Dynamic Data Structures with C

Terry Marris October 2010

3 *Stacks*

We are familiar with static data structures such as arrays and *structs* in C. An array is static because its size is defined at compile time. Define an array size too small and users will complain.   Define an array size too large and computer memory, which is a finite resource, will be wasted. We need data structures that will expand and shrink at run-time, according to needs. We need dynamic data structures. Linear dynamic data structures include stacks, queues and ordered linked lists - linear because the data is organised as if on a line. We begin by looking at stacks, at how the data is organised, at how data is added to the stack, and at how data is removed.

3.1 Stack

A stack is a sequence of data items together with operations to add data to, and remove data from, the stack

$$\langle\ \underset{\substack{\uparrow \\ \text{head}}}{\text{sue}},\ \underbrace{\text{kim, ian, pam}}_{\substack{\uparrow \\ \text{tail}}}\ \rangle$$

The head represents the first element in the sequence. The tail is the sequence without the head.

Items are added to, and removed from, the stack at just one end. Here, item *sue* is the most recently added item, and will be the first to be removed. Item *pam* has been in the stack for the longest time and will be the last to be removed. A stack is described as being a first-in/last-out (FILO) data structure.

A familiar use of stacks is the undo/redo feature of packages such as word processors. Operations such as typing, clear, bold, ... are placed on a stack as you use them, enabling you to undo your operations in order, the most recent operation first.

3.2 Stack Interface

We introduce the interface to our *Stack* type. We specify *Stack* as a pointer to a *StackHeader* (which is defined in the implementation), a stack iterator, a duplicate function, and a collection of function prototypes.

```
/* stack.h */

#ifndef STACK
#define STACK

typedef struct StackHeader *Stack;
typedef struct StackIterStruct *StackIterator;
typedef void *(*Duplicate)(void *);

/* stackError: reports stack errors, terminates program execution */
int stackError(const char *e);

/* newStack: returns a new, empty stack */
Stack newStack(Duplicate);

/* disposeStack: deallocates the memory occupied by Stack */
Stack disposeStack(Stack *s);

/* push: adds a data item to the stack */
Stack push(void * const data, const Stack s);

/* peek: returns item at top of stack */
void *peek(const Stack s);

/* pop: removes the top item in the stack */
Stack pop(const Stack s);

/* isEmptyStack: returns 1 if stack has no elements */
int isEmptyStack(const Stack s);

/* reverseStack: reverses the order of the elements in the given
   stack */
Stack reverseStack(const Stack s);

/* copyStack: returns a copy of the given stack */
Stack copyStack(const Stack s);

/* newStackIterator: returns a new iterator for the given stack */
StackIterator newStackIterator(Stack);

/* stackIteratorHasNext: returns 1 if there is an element not
   yet visited in the current iteration */
int stackIteratorHasNext(StackIterator);

/* stackIteratorNext: returns the next element in the stack. */
void *stackIteratorNext(StackIterator);

/* disposeStackIterator: deallocates the memory occupied by the
   iterator */
StackIterator disposeStackIterator(StackIterator *);

#endif
```

A stack is defined as a pointer to a stack header. The stack header is defined in the implementation, *stack.c*.

```
typedef struct StackHeader *Stack;
```

An iterator visits each element in a collection in turn. A stack iterator is a pointer to a stack iterator structure, which is also defined in the implementation.

```
typedef struct StackIterStruct *StackIterator;
```

We rely on the user to supply a function that returns a duplicate copy of an element stored in the stack.

```
typedef void *(*Duplicate)(void *);
```

Who is the user? The user is a programmer using our stack for their own purposes.

stackError() reports errors such as *out of memory* or an *attempt to retrieve an element from an empty stack*. This function also halts program execution.

The usual stack operations can be performed.

Create a new stack:

```
Stack s = newStack((Duplicate)copy);
```

where *copy()* is a duplicate copy function supplied by the user.

Add an item to the stack:

```
char ch = 'X';
s = push(&ch, s);
```

push() requires a pointer to data. We provide one by supplying the address of a *char* variable. Notice that both the data and stack parameters are qualified by *const*. This means that the intention is for both parameters to remain unchanged by the function. The updated stack is returned.

```
/* push: adds a data item to the stack */
Stack push(void * const data, const Stack s);
```

We can look at the item at the head of the stack:

```
char ch = *(char *)peek(s);
```

We rely on the programmer using our stack to know the data type of the items added, and, consequently, the items retrieved from the stack. *peek()* returns a pointer to the item retrieved. The programmer is obliged to make an appropriate cast. We remember that any pointer can be cast to *void* * and back again without loosing information, and that pointer arguments are automatically cast to *void* * parameters. So, this statement says: cast the pointer returned by *peek()* to *char* *, then use the indirection operator, *, to access the value pointed to.

