

Unit 12 Dynamic Memory Allocation and Linked List

Structure:

- 12.1 Introduction
 - Objectives
- 12.2 Dynamic Memory Allocation
 - Allocating Memory with **malloc**
 - Allocating Memory with **calloc**
 - Freeing Memory
 - Reallocating Memory Blocks
- 12.3 Pointer Safety
- 12.4 The Concept of linked list
 - Inserting a node by using Recursive Programs
 - Sorting and Reversing a Linked List
 - Deleting the Specified Node in a Singly Linked List
- 12.5 Summary
- 12.6 Terminal Questions
- 12.7 Answers to Self Assessment Questions
- 12.8 Answers for Terminal Questions
- 12.9 Exercises

12.1 Introduction

In the previous unit, you studied about the preprocessor in C. You studied that the preprocessor is a program that processes the source code before it passes through the compiler and hence it can be used to make C an easy and efficient language. In this unit, you will study about the dynamic memory allocation techniques. You will also study about the linked list that is another useful application using C.

You can use an array when you want to process data of the same data type and you know the size of the data. Sometimes you may want to process the data but you don't know what the size of the data is. An example of this is when you are reading from a file or keyboard and you want to process the values. In such a case, an array is not useful because you don't know what the dimension of the array should be. C has the facility of dynamic memory allocation. Using this, you can allocate the memory for your storage. The

allocation is done at runtime. When your work is over, you can deallocate the memory. The allocation of memory is done using three functions: **malloc**, **calloc**, and **realloc**. These functions return the pointers to void, so it can be typecast to any data type, thus making the functions generic. These functions take the input as the size of memory requirement.

Objectives:

After studying this unit, you should be able to:

- explain about the dynamic memory allocation
- implement dynamic memory allocation functions like -malloc, calloc and realloc
- explain the concept of linked lists
- discuss the different operations on linked lists

12.2 Dynamic Memory Allocation

The process of allocating memory at run time is known as Dynamic Memory Allocation. Although C does not inherently have this facility, there are four library routines known as “Memory Management Functions” that can be used for allocating and freeing memory during program execution. They are listed in Table 12.1. These functions help us build complex application programs that use the available memory intelligently.

Table 12.1: Functions used for Dynamic Memory allocations and its tasks

Function	Tasks
malloc	Allocates requested size of bytes and returns a pointer to the first byte of the allocated space
calloc	Allocates space for an array of elements, initializes them to zero and then returns a pointer to the memory
free	Frees previously allocated space
realloc	Modifies the size of previously allocated space

Figure 12.1 shows the conceptual view of storage of a C program in memory.

Local variables
Free memory
Global variables
C program instructions

Figure 12.1: storage of a C program- a conceptual view

The program instructions and global and static variables are stored in a region known as *permanent storage area* and the local variables are stored in another area called *stack*. The memory space that is located between these two regions is available for dynamic allocation during execution of the program. This free memory region is called *heap*. The size of the heap keeps changing when program is executed due to creation and death of variables that are local to functions and blocks. Therefore, it is possible to encounter memory “overflow” during dynamic allocation process.

12.2.1 Allocating Memory with malloc

A problem with many simple programs, including in particular little teaching programs such as we've been writing so far, is that they tend to use fixed-size arrays which may or may not be big enough. We have an array of 100 ints for the numbers which the user enters and wishes to find the average of them, what if the user enters 101 numbers? We have an array of 100 chars which we pass to `getline` to receive the user's input, what if the user types a line of 200 characters? If we're lucky, the relevant parts of the program check how much of an array they've used, and print an error message or otherwise gracefully abort before overflowing the array. If we're not so lucky, a program may sail off the end of an array, overwriting other data and behaving quite badly. In either case, the user doesn't get his job done. How can we avoid the restrictions of fixed-size arrays?

