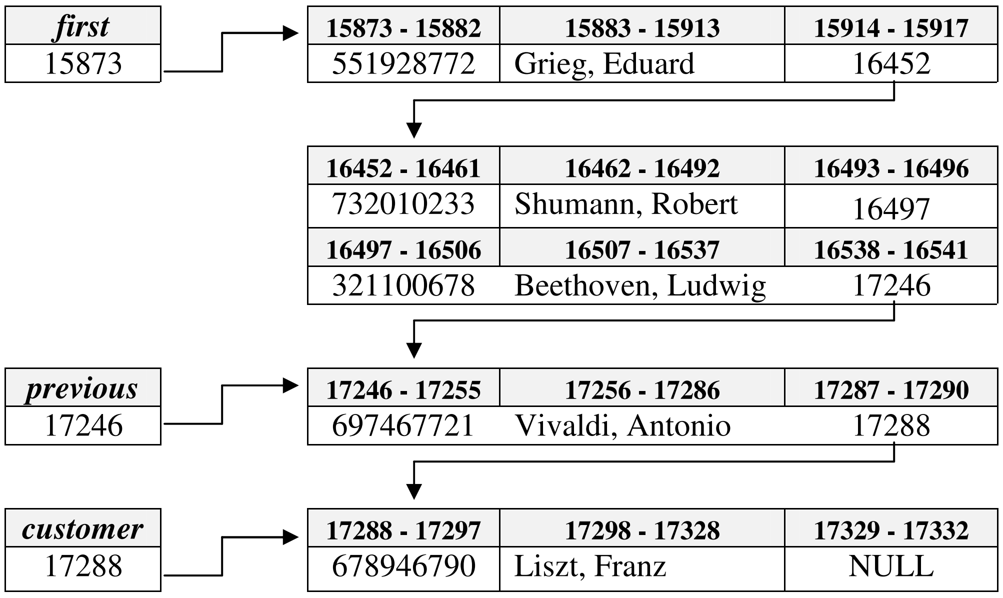
Figure 11.5.



Notice that in this case, addresses 16542 (through 16586, which we would for 1 record) were not available. The next block of 45 contiguous bytes of RAM was available starting at 17246 (and running through 17290).

After Record #5 (the final record) is Added:

Figure 11.6.



Let’s assume that the next block of 45 contiguous bytes of RAM was available starting at 17288 (and running through 17332). The only difference between how we added this record and how the others were added is that we stored a NULL address in the *next* field, meaning that the list is now complete.

How do allocate memory dynamically in C ???

Dynamic Memory Allocation in C

The c programming language uses a very simple command to request memory (on-the-run, if you will). Assuming we have already established our structure (as we did in C Code 11.2.), and that we have already made our variable declarations (as we did in C Code 11.3.), then to store the base address of our block of available memory (into pointer variable customer), we would issue the command:

C/C++Code 11.4.

```
customer = (struct custrec *) malloc (sizeof (struct custrec));
```

It looks complicated, but it really isn't. Let's look at the individual components in the command, starting with the innermost parentheses.

sizeof (struct custrec)

sizeof is a built-in (unary) operator which returns the number of bytes necessary for a certain data type. For example, let's assume that we had declared a variable called *intvar* of type **int**, *floatvar* of type **float**, and a variable called *doublevar* of type **double**. The command **sizeof**(*intvar*) would return the value 2, the command **sizeof**(*floatvar*) would return the value 4, and the command **sizeof**(*doublevar*) would return the value 8. These may seem obvious (since we know that integers require 2-bytes of storage, single precision real numbers require 4-bytes, and double precision real numbers require 8-bytes), but when it comes to data types that we develop, the number of bytes needed for storage varies. For structured data objects with many fields, we really don't want to calculate the number of bytes needed each time we ask for memory.

malloc

We Already Know:
<stdlib.h> also contains the functions atoi, atof, atol, itoa, ftoa, and ltoa

malloc (short for **m**emory **alloc**ation) is a function which is found in the header file <stdlib.h>, and performs one action: checks with the operating to find available memory. It takes one argument, the number of contiguous bytes needed, and returns the address of where that memory can be found. In short, it is the nucleus of our memory allocation activity. The **sizeof** operator that we discussed above determines the number of bytes we need, and passes the return value as a parameter to function malloc.

Since we know that our* struct custrec *requires 45 bytes of contiguous storage, could we have passed the integer value 45 to function* malloc*??

Yes, but as we already noted, we generally do not want to add up the number of bytes a structured data object requires. Letting the program determine the number of bytes needed also saves us time when we begin editing our code. Suppose we decide that we wish to add new fields (or delete existing ones), or change the way we store fields (from storing a field as an **int** to a **float**, for example). Using the **sizeof** operator let's us do so without also having to change the number of bytes we need for a data structure.