We can remove an item from a stack:

```
s = pop(s);
```

We can see if a stack is empty:

```
if isEmptyStack(s) ...
```

We can return, to the operating system, the memory claimed for a stack .

```
disposeStack(&s);
```

Notice that we supply the address of the stack to be disposed of.

In addition, we have an iterator to visit and return each element in the stack. We create a new iterator and provide it with the stack to be iterated over.

```
StackIterator iter = newStackIterator(s);
```

We loop for as long as there is an element in the stack not yet visited.

```
while (stackIteratorHasNext(iter))
  ...
```

We print each element retrieved from the stack

```
printf("%c ", *(char *)stackIteratorNext(iter));
```

The pointer to a structure and the function prototypes are wrapped within

```
#ifndef STACK
#define STACK
...
#endif
```

This mechanism ensures that the header file is actually translated only once. The first time the file is *#included, STACK*, which has not previously been defined, is defined. The second (and subsequent) times the file is included *STACK* has already been defined and so processing skips to the *#endif*. So the compiler translates the contents of the header file just the once no matter how many times the file is *#include*d.

stack.h contains all a programmer needs to know when using our *Stack* type.

3.3 Testing

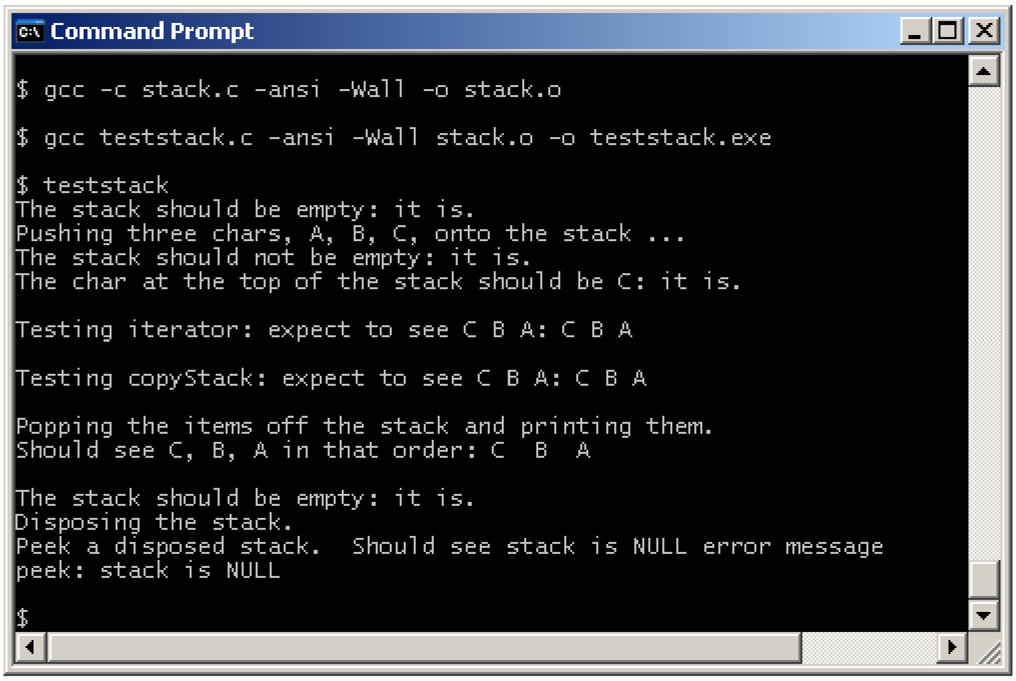
We supply a copy function, and test each stack function looking for errors.

```
/* TestStack.c */

#include <stdio.h>
#include <stdlib.h>
#include "stack.h"

/* copyChar: returns a duplicate copy of its parameter */
char *copyChar(char *c)
{
  char *t = malloc(sizeof(char));
  t = c;
  return t;
}
```

```
int main()
{
  Stack s = newStack((Duplicate)copyChar);
  printf("The stack should be empty: ");
  if (isEmptyStack(s))
    printf("it is.\n");
  else
    printf("it is not.\n");

  printf("Pushing three chars, A, B, C, onto the stack ... \n");
  char ch = 'A';
  char ch2 = 'B';
  char ch3 = 'C';
  s = push(&ch, s);
  s = push(&ch2, s);
  s = push(&ch3, s);
  printf("The stack should not be empty: ");
  if (!isEmptyStack(s))
    printf("it is.\n");
  else
    printf("it is not.\n");
  printf("The char at the top of the stack should be C: ");
  if (*(char *)peek(s) == 'C')
    printf("it is.\n");
  else
    printf("it is not.\n");
  printf("\n");

  printf("Testing iterator: expect to see C B A: ");
  StackIterator iter = newStackIterator(s);
  while (stackIteratorHasNext(iter))
    printf("%c ", *(char *)stackIteratorNext(iter));
  printf("\n\n");

  printf("Testing copyStack: expect to see C B A: ");
  Stack t = copyStack(s);
  iter = newStackIterator(t);
  while (stackIteratorHasNext(iter))
    printf("%c ", *(char *)stackIteratorNext(iter));
  printf("\n\n");

  printf("Popping the items off the stack and printing them.\n");
  printf("Should see C, B, A in that order: ");
  while (!isEmptyStack(s)) {
    printf("%c  ", *(char *)peek(s));
    s = pop(s);
  }
  printf("\n\n");

  printf("The stack should be empty: ");
  if (isEmptyStack(s))
    printf("it is.\n");
  else
    printf("it is not.\n");

  printf("Disposing the stack.\n");
  disposeStack(&s);

  printf("Peek a disposed stack.
          Should see stack is NULL error message \n");
  ch = *(char *)peek(s);
```

```
          printf("%c\n", ch);

          return 0;
        }
```