The answers all involve the standard library function `malloc`. Very simply, `malloc` returns a pointer to *n* bytes of memory which we can do anything we want to with. If we didn't want to read a line of input into a fixed-size array, we could use `malloc`, instead. It takes the following form:

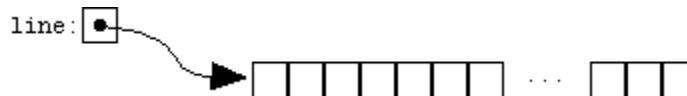
```
ptr=(cast-type *)malloc(byte-size)
```

where `ptr` is a pointer of type *cast-type*. The `malloc` returns a pointer(of *cast-type*) to an area of memory with size *byte-size*.

Here is an example

```
#include <stdlib.h>
char *line;
int linelen = 100;
line = (char *)malloc(linelen);
/* incomplete -- malloc's return value not checked */
getline(line, linelen);
```

`malloc` is declared in `<stdlib.h>`, so we `#include` that header in any program that calls `malloc`. A "byte" in C is, by definition, an amount of storage suitable for storing one character, so the above invocation of `malloc` gives us exactly as many chars as we ask for. We could illustrate the resulting pointer like this:



The 100 bytes of memory (not all of which are shown) pointed to by `line` are those allocated by `malloc`. (They are brand-new memory, conceptually a bit different from the memory which the compiler arranges to have allocated automatically for our conventional variables. The 100 boxes in the figure don't have a name next to them, because they're not storage for a variable we've declared.)

As a second example, we might have occasion to allocate a piece of memory, and to copy a string into it with `strcpy`:

```
char *p = (char *)malloc(15);  
  
/* incomplete -- malloc's return value not checked */  
  
strcpy(p, "Hello, world!");
```

When copying strings, remember that all strings have a terminating `\0` character. If you use `strlen` to count the characters in a string for you, that count will *not* include the trailing `\0`, so you must add one before calling `malloc`:

```
char *somestring, *copy;  
...  
copy = (char *)malloc(strlen(somestring) + 1); /* +1 for \0 */  
/* incomplete -- malloc's return value not checked */  
strcpy(copy, somestring);
```

What if we're not allocating characters, but integers? If we want to allocate 100 ints, how many bytes is that? If we know how big ints are on our machine (i.e. depending on whether we're using a 16- or 32-bit machine) we could try to compute it ourselves, but it's much safer and more portable to let C compute it for us. C has a `sizeof` operator, which computes the size, in

bytes, of a variable or type. It's just what we need when calling malloc. To allocate space for 100 ints, we could call

```
int *ip =(int *)malloc(100 * sizeof(int));
```

The use of the sizeof operator tends to look like a function call, but it's really an operator, and it does its work at compile time.

Since we can use array indexing syntax on pointers, we can treat a pointer variable after a call to malloc almost exactly as if it were an array. In particular, after the above call to malloc initializes ip to point at storage for 100 ints, we can access ip[0], ip[1], ... up to ip[99]. This way, we can get the effect of an array even if we don't know until run time how big the "array" should be. (In a later section we'll see how we might deal with the case where we're not even sure at the point we begin using it how big an "array" will eventually have to be.)

Our examples so far have all had a significant omission: they have not checked malloc's return value. Obviously, no real computer has an infinite amount of memory available, so there is no guarantee that malloc will be able to give us as much memory as we ask for. If we call malloc(1000000000), or if we call malloc(10) 10,000,000 times, we're probably going to run out of memory.

When malloc is unable to allocate the requested memory, it returns a *null pointer*. A null pointer, remember, points definitively nowhere. It's a "not a pointer" marker; it's not a pointer you can use. Therefore, whenever you call malloc, it's vital to check the returned pointer before using it! If you call malloc, and it returns a null pointer, and you go off and use that null pointer as if it pointed somewhere, your program probably won't last long. Instead, a program should immediately check for a null pointer, and if it receives one, it should at the very least print an error message and exit, or perhaps figure out some way of proceeding without the memory it asked for. But it cannot go on to use the null pointer it got back from malloc in any way, because that null pointer by definition points nowhere. ("It cannot use a null pointer in any way" means that the program cannot use the * or [] operators on such a pointer value, or pass it to any function that expects a valid pointer.)

