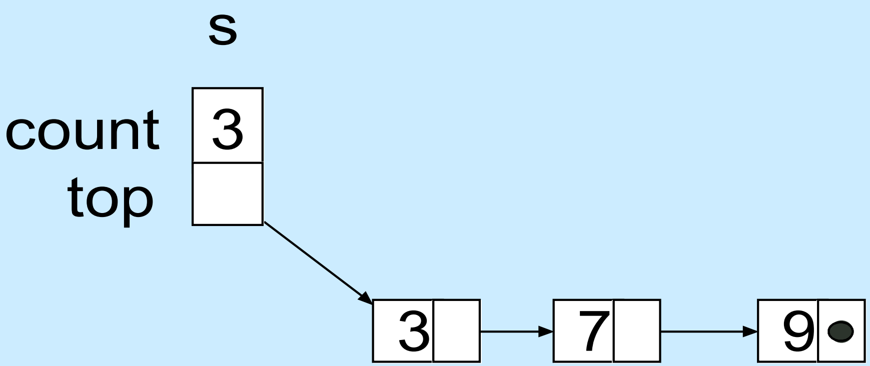
Linked Implementation of the Stack ADT

```
/*-----------------------------------------------*/
/*------------- stackDefs.h ---------------*/
/*-----------------------------------------------*/

#include “stackEntry.h"        /* defines StackEntry
*/

typedef struct stacknode {
    StackEntry info;
    struct stacknode* next;
} StackNode;

typedef struct stack {
    int  count;            /* Often not included in the definition */
    StackNode* top;
} Stack;
```


s
count
3
top
3
7
9

```
/*---------------- stack.c ---------------*/
#include "stack.h"
int InitStack(Stack *s)
{
    S->top = NULL;
    s->count = 0;
    return 1;
}
```

```
/*---------------- stack.c ---------------*/

int Push(StackEntry item, Stack *s)
{
  StackNode* tmp = malloc(sizeof(StackNode));

  if (tmp == NULL) {
      printf("Cannot push onto a full stack");
      return 0;
  }
  s->count++;
  tmp->info = item;
  tmp->next = s->top;
  s->top = tmp;
  return 1;
}
```

```
int Pop(StackEntry *item, Stack *s)
{
  StackNode* tmp = s->top;

  if (StackEmpty(s)) {
      printf("Cannot pop an empty stack");
      return 0;
  }

  s->count--;
  *item = s->top->info;
  tmp = s->top;
  s->top = s->top->next;
  free(tmp);
  return 1;
}
```

```
int StackEmpty(const Stack*s )
{
    return (s->top == NULL);
}


int StackFull(const Stack *s)
{
  StackNode* tmp = malloc(sizeof(StackNode));
  if (tmp == NULL)
     return 1;
  else {
     free(tmp);
     return 0;
  }
}
```

```
int StackSize(const  Stack* s)
{
  return s->count;
}



int StackTop(StackEntry* x, const Stack* s)
{
  if (StackEmpty(s)) {
      Warning("Cannot access an empty stack");
      return 0;
  }
  *x =  s->top->info;
  return 1;
}
```

```
void ClearStack(Stack* s)
{
  StackNode* tmp;

  while (s->top != NULL) {
      tmp = s->top;
      s->top = s->top->next;
      free(tmp);
  }

  s->count = 0;
}
```

Testing the Implementation

- Let’s add some code to test the stack ADT implementation.

- New file stack_test.c

stack_test.c

```
/* Trusted functions:  StackEmpty, StackSize */
#include <stdio.h>
#include "stack_tests.h" /* contains define for MAX */
#include "stack.h“
int main()
{
  int i = 0;
  StackEntry x;
  Stack S;
  char response;
  printf ("Stack Test starting\n");
  printf("InitStack: ")
  InitStack(&S);
  if (!StackEmpty(&S))
  {
      printf ("failed (stack is not empty after init)\n");
  } else {
      printf("passed\n");
  }
```

stack_test.c

```c
  printf("ClearStack: ");
  ClearStack(&S);
  if (StackEmpty(&S))
      printf("passed\n");
  else
      printf("failed\n");
  printf("\n");
  printf ("Stack test complete\n");
  return 0;
}
```

Optional Material:

Reversing Input Example

Array Implementation

Example: reversing input

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

int main()
{
   Stack S;
   int buffer = 0;