A program run is shown below.



```
$ gcc -c stack.c -ansi -Wall -o stack.o

$ gcc teststack.c -ansi -Wall stack.o -o teststack.exe

$ teststack
The stack should be empty: it is.
Pushing three chars, A, B, C, onto the stack ...
The stack should not be empty: it is.
The char at the top of the stack should be C: it is.

Testing iterator: expect to see C B A: C B A

Testing copyStack: expect to see C B A: C B A

Popping the items off the stack and printing them.
Should see C, B, A in that order: C   B   A

The stack should be empty: it is.
Disposing the stack.
Peek a disposed stack.  Should see stack is NULL error message
peek: stack is NULL

$
```

gcc -c stack.c -ansi -Wall -o stack.o says load the GNU C Compiler and compile *stack.c* to object code. The *-ansi* and *-Wall* switches ensure we use standard ansi C.

We could give our *stack.o* file to another programmer to link in with their programs, as shown below:

gcc teststack.c -ansi -Wall stack.o -o teststack.exe

The $ is just my system prompt, and the MSDOS environment is my preferred choice for developing C programs.

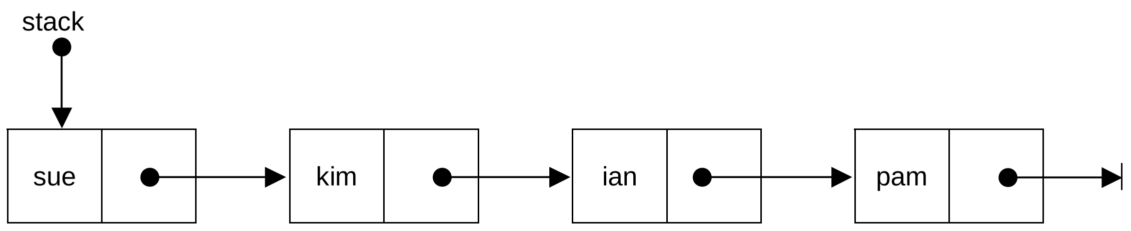
How do we create *stack.c*? We are coming to that.

3.3 Stack Implementation

A stack is a sequence of data items. Items are added and removed from the same end.

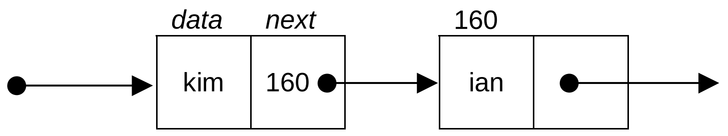
3.3.1 Nodes

We can picture the elements of a stack as a sequence of *nodes*.

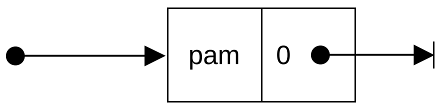


Each node has two component parts. One part holds a data item (or, more usually, a pointer to a data item). The second part contains a pointer to the next node (if there is one).

A pointer is just a variable that holds the address of another variable. Each variable is located at some address in a computer's memory. So, for example, if the node with the data value *ian* is located at address 160, then the variable that points to it has the value 160.



A next node pointer that does not point to a node has the null value, 0. This guarantees that it points to nothing important.



We specify the stack node type as a pointer to a C structure named *StackNodeStruct*.

```
typedef struct StackNodeStruct *StackNode;

struct StackNodeStruct {
  void *data;
  StackNode next;
};
```

*void ** (pointer to *void*) is a generic pointer. A pointer to any data type can be cast to *void ** and back again without loss of information. *void * data* allows us to store data of any type in the stack.

The *next* member is a pointer to a *StackNodeStruct*. The definition of *StackNode* is recursive; it contains a reference to itself. *StackNode* is a pointer to a *StackNodeStruct*. *StackNodeStruct* contains a *StackNode* member.