A call to malloc, with an error check, typically looks something like this:

```
int *ip = (int *)malloc(100 * sizeof(int));
if(ip == NULL)
{
    printf("out of memory\n");
    exit or return
}
```

After printing the error message, this code should return to its caller, or exit from the program entirely; it cannot proceed with the code that would have used ip.

Of course, in our examples so far, we've still limited ourselves to "fixed size" regions of memory, because we've been calling malloc with fixed arguments like 10 or 100. (Our call to getline is still limited to 100-character lines, or whatever number we set the linelen variable to; our ip variable still points at only 100 ints.) However, since the sizes are now values which can in principle be determined at run-time, we've at least moved beyond having to recompile the program (with a bigger array) to accommodate longer lines, and with a little more work, we could arrange that the "arrays" automatically grew to be as large as required. (For example, we could write something like getline which could read the longest input line actually seen).

12.2.2 Allocating memory with calloc

calloc is another memory allocation function that is normally used for requesting memory space at run time for storing derived data types such as arrays and structures. While malloc allocates a single block of storage space, calloc allocates multiple blocks of storage, each of the same size, and then sets all bytes to zero. The general form of calloc is:

```
ptr=(cast-type *)calloc(n, elem-size);
```

The above statement allocates contiguous space for n blocks, each of size *elem-size* bytes. All bytes are initialized to zero and a pointer to the first byte of the allocated region is returned. If there is not enough space, a NULL pointer is returned.

Program 12.1 Program to illustrate the use of malloc and calloc functions

```
#include <stdio.h>
#include <malloc.h>
main()
{
    int *base;
    int i;
    int cnt=0;
    int sum=0;
    printf("how many integers you have to store \n");
    scanf("%d",&cnt);
    base = (int *)malloc(cnt * sizeof(int));
    printf("the base of allocation is %16lu \n",base);
    if(!base)
        printf("unable to allocate size \n");
    else
    {
        for(int j=0;j<cnt;j++)
            *(base+j)=5;
    }
    sum = 0;
    for(int j=0;j<cnt;j++)
        sum = sum + *(base+j);
    printf("total sum is %d\n",sum);
    free(base);
    printf("the base of allocation is %16lu \n",base);
    base = (int *)malloc(cnt * sizeof(int));
    printf("the base of allocation is %16lu \n",base);
    base = (int *)malloc(cnt * sizeof(int));
    printf("the base of allocation is %16lu \n",base);
    base = (int *)calloc(10,2);
    printf("the base of allocation is %16lu \n",base);
}
```

12.2.3 Freeing Memory

Memory allocated with malloc lasts as long as you want it to. It does not

automatically disappear when a function returns, as automatic variables do, but it does not have to remain for the entire duration of your program, either. Just as you can use `malloc` to control exactly when and how much memory you allocate, you can also control exactly when you deallocate it.

In fact, many programs use memory on a transient basis. They allocate some memory, use it for a while, but then reach a point where they don't need that particular piece any more. Because memory is not inexhaustible, it's a good idea to deallocate (that is, release or *free*) memory you're no longer using.

Dynamically allocated memory is deallocated with the `free` function. If `p` contains a pointer previously returned by `malloc`, you can call

```
free(p);
```

which will "give the memory back" to the stock of memory (sometimes called the "arena" or "pool") from which `malloc` requests are satisfied. Calling `free` is sort of the ultimate in recycling: it costs you almost nothing, and the memory you give back is immediately usable by other parts of your program. (Theoretically, it may even be usable by other programs.)

(Freeing unused memory is a good idea, but it's not mandatory. When your program exits, any memory which it has allocated but not freed should be automatically released. If your computer were to somehow "lose" memory just because your program forgot to free it, that would indicate a problem or deficiency in your operating system.)

Naturally, once you've freed some memory you must remember not to use it any more. After calling

```
free(p);
```

it is probably the case that `p` still points at the same memory. However, since we've given it back, it's now "available," and a later call to `malloc` might give that memory to some other part of your program. If the variable `p` is a global variable or will otherwise stick around for a while, one good way to record the fact that it's not to be used any more would be to set it to a null pointer:

```
free(p);  
p = NULL;
```


Now we don't even have the pointer to the freed memory any more, and (as long as we check to see that `p` is non-NULL before using it), we won't misuse any memory via the pointer `p`.

