

C Programming

Learning Package 11

Computer Science bit

Dr Scott Turner

School of Science and Technology

University of Northampton



**Introduction**

In this Learning Package, we are going to look at two data structures, the linked list and the stack.

**Aims**

During this Learning Package the reader will:

* Meet two widely used data structure, the linked list and the stack.
* Be introduced to further uses of pointers
* Use user defined data types
* Use of recursion in functions.

**Study guide**

Session 1:

Section 11.1

Task 11.1

SAQ 11.1

Task 11.2

SAQ 11.2

Task 11.3

Session 2:

Section 11.2

Task 11.4

Section 11.3

Session 3:

Section 11.4

Task 11.5

SAQ 11.3

* 1. **Linked List 1**

A link list is a data structure, which can be expanding by just adding another block (element) in the appropriate place in the list. The element has two parts:

* + - first part with the data to be stored,
		- second part that stores where in memory the next element to be connected is.

Figure 11.1

struct list{

Contents c; struct list \*pn;

};

typedef struct list L\_member;

typedef L\_member \*Link; /\*Link is a type pointer to L\_member\*/

This structure has a member field (pn) that points to the another data structure of the same type. The pointer variable pn is a link. Each structure is linked to another similar structure by this variable pn. This pointer pn contains either address of the memory location of the next list element, or the value NULL defined as 0 (the end of the list).

Figure 11.2

What does the -> operator do? This is the member access operator (->), and is used to ‘get to’ a member of the structure via a pointer. The pointer is assigned the address of the structure. So for the line p->pn=NULL. The pointer p points to an address of start address of an element in the link list.

Part (a member) of the element is the variable pn (which points to the next element of the link list), this particular instance is assigned to point to NULL the end of the list.

Task 11.1 Type in the program below and put comments into explain what the program does.

#include <stdio.h>

#include <stdlib.h> typedef char Contents;

struct list{

Contents data; struct list \*pn;

};

typedef struct list L\_member;

int main()

{

struct list a, b, c,d; a.data='A';

b.data='B';

c.data='C';

d.data='S'; a.pn=b.pn=c.pn=d.pn=NULL; a.pn=&b;

b.pn=&c; c.pn=&d;

printf("\ndata in a=%c ",a.data); printf("\ndata in b=%c ",a.pn->data); printf("\ndata in c=%c ",a.pn->pn->data); printf("\ndata in d=%c ",a.pn->pn->pn->data); printf("\n\n");

system("PAUSE"); return 0;

}

Figure 11.3 Example of the program in Task 11.1

SAQ 11.1

Filling in the missing letters in the following:

1. The -> operator is the **m r a s** operator and is used to ‘get to’ a

**m r** of the **s e** via a **p r**.

1. A **l d l t** is a data **s e** where can be expanding by adding another **e t**. The **e t** has **t\_o p s** one with the **d a** to be stored, and a second part that holds where the **n t e t** to be connected is.

Lets develop this a bit further, if instead of storing individual items in separate lists as in Task 11.1 lets look at a way of setting up a linked list so we can add items when we need to. We can tackle this by using pointers and assigning blocks of memory to store the new element each memory.

The first step is set up a new data type that is a pointer to elements in the link list.

struct list{

Contents data; struct list \*pn;

};

typedef struct list L\_member; typedef L\_member \*Link;

In the code above, a variable called Link is defined as a pointer to the type list (our elements). Remember each element in our linked list is of the type list.

We next need some way of allocating memory each time we want a new element. This can be done with the function malloc(size) where size (we can use the function sizeof() for this) is the amount of memory needed for the element. Therefore, in this case malloc(sizeof(L\_member)) and it returns a pointer containing the address of the start of the assigned block of memory.

We do not use any variables of type list directly, we use pointers to list variables for example Link head . This piece of code defines head to be a pointer to a list element. Once is this defined we can start using it.

head=malloc(sizeof(L\_member)); head->data='A';

head->pn=NULL;

head->pn=malloc(sizeof(L\_member)); head->pn->data='B';

head->pn->pn=NULL;

Task 11.2

Alter this program where indicated to insert a new element at the end of the linked list.