3.3.2 New Node

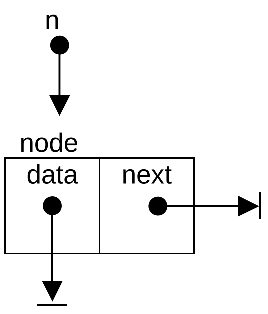
To create a new node we make a call to the memory allocator function named *malloc()*.

```
StackNode n = malloc(sizeof(struct StackNodeStruct));
```

malloc() returns a pointer to the requested size of memory, or *NULL* if there is not enough space available. *malloc()* returns a pointer to *void*. So we can assign the value returned by *malloc()* to any pointer variable no problem. *malloc()* is defined in *stdio.h*.

Having successfully created a new node we assign *NULL* to both members.

```
/* newStackNode: returns a new, empty stack node */
StackNode newStackNode()
{
 StackNode n = malloc(sizeof(struct StackNodeStruct));
  if (n == NULL)
    stackError("newStackNode: out of memory");
  n->next = NULL;
  n->data = NULL;
  return n;
}
```



The function returns a pointer to the new node. Calls to *newNode()* are typically made by the *push()* and *newStack()* functions.

3.3.3 Dispose Node

Every call to *malloc()* should be matched with a corresponding call to *free()*. Why? Because repeated calls to *malloc()* can eventually consume all available memory and cause unexpected and unwanted program behaviour.

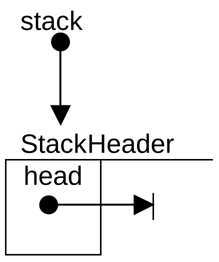
```
/* disposeStackNode: releases memory occupied by StackNode */
StackNode disposeStackNode(StackNode *n)
{
  if (*n == NULL)
    stackError("disposeStackNode: node is NULL");
  free((*n)->data);
  free(*n);
  *n = NULL;
  return *n;
}
```

free() deallocates memory previously allocated by *malloc()*. We assign *NULL* to the pointer after deallocating it; this prevents unexpected results if the pointer is accidentally reused.

3.3.4 Stack

The stack data structure looks like:

```
typedef struct StackHeader *Stack;

struct StackHeader {
  StackNode head;
  Duplicate duplicate;
};
```



A stack is a pointer to a stack header, whose member, head, points to the first node in the sequence, or to null if there are no nodes. notice that we have a member for the pointer to a duplicate function provided by the user.

3.3.5 New Stack

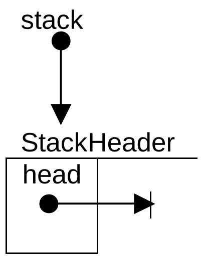
The creation of a new, empty stack is straightforward.

```
/* newStack: returns a new, empty stack */
Stack newStack(Duplicate dup)
{
  Stack s = malloc(sizeof(struct StackHeader));
  if (s == NULL)
    stackError("newStack: out of memory");
  s->head = NULL;
  s->duplicate = dup;
  return s;
}
```

A new stack has no nodes.

3.3.6 Empty Stack

A stack is empty if its *head* pointer contains *NULL*.



```
/* isEmptyStack: returns 1 if stack has no elements */
int isEmptyStack(const Stack s)
{
  if (s == NULL)
    stackError("isEmptyStack: stack is NULL");
  return s->head == NULL;
}
```

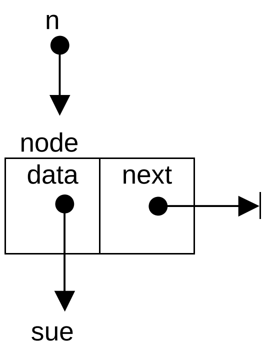
3.3.7 Push

We add a new node, with a data item, to the front of the sequence of nodes. First, we create a new stack, *t*.

```
Stack t = newStack(s->duplicate);
```

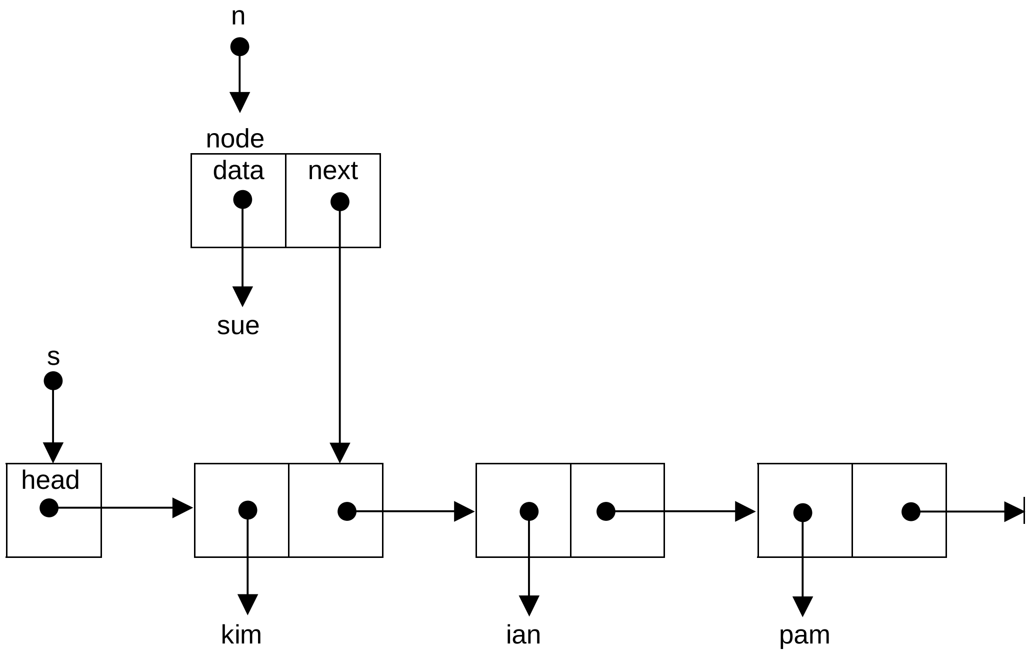
Then we create a new node and copy the given data item, e.g. *sue*.

```
  StackNode n = newStackNode();
  n->data = s->duplicate(data);
```



