Queues, Stack Modules, and Abstract Data Types

CS2023 Winter 2004
Outcomes: Queues, Stack Modules, and Abstract Data Types

- *C for Java Programmers*, Chapter 11 (11.5) and *C Programming - a Modern Approach*, Chapter 19

- After the conclusion of this section you should be able to
  - Write queue and stack modules using a linked list
  - Use `static` to modify the linkage of functions and global variables
  - Use opaque data types to create modules that implement abstract data types
Queue

Queue can be implemented as a linked list:

typedef unsigned int data;
typedef struct node{
    data d;
    struct node *next;
} NodeT;

NodeT *front;
NodeT *rear;

int initialize() {
    front = NULL;
    rear = NULL;
}

Void enqueue(data d) {
    NodeT *newNode;

    newNode = malloc(sizeof(NodeT));
    if (newNode == NULL) {
        fprintf(stderr, "queue is full\n");
        exit(EXIT_FAILURE);
    }
    newNode->d = d;
    newNode->next = NULL;
    if(!empty()) {
        rear->next = newNode;
        rear = newNode;
    } else
    front = rear = newNode;
}
data dequeue(data d) {
    data d;
    NodeT *oldNode;

    d = front->d;
    oldNode = front;
    front = front->next;
    free(oldNode);
    return d;
}

data get_front() {
    return front->d;
}

int is_empty() {
    return front == NULL;
}
A well-designed module often keeps information secret from its clients

- information hiding

Clients of a stack module have no need to know whether it is stored in an array, in a linked list, or in some other way

In C, the major tool for enforcing information hiding is the `static` storage class
Storage Duration, Scope & Linkage

• Every variable in a C program has three properties:
  1. Storage duration
     • automatic
     • static
  2. Scope
     • block scope
     • file scope
  3. Linkage
Linkage

- Linkage of a variable determines extent to which it can be shared by different files in a program.
  - Variables with **external linkage** may be shared by several files in a program
  - Variables with **internal linkage** are restricted to a single file, but may be shared by the functions in that file
  - Variables with **no linkage** belong to a single function and can't be shared at all
Storage Duration, Scope & Linkage

```c
int i;
void g()
{
...
}

void f(void)
{

int j;
}
```

- `int i;` has **static storage duration**
- `void g()` has **file scope** and **external linkage**
- `void f(void)` has **automatic storage duration**
- `int j;` has **block scope** and **no linkage**
Static keyword

```c
static int i;
static void g()
{
...
}

void f(void)
{

static int j;

}
```

- **Static storage duration**
- **File scope**
- **Internal linkage**
- **Block scope**
- **No linkage**
Extern keyword

```c
extern int i;    // static storage duration, file scope

void f(void) {
    extern int j;    // static storage duration, block scope, external linkage
}
```
What Gets Printed?

```c
int z;
void g(void);
void f(int x)
{
    x = 2;
    z += x;
}

int main()
{
    z = 5;
    f(z);
    g();
    printf("z=%d\n", z);
    return 0;
}
```
What Gets Printed Now?

```c
static int z;
void g(void);
void f(int x)
{
    x = 2;
    z += x;
}

int main()
{
    z = 5;
    f(z);
    g();
    printf("z=%d\n", z);
    return 0;
}

extern int z;
void g(void)
{
    z *= 2;
}
```
Stack Module

stack.h

 ifndef STACK_H
 define STACK_H

 void make_empty(void);
 int is_empty(void);
 void push(int i);
 int pop(void);

 endif
Stack as array

#include "stack.h"
#define STACK_SIZE 100

static int contents[STACK_SIZE];
static int top = 0;

void make_empty(void){
    top = 0;
}
int is_empty(void){
    return top == 0;
}
static int is_full(void) {
    return top == STACK_SIZE;
}

void push(int i) {
    if (is_full()) {
        fprintf(stderr, "push: stack is full\n");
        exit(EXIT_FAILURE);
    }
    contents[top++] = i;
}

int pop(void) {
    if (is_empty()) {
        fprintf(stderr, "pop: stack is empty\n");
        exit(EXIT_FAILURE);
    }
    return contents[--top];
}
Stack as Linked List