#include <stdio.h>

#include <stdlib.h>

typedef char Contents; struct list{

Contents data; struct list \*pn;

};

typedef struct list L\_member; typedef L\_member \*Link;

int main()

{

Link head; head=malloc(sizeof(L\_member)); head->data='A';

head->pn=NULL;

head->pn=malloc(sizeof(L\_member)); head->pn->data='B';

head->pn->pn=NULL;

/\*insert a line here to assign memory to a new element\*/

/\*insert C is to the data member of the new element\*/

/\*insert NULL into the next pointer (pn) of the new element\*/ printf("\ndata in element 1=%c ",head->data);

printf("\ndata in element 2=%c ",head->pn->data); printf("\ndata in element 3=%c ",head->pn->pn->data); printf("\n\n");

system("PAUSE"); return 0;

}

Figure 11.4 Example of Task 11.2

Lets add a subroutine to allocate space in memory for each next element of the list

int main()

{

Link head; head=allocate(); head->data='A'; head->pn=NULL;

head->pn=allocate(); head->pn->data='B'; head->pn->pn=NULL;

head->pn->pn=allocate(); head->pn->pn->data='C'; head->pn->pn->pn=NULL;

printf("\ndata in element 1=%c ",head->data); printf("\ndata in element 2=%c ",head->pn->data); printf("\ndata in element 3=%c ",head->pn->pn->data); system("PAUSE");

return 0;

}

Link allocate(void)

{

return((Link) malloc(sizeof(L\_member)));

}

SAQ 11.2

Filling in the missing letters in the following:

We need some way of allocating **m y** each time we want a new **e t**. This can be done with the **f n m c**(size) where **s e** is the amount of **m y** needed for the **e t**.

Task 11.3

Type in the program below and describe its function with the aid of comments.

#include <stdio.h>

#include <stdlib.h>

typedef char Contents; struct list{

Contents data; /\*data is of type contents defined above\*/ struct list \*pn;/\*pn is a pointer to a struct of type list\*/

};

typedef struct list L\_member; /\*L\_member is of type list\*/ typedef L\_member \*Link; /\*Link is a type pointer to L\_member\*/ Link allocate(void);

Link assign(Contents \*q); Link generate(char s); int main()

{

Link head,p1;

head= generate('A'); head->pn=generate('B');

head->pn->pn=generate('C'); head->pn->pn->pn=generate('D');

printf("\ndata in element 1=%c ",head->data); printf("\ndata in element 2=%c ",head->pn->data); printf("\ndata in element 4=%c ",head->pn->pn->pn->data); system("PAUSE");

return 0;

}

Link allocate(void)

{

return((Link) malloc(sizeof(L\_member)));

}

Link assign(Contents \*q)

{

Link p; p=allocate(); if (p==NULL)

{

puts ("\n\*\*\*\*\*\*\*\*\*\*\*Unable to allocate storage\*\*\*\*"); exit(0);

}

else

{

p->data=\*q; p->pn=NULL;

return(p);

}

}

Link generate(char s)

{

Link p; p=assign(&s); return(p);

}

* 1. **Link List 2**

Looking at the lines in the main function in Task 11.3 that assigned the letters to the linked list, it looks like if we could get the program to repeat the same action repeatedly. By moving a pointer each time to the next element we could reduce the code. We can alter the function generate so it takes in a pointer to the beginning of a string we want to store, and get the new function repeatly call itself. This is a recursive function.

Link generate1(char \*s)

{

Link p;

if (\*s=='\0')

{

return (NULL);

}

else

{

p=assign(s);

p->pn=generate1(s+1); return(p);

}

}

This function's input is a pointer to a string and will return a pointer of the type Link. This is a pointer to an element of the link list.

Link generate1(char \*s)

In the code below a pointer variable of type Link is assigned. The second part of the block of code is a routine for dealing with what happens when the end of the string is reached. When this happens, a NULL is returned from the function. In other words, the end of the list and the pointer to the next element points to the NULL, saying that this is the end of the list.