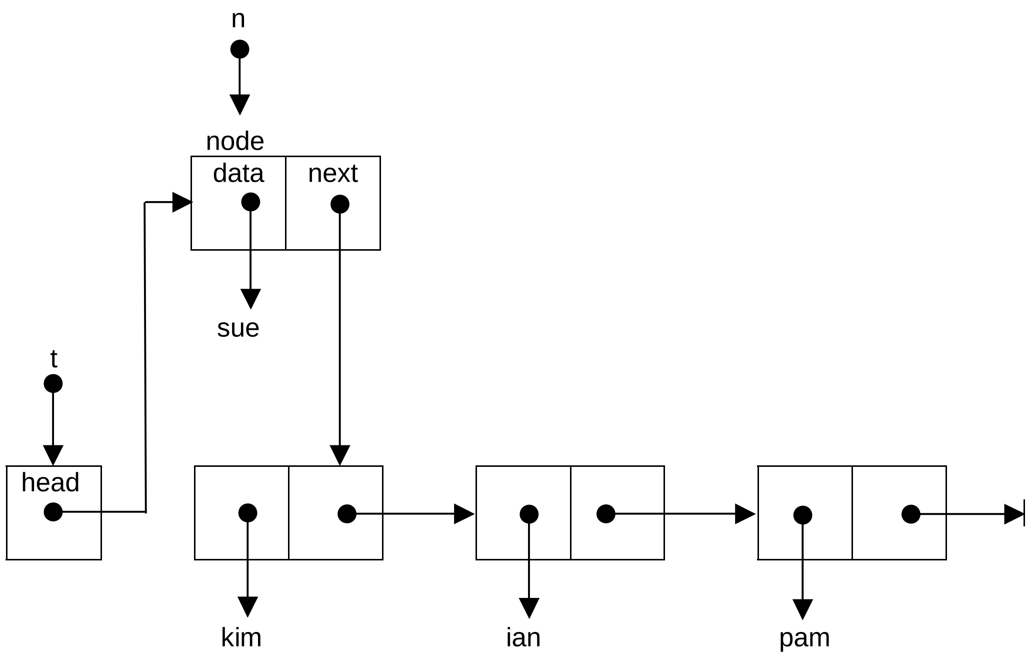
We update the next member of the new node with the stack's head; *n->next* now refers to the tail of the stack.

```
  n->next = s->head;
```

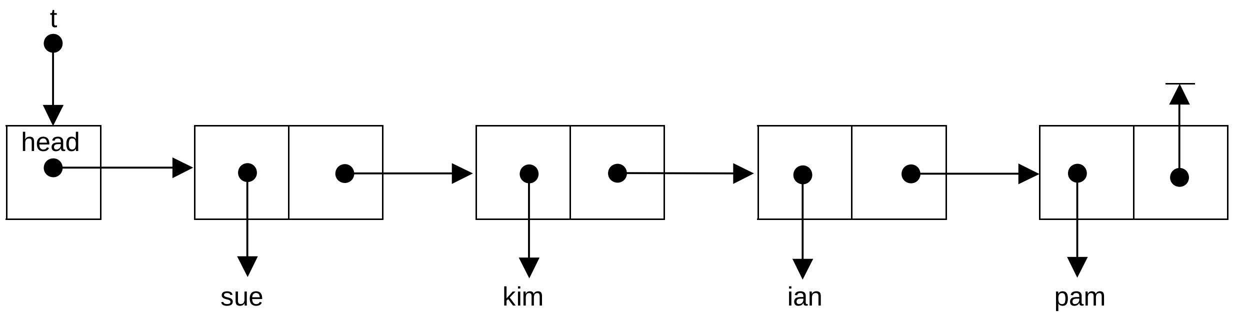


We update the new stack's head to point to the new node.

```
t->head = n;
```



Straightening out the diagram:



The entire function is:

```
/* push: adds a data item to the stack */
Stack push(void * const data, const Stack s)
{
  if (s == NULL)
    stackError("push: stack is NULL");
  Stack t = newStack(s->duplicate);
  StackNode n = newStackNode();
  n->data = s->duplicate(data);
  n->next = s->head;
  t->head = n;
  return t;
}
```

3.3.8 Peek

peek() lets us examine the item at the head of a stack. The stack remains unchanged.