#include "stack.h"

typedef struct node {
    int data;
    struct node *next;
} NodeT;

static NodeT *top = NULL;
```c
void make_empty(void)
{
    NodeT *next;
    while(top != NULL){
        next = top->next;
        free(top);
        top = next;
    }
}

int is_empty(void) {
    return top == NULL;
}
```
void push(int i) {
    NodeT *newNode;

    newNode = malloc(sizeof(NodeT));
    if (newNode == NULL) {
        fprintf(stderr, "push: stack is full\n");
        exit(EXIT_FAILURE);
    }
    newNode->data = i;
    newNode->next = top;
    top = newNode;
}

int pop(void) {
    NodeT *oldTop;
    int i;
    if(is_empty()) {
        fprintf(stderr, "pop: stack is empty\n");
        exit(EXIT_FAILURE);
    }
    oldTop = top;
    i = top->data;
    top = top->next;
    free(oldTop);
    return i;
}
Stack Data Type

- Can only have one instance of preceding stack modules
- Need to create a stack *type*:

```c
#include "stack.h"
int main()
{
    StackT s1, s2;
    new_stack(&s1);
    new_stack(&s2);
    push(&s1, 1);
    if (!is_empty(&s1))
        printf("%d\n", pop(&s1)); /* prints "1" */
    ...
}
```
Stack Data Type

stack.h

typedef struct node {
    int data;
    struct node *next;
} NodeT;
typedef struct {
    NodeT *top;
} StackT

void new_stack(StackT *s);
void make_empty(StackT *s);
int is_empty(const StackT *s);
void push(StackT *s, int i);
int pop(StackT *s);
Stack Type as Linked List

#include "stack.h"

void new_stack(StackT *s){
    s->top = NULL;
}

void make_empty(StackT *s){
    NodeT *next;
    while(s->top != NULL){
        next = s->top->next;
        free(s->top);
        s->top = next;
    }
}
}
void push(StackT *s, int i) {
    NodeT *newNode;

    newNode = malloc(sizeof(NodeT));
    if (newNode == NULL) {
        fprintf(stderr, "push: stack is full\n");
        exit(EXIT_FAILURE);
    }
    newNode->data = i;
    newNode->next = s->top;
    s->top = newNode;
}
```c
int is_empty(const StackT *s) {
    return s->top == NULL;
}

int pop(StackT *s) {
    NodeT *oldTop;
    int i;

    if(is_empty()) {
        fprintf(stderr, "pop: stack is empty\n");
        exit(EXIT_FAILURE);
    }
    oldTop = s->top;
    i = s->top->data;
    s->top = s->top->next;
    free(oldTop);
    return i;
}```
Stack Type

- The previous module allowed for multiple instances, but at the expense of information hiding!
- Nothing prevents a client from using a `StackT` variable as a structure:

```c
StackT s1;

s1.top = NULL;
...
```
Opaque Data Type

- Incomplete structure definition:
  - can define the type of a structure that hasn't been defined yet:

```c
typedef struct hidden *Visible;
```

- allows one to use the type `Visible` as a synonym for
  `struct hidden *`. 
Opaque Data Type

• A data type is opaque because the client cannot access its full representation
  - all the client knows is that it is represented by another data type
  - client doesn't know that that data type is

• Consider module Mod that exports a data type called Abstract to the client
  - mod.h defines a type Abstract as a pointer to a structure type called Concrete
    typedef struct Concrete *Abstract;
  - there is no definition of Concrete in this file
Client

mod.h

typedef struct
Concrete * Abstract;

Implementation

mod.c

struct Concrete {...};
Opaque Data Type

- The type **Concrete** is defined in an implementation file.

- The client can use the type provided there are no attempts to dereference values of this type.

```c
void f(Abstract p);
```
- is legal, but

```c
Abstract p;
p->x;
```
- is illegal, because the type **Abstract** represents a pointer to the **Concrete** type, and the compiler has no information about this type.
Stack ADT Module

1) the type `DataType` of the data stored in the stack is known to the implementation

2) any number of stacks can be created; all stacks must have elements of the same type, `DataType`

3) the representation of the stack and stack elements are not visible to the client.

The first version will operate on a stack of integers.
Stack ADT Header

```c
#ifndef STACK_H
#define STACK_H

typedef int DataType;
typedef struct StackCDT *StackADT;

StackADT Stack_new(void);
int Stack_empty(StackADT s);
void Stack_push(StackADT s, DataType i);
DataType Stack_pop(StackADT s);
void Stack_free(StackADT *s);