(**struct custrec** *)

This component does look a little strange, but mainly because of our use of parentheses. Suppose we saw the declaration **int** * *i*; By now, we should be quite used to this notation. We are asking for 4-bytes of storage at location *i*, where we will store an address at which we can find an integer. In C Code 11.3., when we used the expression **struct custrec** * *customer*; we were requesting 4 of contiguous memory at location *customer*, where we will store an address pointing to the data type **struct custrec**. The only difference here is that we will get an address at which we expect to find the data type **struct custrec** from function malloc (and store it at location *customer*) as we need it (not before we begin program execution).

What if there is not enough contiguous memory available??

It is possible, although not likely since we are only requesting 45-bytes for our data type. If there is not enough RAM available, function malloc will return a NULL pointer. We should always check to make sure that we do have a valid address for our data. This is relatively simple to do:

C Code 11.5.

```
customer = (struct custrec *) malloc (sizeof (struct custrec));
if (customer == NULL)
{      puts("Memory allocation failed - Bye!");
       return(0);  }  }
```

or, combining the commands:

C Code 11.6.

```
if ((customer = (struct custrec *) malloc (sizeof (struct custrec))) == NULL)
{      puts("Memory allocation failed - Bye!");
       return(0);  }  }
```

In either case, the program terminates (we are assuming **int main**()) since if we can't get the memory we need, we might as well terminate processing.

Does that mean that we could never run the program??

Not necessarily. We might have other programs (such as a word processor or spreadsheet) which we are not using at the present time, but are still taking up space in RAM. We could remove those. Or, we could release some the RAM we have reserved for our present program, but no longer need.

There is another function in the <stdlib.h> file called ***free*** which allows us to dispose of memory allocations as we need it. Suppose, for example, that variable *customer* contained the address of a record which we no longer needed. We could free-up the memory which that record is presently taking up (and make the 45-contiguous bytes that are presently reserved for it available for future usage) with the simple command:

C/C++ Code 11.7.

```
free(customer);
```

There are additional memory management functions available to us, but they go beyond what we need to know in this chapter.

What about the C code necessary to dynamically set up our list??

Establishing Dynamically Linked Lists in C

We are going to illustrate this program (C Code 11.8.) in a strange manner: We are going to first store the elements as an array of structured data objects, and then move them over to a (different) dynamically linked list.

Why??

Merely for the purposes of illustration. Normally, these records would be read from a data file (either ASCII or binary), or entered from the keyboard (we are certainly not using the advantages of dynamic memory allocation here). Rather than worry about the additional code necessary, we will just transfer them from one list to the other.

The code follows the procedure as given in Figures 11.2. through 11.6. We have also included some printf statements, so we can see how the addresses come into play. The actual output (for the run made; note that the addresses will vary each time the program is run) from the program is given in Program Output 11.1. Because we are concerned with the addresses, they are printed in **boldface**.

Looking at the output, it can be noted that the array element addresses (from the command printf("Record #%d is located at %lu\n",*i*, &*custs*[*i*]);) are as we might expect. Given a base address (&*custs*[0] == *custs*) for the array of 498863890, the next record (&*custs*[1]) would be found at 498863890 + 45 = 498863935), and so forth, through the last record (&custs[4] = 498863890 + 4 * 45 = 498863890 + 180 = 498864070).

Looking at the addresses for the linked list, however, we notice that the addresses are not at all predictable. Our first record (for “Grieg, Eduard”) is stored at 520028164 (21164050 addresses from the end of the last record in our array). The second address (for “Shumann, Robert”) starts at address 520290308 (262100 addresses from the end of the first record (at address 520028208). When we examine each of the linked list address, relative to the previous record’s address, we see that none of the records are contiguous.

C/C++ Code 11.8.