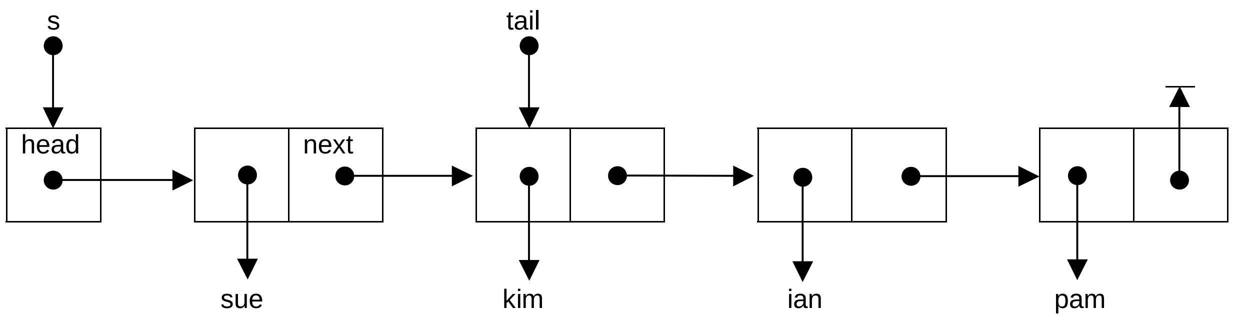
```
/* peek: returns item at top of stack */
void *peek(const Stack s)
{
  if (s == NULL)
    stackError("peek: stack is NULL");
  if (isEmptyStack(s))
    stackError("peek: stack is empty");
  return s->duplicate(s->head->data);
}
```

3.3.9 Pop

We remove the item at the front of the sequence of nodes.

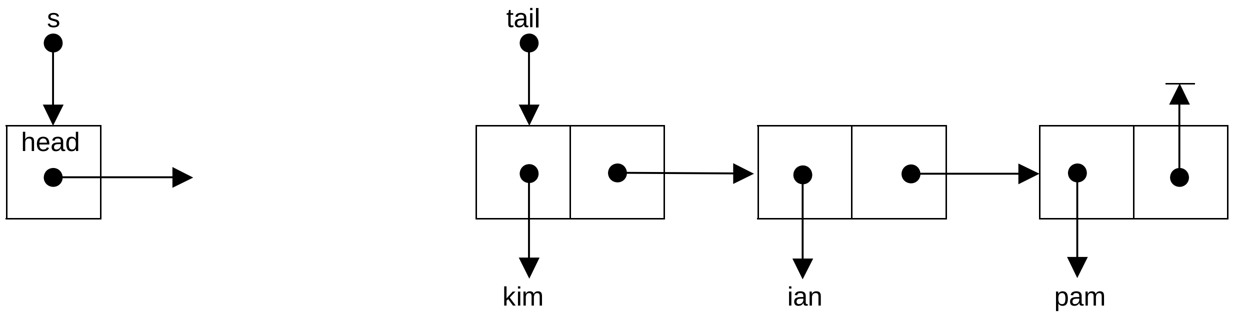
We place an anchor on the tail of the stack.

```
StackNode tail = s->head->next;
```



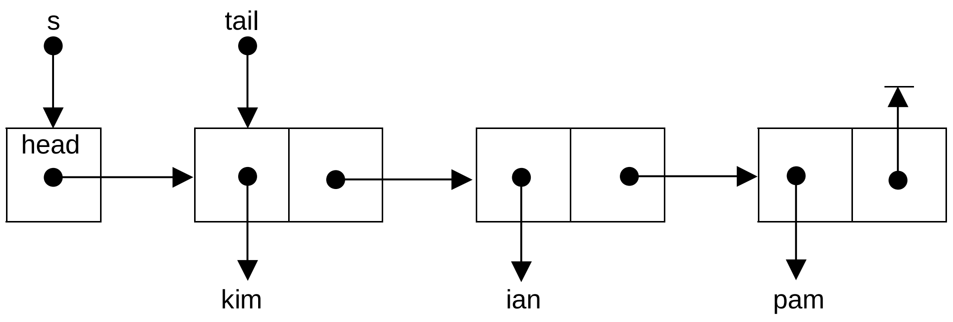
We zap the node pointed to by head.

```
disposeStackNode(&s->head);
```