Link p;

if (\*s=='\0')

{

return (NULL);

}

When we are not dealing with the end of the string, we need to allocate space for each member of the structure. Each member of the structure consists of a character and a pointer to the next element of the list. The sneaky thing we can do here is make the function repeating itself as many times as necessary by calling itself as needed (see figure 11.5)

p=assign(s);

p->pn=generate1(s+1); return(p);

Figure 11.5

Task 11.4

Copy the program you produce for Task 11.3 into a new project and then alter the programs to accommodate then new code shown below.

Link generate1(char \*s); int main()

{

Link head,p1;

head= generate1("ABCD");

printf("\ndata in element 1=%c ",head->data); printf("\ndata in element 2=%c ",head->pn->data); printf("\ndata in element 4=%c ",head->pn->pn->pn->data); system("PAUSE");

return 0;

}

Link generate1(char \*s)

{

Link p;

if (\*s=='\0')

{

return (NULL);

}

else

{

p=assign(s);

p->pn=generate1(s+1); return(p);

}

}

Comment the new code and write an explanation of what the program does and how it does? What does the function generate1() return to the main function()?

* 1. **Link list 3**

Read from 11.1 *Introduction* on page 251 to end of section labelled *Simple Lists* on page 260. Follow it through to build the link list.

I had problems with Program 11.3 in the book in particular with the function list\_print(Link p), it would not allow the NULL character to be printed. I found the following function to work though so I suggest you use this one. (A complete program for Program 11.3 is included at the end of this learning package):

void list\_print(Link p)

{

if (p==NULL)

{

printf("Null: (%6u)\n",p);

}

else

{

printf("%c (%6u)--> ",p->c,p); list\_print(next(p));

}

}

Figure 11.6 Example of Program 11.3.

Have a go at the Exercise on page 260 of the module textbook (if you are stuck a partial solution can be found at the end of the Learning Package). Example of one test can be seen in Figure 11.7.

Figure 11.7 Example

This program is very similar to the one you built previous.

* 1. **Stacks**

We are now going to extend some of the ideas we met in the linked list programs, to using a linked list to produce a data structure called the stack.

* 1. 1 Setting up the stack

You may have met stacks previously in other modules in the course. A stack can be thought of as a pile of items, one stacked on top of the other. If you want to put an item on top the stack, you push it on at the top of the stack. If you want to take something off the stack, it has to be taken off the top of the stack. Access to the stack is limited to the item that is current at the top of this pile of items.

Figure 11.8 Stack

In Figure 11.8, an example of a linked list being used to form a stack.

Let’s start building by defining an element of the stack

typedef char data; struct element{

data c;

struct element \*pn;

};

typedef struct element elem;

We need to store certain piece of data store about the stack:

* + - How many elements are there in the stack
		- pointer to the element at the top of the stack.

We can does this be define a new data structures as below

struct stack1{ int count; elem \*top;

};

typedef struct stack1 stack;

We know need to initialise the stack so that it the stack is empty and not pointing to any elements.

void initial1(stack1 \*stck)

{

stck->cnt=0; stck->top=NULL;

}

* + 1. Putting items on to the stack

Up to this point, we have defined the elements of the stack, created a separate data structure to contain information about the stack, and produced a function that initialises this data structure.

We now need a way of putting new elements on to the stack. This done by creating a new element, setting the next pointer to in the element that is currently at the top of the stack and change the top pointer in the stack structure to point to the new element. The new element is now the top of the stack (see Figure 11.9).

Figure 11.9 New element added to the stack The code for doing this is

void push(data d, stack1 \*stck)

{

elem \*p; p=malloc(sizeof(elem)); p->c=d;

p->pn=stck->top; stck->top=p;

stck->count=stck->count+1;

}

This routine allocates memory to this element, then fills it with the necessary data and pointer values. The routine also alter the element of the stack

structure increasing the count by one and making the top of the stack pointer point to the new element.