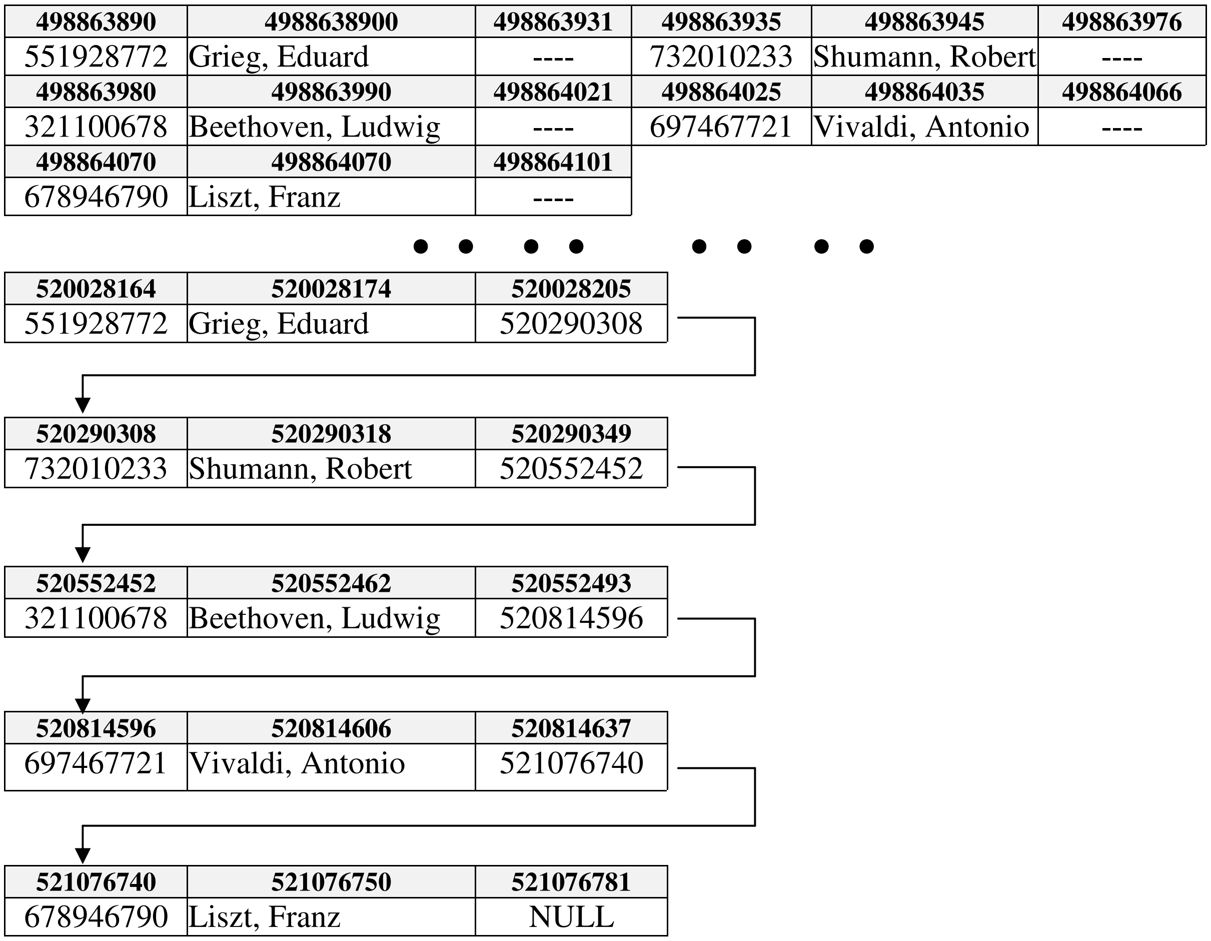
```
#include #include <stdio.h>
#include <stdlib.h>
#include <string.h>
struct custrec
{    char    ssn[10], name[31];
     struct custrec * next; }
int main()
{   struct custrec custs[5] = {{"551928772","Grieg, Eduard",}, {“732010233”,
             “Shumann, Robert”,}, {“321100678”,”Beethoven, Ludwig”,},{“697467721”,
             “Vivaldi, Antonio”,}, {“678946790”,”Liszt, Franz”,}};
    struct custrec * customer,  * first, * previous;
    int i;
    for (i = 0; i < 5; i++)         // print array information and transfer the records
    {  printf("Record #%d is located at %lu\n",i, &custs[i]);
/*     Get some memory if we can and store the address to it in variable customer        */
        if ((customer = (struct custrec *) malloc (sizeof(struct custrec))) == NULL)
        { puts("Memory Allocation Failed – Bye!\n");
           return 0; }
        strcpy(customer -> ssn, custs[i].ssn);   // move over customer ssn
        strcpy(customer -> name, custs[i].name);   // move over customer name
        if (i == 0)                                 // store the first dynamically
           first = customer;                        //     allocated address
        else
            previous -> next = present;             // old record points to the new one
        previous = customer;   }                    //get ready for a new address
    customer -> next = NULL;                        // the last record will point to NULL
    customer = first;                               // Start at the the top of the list
    while (customer != NULL)
    {   printf ("SSN: %12s Name: %15s is stored at at %lu and the next rec is at %lu\n",
                customer->ssn, customer->name, customer, customer->next);
        customer = customer -> next; }              // Get the next record in the list
    }
    return 0;                                       // We’re done
}
```

Program Output 11.1.

```
Record #0 is located at 498863890
Record #1 is located at 498863935
Record #2 is located at 498863980
Record #3 is located at 498864025
Record #4 is located at 498864070
SSN: 551928772  Name        Grieg, Eduard  at  520028164  next at  520290308
SSN: 732010233  Name:    Shumann, Robert   at  520290308  next at  520552452
SSN: 321100678  Name:  Beethoven, Ludwig   at  520552452  next at  520814596
SSN: 697467721  Name:    Vivaldi, Antonio  at  520814596  next at  521076740
SSN: 678946790  Name:        Liszt, Franz  at  521076740  next at  0
```

If we were to examine the way in which this data was stored in RAM, we would see (In order to save space, only the base address of each field is shown):

Figure 11.7.