Finally, we update the stack's head with the tail.

```
s->head = tail;
```



The entire function is:

```
/* pop: removes the top item in the stack */
Stack pop(const Stack s)
{
  if (s == NULL)
    stackError("pop: stack is NULL");
  if (isEmptyStack(s))
    stackError("pop: stack is empty");
  Stack t = newStack(s->duplicate);
  t->head = s->head->next;
  disposeStackNode(&s->head);
  return t;
}
```

3.3.10 Copy Stack

One way of creating a duplicate copy of a stack is reverse its contents twice.

```
/* reverseStack: reverses the order of the elements in the given
   stack */
Stack reverseStack(const Stack s)
{
  if (s == NULL)
    stackError("copyStack: stack is NULL");
  Stack t = newStack(s->duplicate);
  StackNode p;
  for (p = s->head; p != NULL; p = p->next) {
    StackNode n = newStackNode();
    n->data = t->duplicate(p->data);
    n->next = t->head;
    t->head = n;
  }
  return t;
}
```

We march a pointer, *p*, along the stack, copying the data in each node to new nodes and inserting each new node at the head of a stack.

Then we use *reverseStack()* twice to create a duplicate copy of a stack.

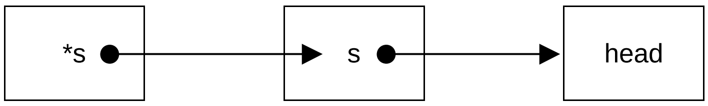
```
/* copyStack: returns a copy of the given stack */
Stack copyStack(const Stack s)
{
  Stack t = reverseStack(s);
  return reverseStack(t);
}
```

3.3.11 Dispose Stack

We return to the operating system the memory previously reserved for the stack.

```
/* disposeStack: deallocates the memory occupied by Stack */
Stack disposeStack(Stack *s)
{
  if (*s == NULL)
    stackError("disposeStack: stack is NULL");
  while (!isEmptyStack(*s))
    *s = pop(*s);
  free(*s);
  *s = NULL;
  return *s;
}
```

We are intent on amending the pointer to a stack header itself. So we use a pointer to the pointer *(Stack *s)*.



We repeatedly call *pop()* to empty the stack of nodes. Then we call *free(*s)*. Here, the * is the indirection operator. So **s* refers to *s*. It is what *s* points to that we want to free. It is *s* itself that we want to set to *NULL*.

So, if we have *Stack s = NULL*, it would be an error to attempt to push data onto it, or to pop data from it. so we include the guard

```
if (*s == NULL)
  stackError("disposeStack: stack is NULL");
```

in our stack handing functions in case *pop()* or *push()* get called with a *NULL* stack.

3.3.12 Iterator

An iterator visits each element in turn in a collection.

We initialise the iterator with the stack to be iterated over.

```
/* newStackIterator: returns a new iterator for the given stack */
StackIterator newStackIterator(Stack s)
{
  StackIterator iter = malloc(sizeof(struct StackIterStruct));
  if (iter == NULL)
   stackError("newStackIterator: out of memory");
  if (s == NULL)
    stackError("newStackIterator: stack is NULL");
  iter->stack = s;
  iter->current = NULL;
  return iter;
}
```

current is used to refer to the node currently being processed. Initially, *current* is *NULL*.

Before we can move the iterator on to the next node we have to check that there is a node to be moved on to.

```
/* stackIteratorHasNext: returns 1 if there is an element not
   yet visited in the current iteration */
int stackIteratorHasNext(StackIterator iter)
{
  if (iter->stack == NULL)
    stackError("stackIteratorHasNext: stack is NULL");
  if (isEmptyStack(iter->stack))
    return 0;
  if (iter->current == NULL)
    return iter->stack->head != NULL;
  return iter->current->next != NULL;
}
```

If there is a next node, we move onto it and return its data item.

```
/* stackIteratorNext: returns the next element in the stack. */
void *stackIteratorNext(StackIterator iter)
{
  if (!stackIteratorHasNext(iter))
    stackError("stackIteratorNext: no next element");
  if (iter->current == NULL)
     iter->current = iter->stack->head;
  else
    iter->current = iter->current->next;
  return iter->stack->duplicate(iter->current->data);
}
```