* + 1. Taking elements off the stack

Taking an element of a stack involves going to the top of the stack and altering the top of stack pointer to point to the element that the removed element was pointing to as the next element.

data pop(stack1 \*stck)

{

data c; elem \*p;

c=stck->top->c; p=stck->top;

stck->top=stck->top->pn; stck->count=stck->count-1; free(p);

return c;

}

Task 11.5

Type in the program below and describe the how the result was achieved:

#include <stdio.h>

#include <stdlib.h>

#define EMPTY 0

#define FULL 1000

typedef char data;

typedef enum {false, true} boolean;

struct element{ data c;

struct element \*pn;

};

typedef struct element elem; struct stack1{

int count; elem \*top;

};

typedef struct stack1 stack;

void initial1(stack \*stck)

{

stck->count=0; stck->top=NULL;

}

void push(data d, stack \*stck)

{

elem \*p; p=malloc(sizeof(elem)); p->c=d;

p->pn=stck->top; stck->top=p;

stck->count=stck->count+1;

}

data pop(stack \*stck)

{

data c; elem \*p;

c=stck->top->c; p=stck->top;

stck->top=stck->top->pn; stck->count=stck->count-1; free(p);

return c;

}

data top(stack \*stck)

{

return (stck->top->c);

}

boolean empty(const stack \*stck)

{

return((boolean) (stck->count==EMPTY));

}

boolean full(const stack \*stck)

{

return((boolean) (stck->count==FULL));

}

int main(void)

{

char string1[]="This stack did this to my message"; int i;

stack s; initial1(&s);

printf("String is: %s\n",string1); for (i=0;string1[i] !='\0';++i)

if (!full(&s))

push(string1[i],&s); printf("From the stack: "); while(!empty(&s))

putchar(pop(&s)); putchar('\n');

system("PAUSE"); return 0;

}

SAQ 11.3

Filling in the missing letters in the following:

1. A **f n** that call itself is a **r e f n**.
2. A stack can be thought of as a pile of items, one **st ked** on **t\_p** of the other. If you want to put an item on top the **st k**, you push it on at the **t\_p** of the stack. If you want to take something off the **st k** it has to be taken off the **t\_p** of the stack.
3. We can put new **e s** on to the **st k** by creating a new **e t**, setting the next **p r** to in the **e t** that is currently at the **t\_p** of the **st k** and change the **p r** in the stack **s e** to point to the new **e t**. The new **e t** is now the **t\_p** of the stack.

**SAQ Answers**

SAQ 11.1

1. The -> operator is the **member access** operator and is used to ‘get to’ a

**member** of the **structure** via a **pointer**.

1. A **linked list** is a data **structure** where can be expanding by adding another **element**. The **element** has **two parts** one with the **data** to be stored, and a second part that holds where the **next element** to be connected is.

SAQ 11.2

We need some way of allocating **memory** each time we want a new **element**. This can be done with the **function malloc**(size) where **size** is the amount of **memory** needed for the **element**.

SAQ 11.3

(a)A **function** that calls itself is a **recursive function**.

(b)A stack can be thought of as a pile of items, one **stacked** on **top** of the other. If you want to put an item on top the **stack**, you push it on at the **top** of the stack. If you want to take something off the **stack** it has to be taken off the **top** of the stack.

1. We can put new **elements** on to the **stack** by creating a new **element**, setting the next **pointer** to in the **element** that is currently at the **top** of the **stack** and change the **pointer** in the stack **structure** to point to the new **element**. The new **element** is now the **top** of the stack.

**Selected tasks** Task 11.1

#include <stdio.h>

#include <stdlib.h>

typedef char Contents; struct list{

Contents data; /\*data is of type contents defined above\*/ struct list \*pn;/\*pn is a pointer to a struct of type list\*/

};

typedef struct list L\_member; /\*L\_member is of type list\*/ typedef L\_member \*Link; /\*Link is a type pointer to L\_member\*/

int main()

{

struct list a, b, c,d;

/\*intialise member variables data in a,b,c,d to A,B,C,S respectively\*/

a.data='A';

b.data='B';

c.data='C';

d.data='S';

/\*pn in all four structures is intially NULL\*/

a.pn=b.pn=c.pn=d.pn=NULL;