498863890	4988638900	498863931	498863935	498863945	498863976
551928772	Grieg, Eduard	----	732010233	Shumann, Robert	----
498863980	**498863990**	**498864021**	**498864025**	**498864035**	**498864066**
321100678	Beethoven, Ludwig	----	697467721	Vivaldi, Antonio	----
498864070	**498864070**	**498864101**			
678946790	Liszt, Franz	----			

● ● ● ● ● ● ● ●

520028164	520028174	520028205
551928772	Grieg, Eduard	520290308

520290308	520290318	520290349
732010233	Shumann, Robert	520552452

520552452	520552462	520552493
321100678	Beethoven, Ludwig	520814596

520814596	520814606	520814637
697467721	Vivaldi, Antonio	521076740

521076740	521076750	521076781
678946790	Liszt, Franz	NULL

But the linked list is not in any order. How can it be ordered??

Ordering a Dynamically Linked List

We have already seen how to order a linked list (in Chapter 10). The rules have not changed. By way of review, let's order our dynamically linked list by social security number (field *ssn*). As we did before, we will get the first record and put it onto our list. Every time we get a new record, we will determine where it should be on the list, and insert accordingly(let's assume the same addresses for each record as above). The procedure to be followed would appear as follows:

Add the First Name Figure 11.8.

First
520028164

Address	*ssn*	*name*	*next*
520028164	551928772	Grieg, Eduard	NULL

The first Name we get will become the only name on our list. Therefore, we will also set our *first* pointer to the base address of the record, and have it point to NULL, since there are no more records on the list (so far).

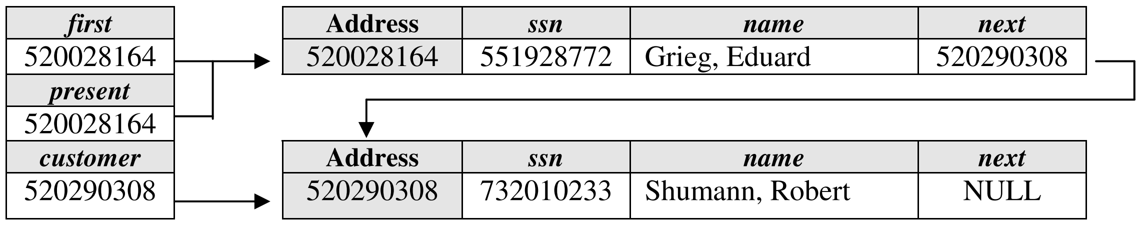
Next, as before, we dynamically allocate space for our next record, saving the address into variable *customer*, and move in the data we wish to store there. We then start comparing the *ssn* of our new (i.e., at location *customer*) record ("732010233") with the *ssn* of every record on the list, starting with the first *ssn* on the list. To do this, we will need one additional pointer variable (let's call it *present*). We will start by copying the address stored in variable *first* into location *present*. We then compare the information stored at the address contained in variable *customer* with the information stored at the address contained in variable *present*. There are two possibilities and two possible actions for each:

Record Insertion Possibilities

1. The new record (pointed to by *customer*) is smaller than the old record (pointed to by *present*)
    a. If the old record is the first record in the list (i.e., it has the same address as that contained in variable *first*), have the new record point to it and move the new record's address into variable *first*.
    b. If the old record is NOT the first record on the list, have the record which was previously pointing to the old record (i.e., the record whose base address we have been tracking in variable previous) point to the new record (move the contents of location customer into the *next* field of the record whose base address is stored in variable previous). Then have the *next* field of the new record point to the old record.
2. The new record is greater than the old record.
    a. If the old record is the last record on the list, have the old record point to the new record, and have the new record point to NULL.
    b. If the old record point to another record, have the new record point to that record. Have the old record then point to the new record.

In our case, we will (quickly) come to the end of the list, and have “Grieg, Eduard” point to “Shumann, Robert”, who will point to NULL.