When thinking about `malloc`, `free`, and dynamically-allocated memory in general, remember again the distinction between a pointer and what it points to. If you call `malloc` to allocate some memory, and store the pointer which `malloc` gives you in a local pointer variable, what happens when the function containing the local pointer variable returns? If the local pointer variable has *automatic duration* (which is the default, unless the variable is declared static), it will disappear when the function returns. But for the pointer variable to disappear says nothing about the memory pointed to! That memory still exists and, as far as `malloc` and `free` are concerned, is still allocated. The only thing that has disappeared is the pointer variable you had which pointed at the allocated memory. (Furthermore, if it contained the only copy of the pointer you had, once it disappears, you'll have no way of freeing the memory, and no way of using it, either. Using memory and freeing memory both require that you have at least one pointer to the memory!)

12.2.4 Reallocating Memory Blocks

Sometimes you're not sure at first how much memory you'll need. For example, if you need to store a series of items you read from the user, and if the only way to know how many there are is to read them until the user types some "end" signal, you'll have no way of knowing, as you begin reading and storing the first few, how many you'll have seen by the time you do see that "end" marker. You might want to allocate room for, say, 100 items, and if the user enters a 101st item before entering the "end" marker, you might wish for a way to say "uh, `malloc`, remember those 100 items I asked for? Could I change my mind and have 200 instead?"

In fact, you can do exactly this, with the `realloc` function. You hand `realloc` an old pointer (such as you received from an initial call to `malloc`) and a new size, and `realloc` does what it can to give you a chunk of memory big enough to hold the new size. For example, if we wanted the `ip` variable from an earlier example to point at 200 ints instead of 100, we could try calling

```
ip = realloc(ip, 200 * sizeof(int));
```

Since you always want each block of dynamically-allocated memory to be contiguous (so that you can treat it as if it were an array), you and `realloc` have to worry about the case where `realloc` can't make the old block of memory bigger "in place," but rather has to relocate it elsewhere in order to find enough contiguous space for the new requested size. `realloc` does this by returning a new pointer. If `realloc` was able to make the old block of memory bigger, it returns the same pointer. If `realloc` has to go elsewhere to get enough contiguous memory, it returns a pointer to the new memory, after copying your old data there. (In this case, after it makes the copy, it frees the old block.) Finally, if `realloc` can't find enough memory to satisfy the new request at all, it returns a null pointer. Therefore, you usually don't want to overwrite your old pointer with `realloc`'s return value until you've tested it to make sure it's not a null pointer. You might use code like this:

```
int *newp;
newp = realloc(ip, 200 * sizeof(int));
if(newp != NULL)
    ip = newp;
else
{
    printf("out of memory\n");
    /* exit or return */
    /* but ip still points at 100 ints */
}
```

If `realloc` returns something other than a null pointer, it succeeded, and we set `ip` to what it returned. (We've either set `ip` to what it used to be or to a new pointer, but in either case, it points to where our data is now.) If `realloc` returns a null pointer, however, we hang on to our old pointer in `ip` which still points at our original 100 values.

Putting this all together, here is a piece of code which reads lines of text from the user, treats each line as an integer by calling `atoi`, and stores each integer in a dynamically-allocated "array":

```
#define MAXLINE 100
char line[MAXLINE];
int *ip;
int nalloc, nitems;
```

```
nalloc = 100;
ip = (int *)malloc(nalloc * sizeof(int));          /* initial allocation */
if(ip == NULL)
{
    printf("out of memory\n");
    exit(1);
}

nitems = 0;

while(getline(line, MAXLINE) != EOF)
{
    if(nitems >= nalloc)
    {
        /* increase allocation */
        int *newp;
        nalloc += 100;
        newp = realloc(ip, nalloc * sizeof(int));
        if(newp == NULL)
        {
            printf("out of memory\n");
            exit(1);
        }
        ip = newp;
    }

    ip[nitems++] = atoi(line);
}
```

We use two different variables to keep track of the “array” pointed to by `ip`. `nalloc` is how many elements we’ve allocated, and `nitems` is how many of them are in use. Whenever we’re about to store another item in the “array,” if `nitems >= nalloc`, the old “array” is full, and it’s time to call `realloc` to make it bigger.