/\*connecting together\*/ a.pn=&b;/\*pn in a points to b\*/ b.pn=&c;/\*pn in b points to c\*/ c.pn=&d;/\*pn in c points to d\*/

/\*prints out the data in the linked list\*/ printf("\ndata in a=%c ",a.data); printf("\ndata in b=%c ",a.pn->data); printf("\ndata in c=%c ",a.pn->pn->data); printf("\ndata in d=%c ",a.pn->pn->pn->data); printf("\n\n");

system("PAUSE"); return 0;

}

Task 11.2

#include <stdio.h>

#include <stdlib.h>

typedef char Contents; struct list{

Contents data; /\*data is of type contents defined above\*/ struct list \*pn;/\*pn is a pointer to a struct of type list\*/

};

typedef struct list L\_member; /\*L\_member is of type list\*/ typedef L\_member \*Link; /\*Link is a type pointer to L\_member\*/

int main()

{

Link head; head=malloc(sizeof(L\_member)); head->data='A';

head->pn=NULL;

head->pn=malloc(sizeof(L\_member)); head->pn->data='B';

head->pn->pn=NULL;

head->pn->pn=malloc(sizeof(L\_member)); head->pn->pn->data='C';

head->pn->pn->pn=NULL;

printf("\ndata in element 1=%c ",head->data); printf("\ndata in element 2=%c ",head->pn->data); printf("\ndata in element 3=%c ",head->pn->pn->data); printf("\n\n");

system("PAUSE"); return 0;

}

# Complete program 11.3

#include <stdio.h>

typedef char Contents; struct list{

Contents c; struct list \*pn;

};

typedef struct list L\_member;

typedef L\_member \*Link; /\*Link is a type pointer to L\_member\*/

Link generate(char \*s); Link assign(Contents \*q);

Link allocate(void); void list\_print(Link p); Link next(Link p);

int main()

{

Link list1, list2; list1=generate("Welcome to "); list2=generate("lists in C\n"); list\_print(list1); list\_print(list2); system("PAUSE");

return 0;

}

Link generate(char \*s)

{

Link p;

if (\*s=='\0')

{

return (NULL);

}

else

{

p=assign(s);

p->pn=generate(s+1); return(p);

}

}

Link assign(Contents \*q)

{

Link p; p=allocate(); if (p==NULL)

{

puts ("\n\*\*\*\*\*\*\*\*\*\*\*Unable to allocate storage\*\*\*\*"); exit(0);

}

else

{

p->c=\*q;

p->pn=NULL;

return(p);

}

}

Link allocate(void)

{

return((Link) malloc(sizeof(L\_member)));

}

void list\_print(Link p)

{

if (p==NULL)

{

printf("Null: (%6u)\n",p);

}

else

{

printf("%c (%6u)--> ",p->c,p); list\_print(next(p));

}

}

Link next(Link p)

{

return(p->pn);

}

# Program for the exercise on page 260 of the module textbook.

#include <stdio.h>

typedef char Contents; struct list{

Contents c; struct list \*pn;

};

typedef struct list L\_member;

typedef L\_member \*Link; /\*Link is a type pointer to L\_member\*/

Link generate(char \*s); Link assign(Contents \*q); Link allocate(void);

void list\_print(Link p); Link next(Link p);

Link get\_list(void);

Link get\_string(char \*s);

int main()

{

/\*Define a variable of the type Link\*/ list1= get\_list();

/\*Display the linked list\*/ system("PAUSE");

return 0;

}

void list\_print(Link p)

{

if (p==NULL)

{

printf("Null: (%6u)\n",p);

}

else

{

printf("%c (%6u)--> ",p->c,p); list\_print(next(p));

}

}

/\*insert the other functions here

Link generate(char \*s)/\*see program 11.2\*/ Link assign(Contents \*q)/\*See program 11.2\*/ Link allocate(void) /\*See program 11.2\*/ Link next(Link p) /\*see program 11.3\*/

Link get\_list(void) /\*See program 11.5\*/

Link get\_string(char \*s) /\*See program 11.5\*/

\*/

Task 11.5

Figure 11.10