Add the 2nd Name Figure 11.9.



first
520028164
present
520028164
customer
520290308

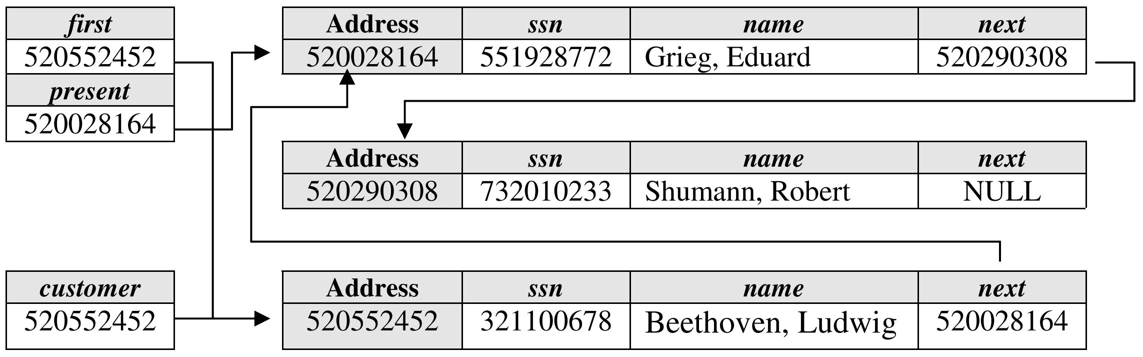
Address	***ssn***	***name***	***next***
520028164	551928772	Grieg, Eduard	520290308

Address	***ssn***	***name***	***next***
520290308	732010233	Shumann, Robert	NULL

Since we are becoming familiar with the procedure, we have included the value of (one) of the other pointers we must use. As we noted above, there is another pointer (previous), but right now we don’t need it, since the ordered list contained only one record above.

When we (dynamically) get our next record (“Beethoven, Ludwig”), storing the address we received in customer. We again start at the top of the list by moving the address from variable *first* into variable *present*. At the address contained at location *present*, we find that the associated ssn (“321100678”) is the smallest one in our linked list. We will therefore have “Beethoven, Ludwig” point to “Grieg, Eduard” **AND** reset our *first* pointer so that it points to “Beethoven, Ludwig”.

Add the 3rd Name Figure 11.10.



first
520552452
present
520028164

Address	***ssn***	***name***	***next***
520028164	551928772	Grieg, Eduard	520290308

Address	***ssn***	***name***	***next***
520290308	732010233	Shumann, Robert	NULL

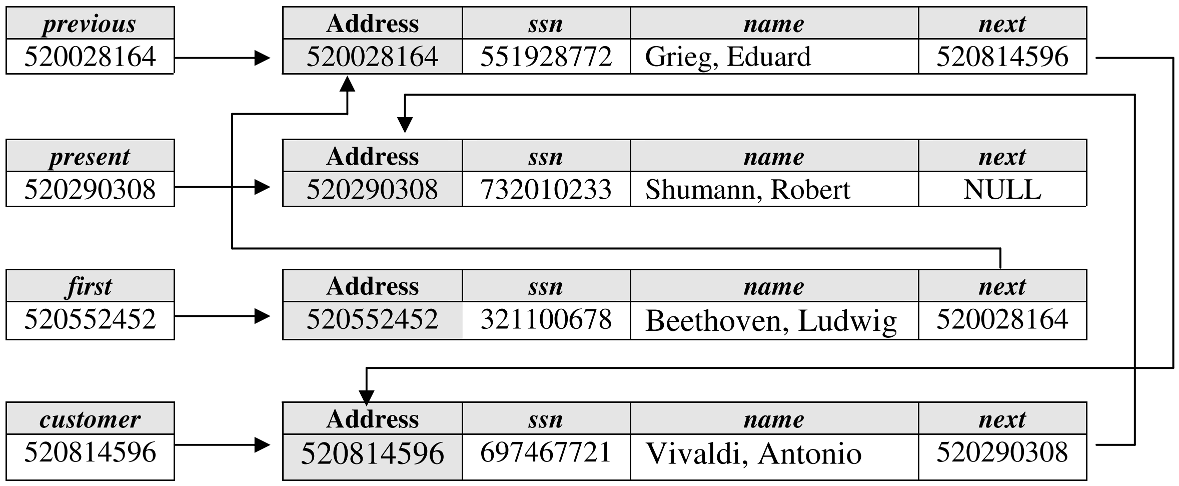
customer
520552452

Address	***ssn***	***name***	***next***
520552452	321100678	Beethoven, Ludwig	520028164

Repeating the process for the 4th record (“Vivaldi, Antonio”) we first get 45-bytes of RAM, store the relevant data at that location, and move the address for “Beethoven, Ludwig” (now the first record on our list) into variable *present* (*present* = *first*). Comparing Vivaldi’s *ssn* with Beethoven’s, we find that it is greater. Since Beethoven is not the last record on the list, we need to keep track of Beethoven’s address (since we might have to insert Vivaldi between Beethoven and whomever he points to) by copying it to variable previous (*previous* = *present*). We then get the next record on the list (*present* = *present* -> *next*), or “Grieg, Eduard”. Comparing Vivaldi’s ssn (“697467721”) with Grieg’s (“551928772”), we find that it is still greater, so we repeat the process above (*previous* will hold Grieg;s address, *present* will hold the next record’s address, or the address for “Shumann, Robert”).