   InitStack(&S);

   printf("Enter integers, one after another, ending input with 0\n");
```

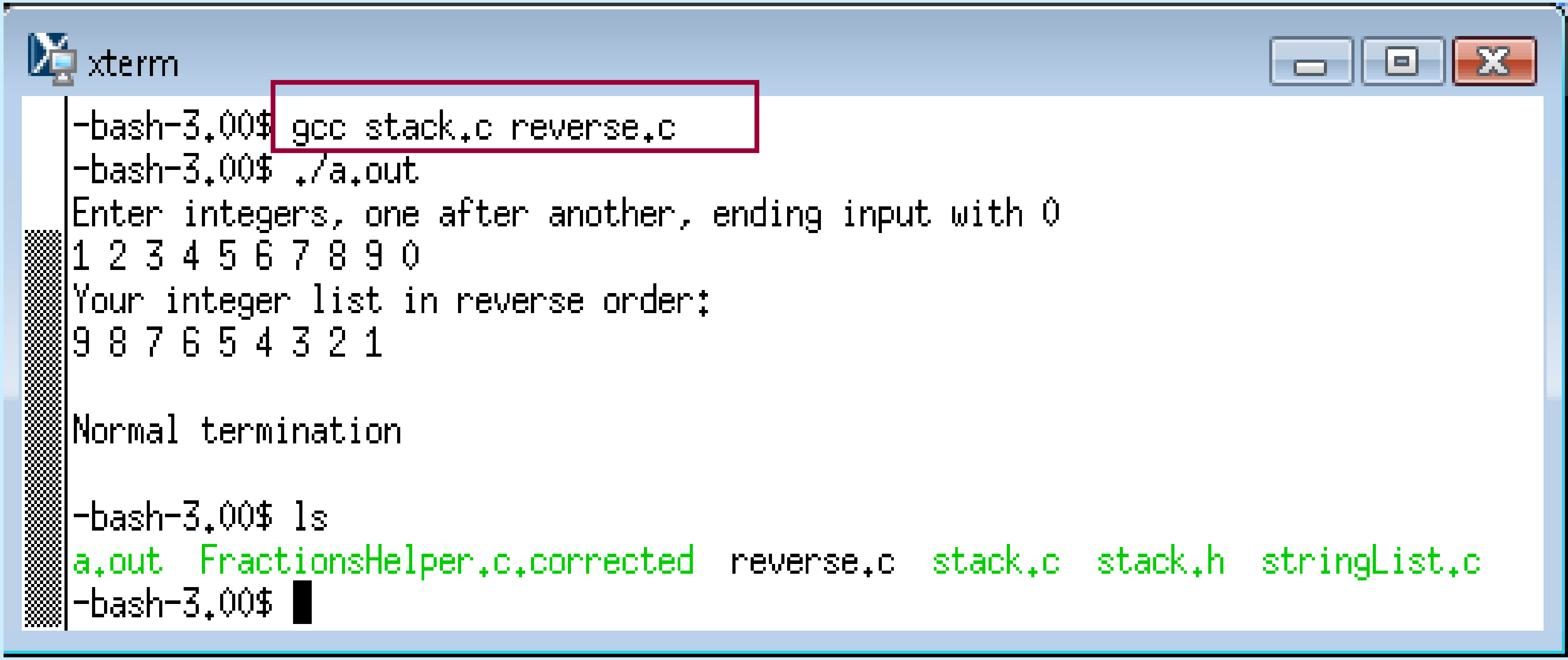
Example: reversing input

```
scanf("%d",&buffer);

while(buffer != 0)
{
   Push(buffer,&S);
   scanf("%d",&buffer);
}
```

Example: reversing input

```
  printf("Your integer list in reverse order:\n");

  while(!StackEmpty(&S))
  {
    Pop(&buffer,&S);
    printf("%d ",buffer);
  }
  printf("\n\nNormal termination\n\n");
  return 0;
}
```

Example: reversing input



```
-bash-3.00$ gcc stack.c reverse.c
-bash-3.00$ ./a.out
Enter integers, one after another, ending input with 0
1 2 3 4 5 6 7 8 9 0
Your integer list in reverse order:
9 8 7 6 5 4 3 2 1

Normal termination

-bash-3.00$ ls
a.out  FractionsHelper.c.corrected  reverse.c  stack.c  stack.h  stringList.c
-bash-3.00$ █
```

Stack Definitions File

-
  ```
  /* stackEntry.h
     Supplies the  typedef for the stack element type
  */
  ```

-
  ```
  typedef int StackEntry;
  ```

Example: reversing input

```
scanf("%d",&buffer);

while(buffer != 0)
{
   Push(buffer,&S);
   scanf("%d",&buffer);
}
```

Stack Definitions File

- /* stackDefs.h
  Supplies the typedef for the stack type
  */

- #include “stackEntry.h”;

- #define MAXSTACK 50

- typedef struct stack {
-     int nextAvail;
-     StackEntry entry[MAXSTACK];
- } Stack;

Stack S is empty if and only if S.nextAvail is 0.

Otherwise, the values S.entry[0], , ... ,S.entry[nextAvail-1]
are the values in the stack S

S.entry[0] is at the ***bottom*** and *S.entry[nextAvail-1]* is at the ***top***.

The InitStack function just sets the *nextAvail* variable to 0.

```
InitStack(&S);
Push(6,&S);
Push(2,&S);
Push(9,&S);
Pop(&hold,&S);
Pop(&hold,&S);
Push(1,&S);
ClearStack(&S);
```

nextAvail	**0**					
i	[0]	[1]	[2]	[3]	[4]	[5]
entry[i]	?	?	?	?	?	?