Finally, we might ask what the return type of `malloc` and `realloc` is, if they are able to return pointers to `char` or pointers to `int` or (though we haven’t seen it yet) pointers to any other type. The answer is that both of these functions are declared (in `<stdlib.h>`) as returning a type we haven’t seen, `void *` (that

is, pointer to void). We haven't really seen type void, either, but what's going on here is that void * is specially defined as a "generic" pointer type, which may be used (strictly speaking, assigned to or from) any pointer type.

Self Assessment Questions

1. The process of allocating memory at run time is known as _____.
2. malloc() function returns a pointer to integer. (True/False)
3. For deallocating memory, you can use _____ function.
4. The function that is used to alter the size of a block previously allocated is _____.

12.3 Pointer Safety

The hard thing about pointers is not so much manipulating them as ensuring that the memory they point to is valid. When a pointer doesn't point where you think it does, if you inadvertently access or modify the memory it points to, you can damage other parts of your program, or (in some cases) other programs or the operating system itself!

When we use pointers to simple variables, there's not much that can go wrong. When we use pointers into arrays, and begin moving the pointers around, we have to be more careful, to ensure that the moving pointers always stay within the bounds of the array(s). When we begin passing pointers to functions, and especially when we begin returning them from functions, we have to be more careful still, because the code using the pointer may be far removed from the code which owns or allocated the memory.

One particular problem concerns functions that return pointers. Where is the memory to which the returned pointer points? Is it still around by the time the function returns? The strstr function (as described in the earlier unit), returns either a null pointer (which points definitively nowhere, and which the caller presumably checks for) or it returns a pointer which points into the input string, which the caller supplied, which is pretty safe. One thing a function must *not* do, however, is return a pointer to one of its own, local, automatic arrays. Remember that automatic variables (which includes all non-static local variables), including automatic arrays, are deallocated and

disappear when the function returns. If a function returns a pointer to a local array, that pointer will be invalid by the time the caller tries to use it.

Finally, when we're doing dynamic memory allocation with malloc, calloc, realloc, and free, we have to be most careful of all. Dynamic allocation gives us a lot more flexibility in how our programs use memory, although with that flexibility comes the responsibility that we manage dynamically allocated memory carefully. The possibilities for misdirected pointers and associated havoc are greatest in programs that make heavy use of dynamic memory allocation. You can reduce these possibilities by designing your program in such a way that it's easy to ensure that pointers are used correctly and that memory is always allocated and deallocated correctly. (If, on the other hand, your program is designed in such a way that meeting these guarantees is a tedious nuisance, sooner or later you'll forget or neglect to, and maintenance will be a nightmare.)

12.4 The Concept of Linked List

When dealing with many problems we need a dynamic list, dynamic in the sense that the size requirement need not be known at compile time. Thus, the list may grow or shrink during runtime. A *linked list* is a data structure that is used to model such a dynamic list of data items, so the study of the linked lists as one of the data structures is important.

An array is represented in memory using sequential mapping, which has the property that elements are fixed distance apart. But this has the following disadvantage: It makes insertion or deletion at any arbitrary position in an array a costly operation, because this involves the movement of some of the existing elements.

When we want to represent several lists by using arrays of varying size, either we have to represent each list using a separate array of maximum size or we have to represent each of the lists using one single array. The first one will lead to wastage of storage, and the second will involve a lot of data movement.

So we have to use an alternative representation to overcome these disadvantages. One alternative is a linked representation. In a linked representation, it is not necessary that the elements be at a fixed distance apart. Instead, we can place elements anywhere in memory, but to make it a

part of the same list, an element is required to be linked with a previous element of the list. This can be done by storing the address of the next element in the previous element itself. This requires that every element be capable of holding the data as well as the address of the next element. Thus every element must be a structure with a minimum of two fields, one for holding the data value, which we call a data field, and the other for holding the address of the next element, which we call link field.