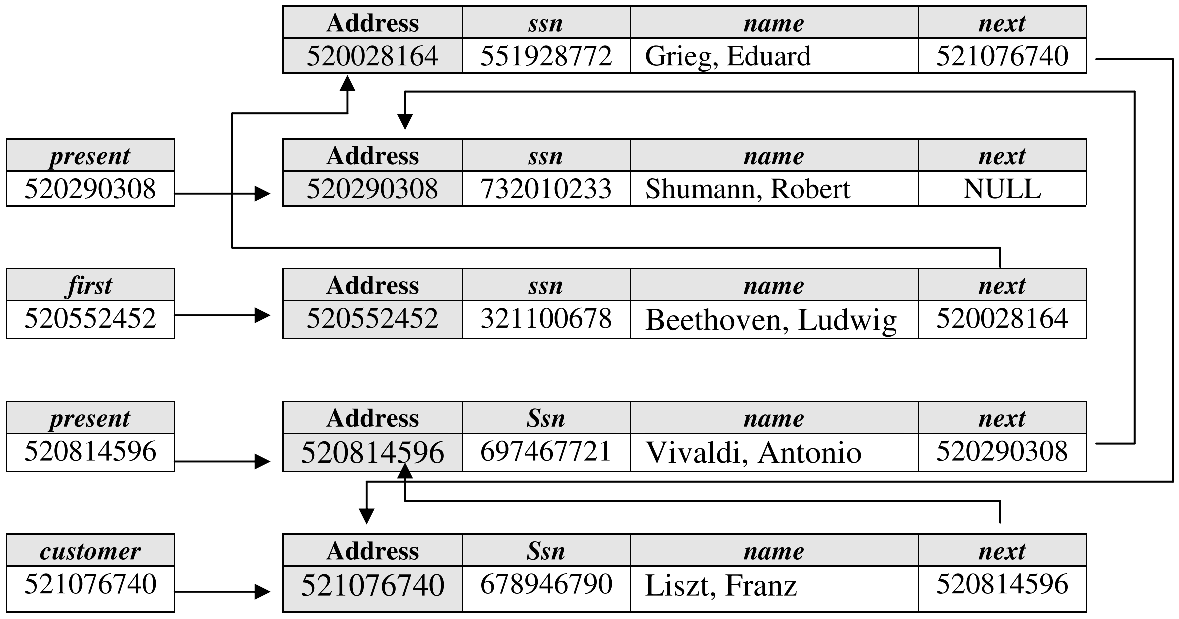
This time, when we comparing Vivaldi's *ssn* with Shumann's *ssn*, we find that Shumann's ("732010233") is greater. Therefore, we must insert Vivaldi between Grieg and Shumann. This is not a problem, since we already have Shumann's address (in variable *present*), and Grieg's address (in variable *previous*).

Add the 4th Name Figure 11.11.


previous
520028164
Address | ssn | name | next
520028164 | 551928772 | Grieg, Eduard | 520814596
present
520290308
Address | ssn | name | next
520290308 | 732010233 | Shumann, Robert | NULL
first
520552452
Address | ssn | name | next
520552452 | 321100678 | Beethoven, Ludwig | 520028164
customer
520814596
Address | ssn | name | next
520814596 | 697467721 | Vivaldi, Antonio | 520290308

There is only one more name (Liszt) to add (to be inserted between Vivaldi and Shumann). To do this, all we need is to have Vivaldi point to Liszt, and Liszt point to Shumann. The list would appear as:

Add the 5th Name Figure 11.12.


Address | ssn | name | next
520028164 | 551928772 | Grieg, Eduard | 521076740
present
520290308
Address | ssn | name | next
520290308 | 732010233 | Shumann, Robert | NULL
first
520552452
Address | ssn | name | next
520552452 | 321100678 | Beethoven, Ludwig | 520028164
present
520814596
Address | Ssn | name | next
520814596 | 697467721 | Vivaldi, Antonio | 520290308
customer
521076740
Address | Ssn | name | next
521076740 | 678946790 | Liszt, Franz | 520814596

And we are done.

What about the C Code necessary for the above procedure?