3.3.13 Implementation

The code for the entire implementation is shown below.

```
/* stack.c */

#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
#include "stack.h"

typedef struct StackNodeStruct *StackNode;

struct StackNodeStruct {
  void *data;
  StackNode next;
};

struct StackHeader {
  StackNode head;
  Duplicate duplicate;
};

struct StackIterStruct {
  Stack stack;
  StackNode current;
};
```

```c
/* stackError: reports stack errors, terminates program execution */
int stackError(const char *error)
{
  printf("%s\n", error);
  exit(1);
}

/* newStack: returns a new, empty stack */
Stack newStack(Duplicate dup)
{
  Stack s = malloc(sizeof(struct StackHeader));
  if (s == NULL)
    stackError("newStack: out of memory");
  s->head = NULL;
  s->duplicate = dup;
  return s;
}

/* isEmptyStack: returns 1 if stack has no elements */
int isEmptyStack(const Stack s)
{
  if (s == NULL)
    stackError("isEmptyStack: stack is NULL");
  return s->head == NULL;
}

/* newStackNode: returns a new, empty stack node */
StackNode newStackNode()
{
 StackNode n = malloc(sizeof(struct StackNodeStruct));
  if (n == NULL)
    stackError("newStackNode: out of memory");
  n->next = NULL;
  n->data = NULL;
  return n;
}

/* push: adds a data item to the stack */
Stack push(void * const data, const Stack s)
{
  if (s == NULL)
    stackError("push: stack is NULL");
  Stack t = newStack(s->duplicate);
  StackNode n = newStackNode();
  n->data = s->duplicate(data);
  n->next = s->head;
  t->head = n;
  return t;
}

/* peek: returns item at top of stack */
void *peek(const Stack s)
{
  if (s == NULL)
    stackError("peek: stack is NULL");
  if (isEmptyStack(s))
    stackError("peek: stack is empty");
  return s->duplicate(s->head->data);
}
```

```
/* disposeStackNode: releases memory occupied by StackNode */
StackNode disposeStackNode(StackNode *n)
{
  if (*n == NULL)
    stackError("disposeStackNode: node is NULL");
  free((*n)->data);
  free(*n);
  *n = NULL;
  return *n;
}

/* pop: removes the top item in the stack */
Stack pop(const Stack s)
{
  if (s == NULL)
    stackError("pop: stack is NULL");
  if (isEmptyStack(s))
    stackError("pop: stack is empty");
  Stack t = newStack(s->duplicate);
  t->head = s->head->next;
  disposeStackNode(&s->head);
  return t;
}

/* reverseStack: reverses the order of the elements in the given
stack */
Stack reverseStack(const Stack s)
{
  if (s == NULL)
    stackError("copyStack: stack is NULL");
  Stack t = newStack(s->duplicate);
  StackNode p;
  for (p = s->head; p != NULL; p = p->next) {
    StackNode n = newStackNode();
    n->data = t->duplicate(p->data);
    n->next = t->head;
    t->head = n;
  }
  return t;
}

/* copyStack: returns a copy of the given stack */
Stack copyStack(const Stack s)
{
  Stack t = reverseStack(s);
  return reverseStack(t);
}

/* disposeStack: deallocates the memory occupied by Stack */
Stack disposeStack(Stack *s)
{
  if (*s == NULL)
    stackError("disposeStack: stack is NULL");
  while (!isEmptyStack(*s))
    *s = pop(*s);
  free(*s);
  *s = NULL;
  return *s;
}
```

```
/* newStackIterator: returns a new iterator for the given stack */
StackIterator newStackIterator(Stack s)
{
  StackIterator iter = malloc(sizeof(struct StackIterStruct));
  if (iter == NULL)
   stackError("newStackIterator: out of memory");
  if (s == NULL)
    stackError("newStackIterator: stack is NULL");
  iter->stack = s;
  iter->current = NULL;
  return iter;
}

/* disposeStackIterator: deallocates the memory occupied by the
iterator */
StackIterator disposeStackIterator(StackIterator *iter)
{
  free(*iter);
  *iter = NULL;
  return *iter;
}

/* stackIteratorHasNext: returns 1 if there is an element not
   yet visited in the current iteration */
int stackIteratorHasNext(StackIterator iter)
{
  if (iter->stack == NULL)
    stackError("stackIteratorHasNext: stack is NULL");
  if (isEmptyStack(iter->stack))
    return 0;
  if (iter->current == NULL)
    return iter->stack->head != NULL;
  return iter->current->next != NULL;
}

/* stackIteratorNext: returns the next element in the stack. */
void *stackIteratorNext(StackIterator iter)
{
  if (!stackIteratorHasNext(iter))
    stackError("stackIteratorNext: no next element");
  if (iter->current == NULL)
     iter->current = iter->stack->head;
  else
    iter->current = iter->current->next;
  return iter->stack->duplicate(iter->current->data);
}
```

Exercise 3.1

1. Design, write and test a function that returns the length of the stack i.e. the number of data items (nodes) it contains. Possible ways of proceeding include:
    a. have a new member, named count perhaps, in the stack header, increment it whenever you push, decrement it whenever you pop, or
    b. have a pointer that marches along the nodes in the stack, incrementing a count as it does so.

2. Test that the stack, described above, functions as expected with variable-length strings

Bibliography

Kernighan B, Ritchie D, *The C Programming Language,* Prentice Hall, 1988
Mark Williams Company, *ANSI C - A Lexical Guide,* Prentice Hall 1988
Dromey R, *How to Solve it by Computer,* Prentice Hall 1982