Therefore, a linked list is a list of elements in which the elements of the list can be placed anywhere in memory, and these elements are linked with each other using an explicit link field, that is, by storing the address of the next element in the link field of the previous element.

Program 12.2: Here is a program for building and printing the elements of the linked list

```
# include <stdio.h>
# include <stdlib.h>
struct node
{
    int data;
    struct node *link;
};
struct node *insert(struct node *p, int n)
{
    struct node *temp;
    /* if the existing list is empty then insert a new node as the
starting node */
    if(p==NULL)
    {
        p=(struct node *)malloc(sizeof(struct node)); /* creates new node
data value passes as parameter */
        if(p==NULL)
        {
            printf("Error\n");
            exit(0);
        }
        p-> data = n;
```

```
        p-> link = p; /* makes the pointer pointing to itself because it
is a circular list*/
    }
    else
    {
        temp = p;
        /* traverses the existing list to get the pointer to the last node of
it */
        while (temp-> link != p)
            temp = temp-> link;
        temp-> link = (struct node *)malloc(sizeof(struct node)); /*
creates new node using data value passes as parameter and puts its
address in the link field of last node of the existing list*/
        if(temp -> link == NULL)
        {
            printf("Error\n");
            exit(0);
        }
        temp = temp-> link;
        temp-> data = n;
        temp-> link = p;
    }
    return (p);
}

void printlist ( struct node *p )
{
    struct node *temp;
    temp = p;
    printf("The data values in the list are\n");
    if(p!= NULL)
    {
        do
        {
            printf("%d\t",temp->data);
            temp=temp->link;
        } while (temp!= p);
    }
```

```
    }
    else
        printf("The list is empty\n");
}

void main()
{
    int n;
    int x;
    struct node *start = NULL ;
    printf("Enter the nodes to be created \n");
    scanf("%d",&n);
    while ( n -- > 0 )
    {
        printf( "Enter the data values to be placed in a node\n");
        scanf("%d",&x);
        start = insert ( start, x );
    }
    printf("The created list is\n");
    printlist ( start );
}
```

12.4.1 Inserting a node by using Recursive Programs

A linked list is a recursive data structure. A *recursive data structure* is a data structure that has the same form regardless of the size of the data. You can easily write recursive programs for such data structures.

Program 12.3

```
# include <stdio.h>
# include <stdlib.h>
struct node
{
    int data;
    struct node *link;
};
struct node *insert(struct node *p, int n)
{
```



```
    struct node *temp;
    if(p==NULL)
    {
        p=(struct node *)malloc(sizeof(struct node));
        if(p==NULL)
        {
            printf("Error\n");
            exit(0);
        }
        p-> data = n;
        p-> link = NULL;
    }
    else
        p->link = insert(p->link,n);/* the while loop replaced by
recursive call */
    return (p);
}
void printlist ( struct node *p )
{
    printf("The data values in the list are\n");
    while (p!= NULL)
    {
        printf("%d\t",p-> data);
        p = p-> link;
    }
}
void main()
{
    int n;
    int x;
    struct node *start = NULL ;
    printf("Enter the nodes to be created \n");
    scanf("%d",&n);
    while ( n - 0 )
    {
        printf( "Enter the data values to be placed in a node\n");
```

```
        scanf("%d",&x);
        start = insert ( start, x );
    }
    printf("The created list is\n");
    printlist ( start );
}
```

12.4.2 Sorting and reversing a linked list

To sort a linked list, first we traverse the list searching for the node with a minimum data value. Then we remove that node and append it to another list which is initially empty. We repeat this process with the remaining list until the list becomes empty, and at the end, we return a pointer to the beginning of the list to which all the nodes are moved.

To reverse a list, we maintain a pointer each to the previous and the next node, then we make the link field of the current node point to the previous, make the previous equal to the current, and the current equal to the next.