Establishing Dynamically Ordered Linked Lists in C

The code corresponds to that which we saw when we ordered our array of records in C Code 10.4. Based on the code we saw in C Code 11.8., The relevant section of code is:

C/C++ Code 11.9.

```
int main()
{ struct arraytemp studentarray[5] = {{. . .}};
  struct arraytemp * first, * customer, * present, * previous;
  int i;
  for (i = 0; i < 5; i++)
  if ((customer = (struct custrec *) malloc (sizeof(struct custrec))) == NULL)
      { puts("Memory Allocation Failed – Bye!\n");
        return 0;  }
  strcpy(customer -> ssn, custs[i].ssn);             // move over customer ssn
  strcpy(customer -> name, custs[i].name);           // move over customer name
  customer -> next = NULL;                           // This is slightly different
  if (i == 0)                                        // store the first dynamically
      first = customer;                              //     allocated address
  else
 {    present = first;                               // Start with the first record
      while ((present != NULL) && (strcmp(customer->ssn, present->ssn) > 0))
      {   previous = present;                        // Store the last address
          present = present -> next;   }             // Get the next record's address
/* Why are we out of the loop?? Should the new record go before the old one??         */
      if (strcmp(customer -> ssn, present -> ssn) < 0)
      {   if (present == first)                      // Was the old record the first one?
              first = customer;                      // If so, the new record is now the first
          else                                       // Otherwise, insert the new record
              previous -> next = customer;           //       between the two records
          customer -> next = present;     }          // Have the new record point to the old
/*  No, the new record should go after the old record                                  */
       else
       {   if (present -> next != NULL)              // If the old record wasn't the last one
               customer -> next = present -> next;   // Point to the one after the old record
           present -> next = customer;   }           // The old points to the new one
    }   }                                            // The list is ordered
  customer = first;                                  // Start at the the top of the list
  while (customer != NULL)                           // Print out the whole list
  {   printf ("SSN: %12s Name: %15s is stored at at %lu and the next rec is at %lu\n",
              customer->ssn, customer->name, customer, customer->next);
       customer = customer -> next;     }            // Get the next record in the list
  return 0;                                          // We're done
}
```

How do I add, or delete, records from the list???

Maintaining Dynamically Linked Lists

To add a record, we follow the exact same procedure are we used when we created the list: We simply get the needed space (using **malloc**), determine where the new record should go, and adjust the pointers accordingly. Deleting records is equally as simple, although there is one additional step: We find the record to delete, and adjust the pointers. There are only three possibilities:

Record Deletion Possibilities

1. If the record we are deleting is the first record on the list, have the *first* pointer point to the second record on the (ordered) list.
2. If the record we are deleting is the last record, have the record which points to that record point to NULL.
3. If the record we are deleting is between two records, have the record which points to it point to the record which the record we are deleting is pointing to.

There is one additional advantage that we have with dynamically ordered linked lists: **We can free up the memory we were using when we delete a record.**

Remember the command we illustrated in C Code 11.7. Once we have the address of the record we wish to delete (for example, in pointer variable customer), we can release it by issuing the command **free(*customer*);.** Unlike our array of linked records, this command frees up the memory used (in our case, 45 contiguous bytes of RAM) which we can later use, if we wish.

What if I want to set up a doubly linked list? Or order the list on multiple fields?

Dynamically Linked Lists With Multiple Linkages

Anything that we can do with the static linked lists, we can do with dynamically linked lists (and more, as we saw above). For example, if we wished to doubly link our list of customers on their *ssn* field AND link them alphabetically by *name*, our list might appear as (Figure 11.13):

Figure 11.13.

first
520552452
last
520290308
fname
520552452

Address	*ssn*	*name*	*nextid*	*previd*	*nextname*
520028164	551928772	Grieg, Eduard	521076740	52055245	521076740
520290308	732010233	Shumann, Robert	NULL	52081459	520814596
520552452	321100678	Beethoven, Ludwig	520028164	NULL	520028164
520814596	697467721	Vivaldi, Antonio	520290308	52107674	NULL
521076740	678946790	Liszt, Franz	520814596	52002816	520290308

Notice that each record now requires 8 additional bytes of storage (4 each per additional pointer) for a total of 53 bytes per record. Also, if we wish to link the records in this fashion, we need two additional pointer variables, one to point to the last *ssn* (*last*), and one to point to the first *name* (*fname*).

Do Dynamically linked lists offer searching advantages over other linked lists???

Dynamically Linked Lists With Multiple Linkages

No, there are no searching advantages for dynamically linked lists over the way we searched for records in our array of linked records, at least given the way in which we set up our lists (wait until the next chapter). In fact, since we can not physically sort the lists (as we did with our array), we can not perform a binary search. Generally speaking, we locate records in the linked list using a sequential search. Unless, of course, we wish to apply one the same options as before:

Improving Searches on Dynamically Linked Lists

1. We can set up doubly linked lists to search from either direction
2. We can set up a sorted array of pointers, and perform a binary search
3. We can set up leveled lists

As with other linked lists, the trade-off is reduced searching time vs. increased storage, complexity of programming, and increased maintenance. It all depends on how important it is to be able to find a record quickly.