```
InitStack(&S);
Push(6,&S);
Push(2,&S);
Push(9,&S);
Pop(&hold,&S);
Pop(&hold,&S);
Push(1,&S);
ClearStack(&S);
```

nextAvail	**1**					
i	[0]	[1]	[2]	[3]	[4]	[5]
entry[i]	**6**	?	?	?	?	?

```
InitStack(&S);
Push(6,&S);
Push(2,&S);
Push(9,&S);
Pop(&hold,&S);
Pop(&hold,&S);
Push(1,&S);
ClearStack(&S);
```

nextAvail	**2**					
i	[0]	[1]	[2]	[3]	[4]	[5]
entry[i]	**6**	**2**	?	?	?	?

```
InitStack(&S);
Push(6,&S);
Push(2,&S);
Push(9,&S);
Pop(&hold,&S);
Pop(&hold,&S);
Push(1,&S);
ClearStack(&S);
```

nextAvail	**3**					
i	[0]	[1]	[2]	[3]	[4]	[5]
entry[i]	**6**	**2**	**9**	?	?	?

```
InitStack(&S);
Push(6,&S);
Push(2,&S);
Push(9,&S);
Pop(&hold,&S);
Pop(&hold,&S);
Push(1,&S);
ClearStack(&S);
```

nextAvail	**2**					
i	[0]	[1]	[2]	[3]	[4]	[5]
entry[i]	**6**	**2**	**9**	?	?	?

```
InitStack(&S);
Push(6,&S);
Push(2,&S);
Push(9,&S);
Pop(&hold,&S);
Pop(&hold,&S);
Push(1,&S);
ClearStack(&S);
```

nextAvail	**1**					
i	[0]	[1]	[2]	[3]	[4]	[5]
entry[i]	**6**	**2**	**9**	?	?	?

```
InitStack(&S);
Push(6,&S);
Push(2,&S);
Push(9,&S);
Pop(&hold,&S);
Pop(&hold,&S);
Push(1,&S);
ClearStack(&S);
```

nextAvail	**2**					
i	[0]	[1]	[2]	[3]	[4]	[5]
entry[i]	**6**	**1**	**9**	?	?	?

```
InitStack(&S);
Push(6,&S);
Push(2,&S);
Push(9,&S);
Pop(&hold,&S);
Pop(&hold,&S);
Push(1,&S);
ClearStack(&S);
```

nextAvail	**0**					
i	[0]	[1]	[2]	[3]	[4]	[5]
entry[i]	6	1	9	?	?	?

With the above example, you should be able to understand the code below. Recall

```
typedef struct stack {
     int  nextAvail;
     StackEntry entry[MAXSTACK];
 } Stack;
```

```
/* InitStack: initialize the stack to be empty.

Pre:    None.
Post:   The stack has been initialized to be  empty.
 */


          int InitStack(Stack *s)
          {
              s->nextAvail = 0;
              return 1;
          }
```

```
/* Push: push an item onto the stack.

 Pre:  The stack exists and it is not full.
Post:  The argument item has been stored at the top of the stack.
*/


int Push(StackEntry item, Stack *s)
{
    if (StackFull(s)) {
        Warning("Stack is full");
        return 0;
    }
    else
        s->entry[s->nextAvail++] = item;

    return 1;
}
```

```
    /* Pop: pop an item from the stack.

    Pre:   The stack exists and it is not empty.
    Post:  The item at the top of stack has
    been
              removed and returned in *item.
    */
int Pop(StackEntry *item, Stack *s)
{
    if (StackEmpty(s)) {
        Error("Stack is empty");
        return 0;
    }
    else
        *item = s->entry[--s->nextAvail];

    return 1;
}
```

```
/* StackEmpty: returns non-zero if the stack  is empty.

Pre:    The stack exists and it has been initialized.
Post:  Returns 1 if the stack is empty;
           returns 0, otherwise.
 */


      int StackEmpty(const Stack *s)
      {
          return (s->nextAvail <= 0);
      }
```

```
/* StackFull: test to see if the stack is full.

Pre:    The stack exists and it has been initialized.
Post:   Returns 1 if the stack is full; otherwise returns 0.
 */

int StackFull(const Stack *s)
{
    return (s->nextAvail >= MAXSTACK);
}
```

```
/* StackTop.

Pre:   The stack exists and it is not empty.
Post:  The item at the top of stack has been
           returned in *item.  The stack is unchanged
*/

int StackTop(StackEntry *item, const Stack *s)
{
    if (StackEmpty(s)) {
        printf("Stack is empty");
        return 0;
    }
    else
        *item = s->entry[s->nextAvail-1];

    return 1;
}
```

```
 /* ClearStack: makes the stack empty.
Pre:     The stack exists and has been
initialized
Post:   The stack is empty.
 */
int ClearStack(Stack *s)
{
    s->nextAvail = 0;
    return 1;
}
```

```
int StackSize(const  Stack* s)
{
  return s->nextAvail;
}
```