Therefore, the code needed to reverse the list is

```
Prev = NULL;
While (curr != NULL)
{
    Next = curr->link;
    Curr -> link = prev;
    Prev = curr;
    Curr = next;
}
```

Program 12.4: Example in sorting lists.

```
# include <stdio.h>
# include <stdlib.h>
struct node
{
    int data;
    struct node *link;
};
struct node *insert(struct node *p, int n)
{
```

```
struct node *temp;
if(p==NULL)
{
    p=(struct node *)malloc(sizeof(struct node));
    if(p==NULL)
    {
        printf("Error\n");
        exit(0);
    }
    p-> data = n;
    p-> link = NULL;
}
else
{
    temp = p;
    while (temp-> link!= NULL)
        temp = temp-> link;
    temp-> link = (struct node *)malloc(sizeof(struct node));
    if(temp -> link == NULL)
    {
        printf("Error\n");
        exit(0);
    }
    temp = temp-> link;
    temp-> data = n;
    temp-> link = null;
}
return(p);
}

void printlist ( struct node *p )
{
    printf("The data values in the list are\n");
    while (p!= NULL)
    {
        printf("%d\t",p-> data);
```

```
        p = p-> link;
    }
}

/* a function to sort reverse list */
struct node *reverse(struct node *p)
{
    struct node *prev, *curr;
    prev = NULL;
    curr = p;
    while (curr != NULL)
    {
        p = p-> link;
        curr-> link = prev;
        prev = curr;
        curr = p;
    }
    return(prev);
}

/* a function to sort a list */
struct node *sortlist(struct node *p)
{
    struct node *temp1, *temp2, *min, *prev, *q;
    q = NULL;
    while(p != NULL)
    {
        prev = NULL;
        min = temp1 = p;
        temp2 = p -> link;
        while ( temp2 != NULL )
        {
            if(min -> data > temp2 -> data)
            {
                min = temp2;
                prev = temp1;
            }
        }
    }
}
```

```
        temp1 = temp2;
        temp2 = temp2-> link;
    }
    if(prev == NULL)
        p = min -> link;
    else
        prev -> link = min -> link;
    min -> link = NULL;
    if( q == NULL)
        q = min; /* moves the node with lowest data value in the list
pointed to by p to the list pointed to by q as a first node*/
    else
    {
        temp1 = q;
        /* traverses the list pointed to by q to get pointer to its
last node */
        while( temp1 -> link != NULL)
            temp1 = temp1 -> link;
        temp1 -> link = min; /* moves the node with lowest data value in the
list pointed to by p to the list pointed to by q at the end of list pointed by q*/
    }
}
return (q);
}

void main()
{
    int n;
    int x;
    struct node *start = NULL ;
    printf("Enter the nodes to be created \n");
    scanf("%d",&n);
    while ( n- > 0 )
    {
        printf( "Enter the data values to be placed in a
node\n");
```

```
        scanf("%d",&x);
        start = insert ( start,x);
    }
    printf("The created list is\n");
    printlist ( start );
    start = sortlist(start);
    printf("The sorted list is\n");
    printlist ( start );
    start = reverse(start);
    printf("The reversed list is\n");
    printlist ( start );
}
```

12.4.3 Deleting the specified node in a singly linked list

To delete a node, first we determine the node number to be deleted (this is based on the assumption that the nodes of the list are numbered serially from 1 to n). The list is then traversed to get a pointer to the node whose number is given, as well as a pointer to a node that appears before the node to be deleted. Then the link field of the node that appears before the node to be deleted is made to point to the node that appears after the node to be deleted, and the node to be deleted is freed.