Summary

In this chapter, we removed our final constraint, namely that we need to reserve a ***fixed*** number of ***contiguous*** blocks of RAM prior to running our program. In fact, we have eliminated all of our previously adhered to constraints since we introduced arrays. Along the way, we developed procedures and techniques for dealing with these new data types, such as linking them, refining search methods, and conserving the RAM that we require.

Our basic task for the remainder of this text is to develop approaches for dealing with our other problem: locating records as quickly as possible without the effort required to sort and maintain our data for physically ordered lists.

In the next chapter, we introduce a new data type which helps resolve this dilemma: **binary trees**. Binary trees rely on the use of structured data objects, pointers, dynamic memory allocation, and linkages, so they are not necessarily new. However, the manner in which we conceive of, and manipulate, them is new. Further, as we already know, while they offer advantages, nothing is without its trade-offs.

Chapter Terminology: Be able to fully describe these terms

Dynamic Memory Allocation	NULL pointer
Function free	**sizeof** operator
Function malloc	Static Declarations

Review Questions

1. What are the Advantages of Dynamic Allocation? What are the Disadvantages?
2. How do we know if we don't have the contiguous memory we need for our data type?
3. Assume that we could see the following section of RAM:

65984 - 66112	**66113 - 66368**	**66368 - 66656**
Available	Unavailable	Available

66657 - 66769	**66770 - 66912**	**66913 - 67177**	**67178 - 67296**
Unavailable	Available	Unavailable	Available

67297 - 67600	**67601 - 68286**
Unavailable	Available

68287 - 68455	**68456 - 68629**	**68630 - 68749**	**68750 - 68774**
Unavailable	Available	Unavailable	Available

Now assume the following has been declared:

struct mytemplate
{ **unsigned int** *id*;
char *name*[36], *address*[46], *city*[25], *state*[3], *zip*[6];
char *creditrating*;
float *creditlimit*, *balance*;
struct mytemplate * *nextid*, * *nextname*; };

Assume further that RAM is allocated from lowest address to highest (i.e., the lowest address available will be allocated first). I dynamically allocated 4 records of data **type struct mytemplate** using the command:

customer = (**struct mytemplate** *) malloc (**sizeof**(**struct custrec**));

where customer was declared as: **struct mytemplate * *customer*;**

a. What will the base addresses of each of the records be?
b. If *customer* presently holds the address of the 3rd record allocated, and when linked by ssn, the 3rd record points to the 2nd record which points to the 4th, what is outcome of the commands:

printf("%lu\n", *customer* -> *nextid*);
printf("%lu\n", &(*customer* -> *balance*));
printf("%lu\n", &(*customer* -> *next* -> *next* -> *city)*);

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

1. **What are the Advantages of Dynamic Allocation? What are the Disadvantages**?

Advantages	**Disdvantages**
No Need to reserve memory in advance	**None**, other than those associated
Freeing of memory not being used	with linked lists in general
All of the advantages associated with	(see Question 1, Chapter
10).	
linked lists (see Question 1, Chapter 10).	

2. **How do we know if we don't have the contiguous memory we need for our data type?**

When function malloc returns a NULL address

3. The structured data object **struct mytemplate** requires:

2 + 36 + 46 + 25 + 3 + 6 + 1 + 4 + 4 + 4 + 4 = 135 contiguous bytes of RAM

Looking at the available space in RAM, we find:

Addresses	Contiguous Bytes	Record Allocation(?)
65984 - 66112	129	NO
66368 - 66656	289	1st record at **66368** (Through 66502)
		2nd record at **66503** (Through 66637)
66770 - 66912	143	3rd record at **66770** (Through 66904)
67178 - 67296	119	NO
67601 - 68286	686	4th record at **67601** (Through 67735)

If *customer* presently holds the address of the 3rd record allocated, and when linked by ssn, the 3rd record points to the 2nd record which points to the 4th, then:

printf("%lu\n", *customer* -> *nextid*); will produce the output: **66503**

printf("%lu\n", &(*customer* -> *balance*)); will produce the output:

66770 (the base address of the 3rd record) + 2 + 36 + 46 + 25 + 3 + 6 + 1 + 4 = **66893**

printf("%lu\n", &(*customer* -> *next* -> *next* -> *city)*);

Since *customer* points to the 3rd record, which points to the 2nd, which points to the 4th :

customer -> *next* -> *next* implies the 4th record

Given that the base address of the 4th record is 67601, then field *city* will be found at:

67601 + 2 + 36 + 46 = **67685**