#endif
```
Application of Stack ADT

```c
#include "stack.h"

StackADT s1, s2;

s1 = Stack_new();
s2 = Stack_new();
Stack_push(s1, 1);
if (!Stack_empty(s1))
    printf("%d\n", Stack_pop(s1));
...
```
Stack ADT Implementation

#include "stack.h"

typedef struct node {
    DataType d;
    struct node *next;
} NodeT;

typedef struct StackCDT {
    int count;
    NodeT *top;
} StackCDT;
Stack ADT Implementation

StackADT Stack_new(void) {
    StackADT s;

    if((s = malloc(sizeof(StackCDT))) == NULL)
        exit(EXIT_FAILURE);
    s->count = 0;
    s->top = NULL;
    return s;
}

int Stack_empty(StackADT s) {
    return s->count == 0;
}
void Stack_push(StackADT s, DataType d) {
    NodeT *newNode;

    if ((newNode = malloc(sizeof(NodeT))) == NULL) {
        fprintf(stderr, "push: stack is full\n");
        exit(EXIT_FAILURE);
    }
    newNode->d = d;
    newNode->next = s->top;
    s->top = newNode;
    s->count++;
}
Stack ADT Implementation

```c
DataType Stack_pop(StackADT s) {
    DataType d;
    NodeT *oldNode;

    oldNode = s->top;
    s->top = oldNode->next;
    s->count--;
    d = oldNode->d;
    free(oldNode);

    return d;
}
```
Stack ADT Implementation

```c
void Stack_free(StackADT *s) {
    NodeT *p, *q;

    for (p = (*s)->top; p; p = q) {
        q = p->next;
        free(p);
    }

    free(*s);
    *s = NULL
}
```
Shallow and Deep Copy

- To push a new element `d` onto the stack:
  
  ```
  newNode->d = d;
  ```

- If `newNode->d` and `d` are pointers, this results in a *shallow* copy.
  - If client deallocates variable pointed to by `d` then
    ```
    newNode->d
    ```
    becomes a dangling reference
Shallow and Deep Copy

For a deep copy, use a callback function

```cpp
DataType copyData_Stack(const DataType v) {
  return strdup(v);
}
```

For example, for strings and doubles:
```c
char *strdup(const char *s) {
    /* return a copy of s */
    char *kopy;    /* copy of s */

    if((kopy = calloc(strlen(s) + 1, sizeof(char))) == NULL)
        return NULL;
    strcpy(kopy, s);

    return kopy;
}
```
Shallow and Deep Copy

We need another `callback` function, `freeDataStack()`.
For example, for string and doubles:

```c
void freeDataStack(DataType v) {
    free(v);
}
```

```c
void freeDataStack(DataType v) {
}
```
# Stack ADT Header with Deep Copy

typedef char* DataType;
typedef struct StackCDT *StackADT;

DataType copyData_Stack(const Datatype v);
void freeData_Stack(DataType v);
StackADT Stack_new(void);
int Stack_empty(StackADT s);
void Stack_push(StackADT s, DataType i);
DataType Stack_pop(StackADT s);
void Stack_free(StackADT s);
Stack ADT Header with Deep Copy

- Implementation of callback functions must be provided by the client

```c
DataType copyData_Stack(const Datatype v);
void freeData_Stack(DataType v);
```

- They are declared in the header file so that the implementation code can call them
Stack ADT Implementation with Deep Copy

```c
void Stack_push(StackADT s, DataType d) {
    NodeT *newNode;

    if ((newNode = malloc(sizeof(NodeT))) == NULL) {
        fprintf(stderr, "push: stack is full\n");
        exit(EXIT_FAILURE);
    }
    newNode->d = copyData_SStack(d);
    newNode->next = s->top;
    s->top = newNode;
    s->count++;
}
```
Stack ADT Implementation with Deep Copy

```c
DataType Stack_pop(StackADT s) {
    DataType d;
    NodeT *oldNode;

    oldNode = s->top;
    s->top = oldNode->next;
    s->count--;
    d = copyData_Stack(oldNode->d);
    freeData_Stack(oldNode->d);
    free(oldNode);

    return d;
}
```
void Stack_free(StackADT *s) {
    NodeT *p, q;

    for (p = (*s)->top; p; p = q) {
        q = p->next;
        freeData_Stream(p->d);
        free(p);
    }

    free(*s);
    *s = NULL
}