Program 12.5: Example of working with nodes.

```
#include <stdio.h>
#include <stdlib.h>
struct node *delet ( struct node *, int );
int length ( struct node * );
struct node
{
    int data;
    struct node *link;
};
struct node *insert(struct node *p, int n)
{
    struct node *temp;
    if(p==NULL)
    {
```

```
p=(struct node *)malloc(sizeof(struct node));
if(p==NULL)
{
    printf("Error\n");
    exit(0);
}
p-> data = n;
p-> link = NULL;
}
else
{
    temp = p;
    while (temp-> link != NULL)
        temp = temp-> link;
    temp-> link = (struct node *)malloc(sizeof(struct node));
    if(temp -> link == NULL)
    {
        printf("Error\n");
        exit(0);
    }
    temp = temp-> link;
    temp-> data = n;
    temp-> link = NULL;
}
return (p);
}

void printlist ( struct node *p )
{
    printf("The data values in the list are\n");
    while (p!= NULL)
    {
        printf("%d\t",p-> data);
        p = p-> link;
    }
}

void main()
```

```
{
    int n;
    int x;
    struct node *start = NULL;
    printf("Enter the nodes to be created \n");
    scanf("%d",&n);
    while ( n-- > 0 )
    {
        printf( "Enter the data values to be placed in a node\n");
        scanf("%d",&x);
        start = insert ( start, x );
    }
    printf(" The list before deletion id\n");
    printlist ( start );
    printf("%d \n Enter the node no \n");
    scanf ( " %d",&n);
    start = delet (start , n );
    printf(" The list after deletion is\n");
    printlist ( start );
}

/* a function to delete the specified node*/
struct node *delet ( struct node *p, int node_no )
{
    struct node *prev, *curr ;
    int i;

    if (p == NULL )
    {
        printf("There is no node to be deleted \n");
    }
    else
    {
        {
            if ( node_no > length (p))
            {
                printf("Error\n");
            }
        }
    }
}
```



```
else
{
    prev = NULL;
    curr = p;
    i = 1 ;
    while ( i < node_no )
    {
        prev = curr;
        curr = curr-> link;
        i = i+1;
    }
    if ( prev == NULL )
    {
        p = curr -> link;
        free ( curr );
    }
    else
    {
        prev -> link = curr -> link ;
        free ( curr );
    }
}
}
return(p);
}
/* a function to compute the length of a linked list */
int length ( struct node *p )
{
    int count = 0 ;
    while ( p != NULL )
    {
        count++;
        p = p->link;
    }
    return ( count ) ;
}
```

Self Assessment Questions

5. A linked list is a data structure that is used to model a dynamic list of data items and is a collection of _____.
6. Linked list make use of _____ memory allocation technique.

12.5 Summary

The process of allocating memory at run time is known as Dynamic Memory Allocation. The allocation of memory is done using three functions: malloc, calloc, and realloc. These functions return the pointers to void, so it can be typecast to any data type, thus making the functions generic. The memory space that is used for dynamic allocation during execution of the program is called *heap*. When a pointer doesn't point where you think it does, if you inadvertently access or modify the memory it points to, you can damage other parts of your program. So safety of pointers is essential in dynamic memory allocation. Linked list is one of the applications of dynamic memory allocation.

12.6 Terminal Questions

1. Assuming that there is a structure definition with structure tag **student**, write the **malloc** function to allocate space for the structure.
2. Write a structure definition to represent a node in the linked list with two data fields of type integer.
3. If **ptr** points to a node and **link** is the address field of the node, how can you move from the node to the next node?
4. Explain about the linked list.

12.7 Answers to Self Assessment Questions

1. Dynamic Memory Allocation.
2. False
3. free()
4. realloc()
5. nodes.
6. dynamic

12.8 Answers to Terminal Questions

1. `ptr=(struct student *) malloc(sizeof(struct student));`
2. `struct node`
`{`
`int data1;`
`int data2;`
`struct node *link;`
`};`
3. `ptr=ptr->link;`
4. A linked list is a data structure that is used to model a dynamic list of data items and is a collection of nodes. (Refer to section 12.4 for more details)

12.9 Exercises

1. Write a menu driven program to create a linked list of a class of students and perform the following operations.
 - i. Write out the contents of a list
 - ii. Edit the details of a specified student
 - iii. Count the number of students above a specified age and weight
2. Write a program to insert a node after the specified node in a linked list.
3. Write a program to count the number of nodes in linked list.
4. Write a program to merge two sorted lists.
5. Explain briefly how to represent polynomials using linked lists. Write a program to add two polynomials.