Program Design (3)

Program Design (II)

2022 Spring

Fu-Yin Cherng
Dept. CSIE, National Chung Cheng University

Quick Recap of Linked List

```c
void push(int i){
  struct node *new_node = malloc(sizeof(struct node));
  if (new_node == NULL){
    terminate("...");
  }
  …//add new node at first
}
```

```c
1.  top = new_node;
```

```c
2. new_node->next = top;
```

```c
3. new_node->data = i;
```

213 <- this is also correct!

321

231

Outline

- A Stack Abstract Data Type
  - How to encapsulate the abstract data type, Stack ADT, using incomplete types?
  - Implementing Stack ADT using a Fixed Length Array -> Dynamic Array -> Linked List

A Stack Abstract Data Type

- The following stack ADT will illustrate how abstract data types can be encapsulated using incomplete types.
- The stack will be implemented in **three** different ways.
  - Fixed Length Array
  - Dynamic Array
  - Linked List

Defining the Interface for the Stack ADT

- `stackADT.h` **defines** the stack ADT **type** and gives **prototypes** for the **functions** that represent stack operations.
- The `Stack` type will be a **pointer** to a `stack_type` **structure** (an incomplete type).
- This structure is an **incomplete** type that will be completed in the file that implements the stack
  - The members of this structure will depend on how the stack is implemented.

```
#ifndef STACKADT_H
#define STACKADT_H

#include <stdbool.h>

typedef struct stack_type  *Stack;

Stack create(void);
void destroy(Stack s);
void make_empty(Stack s);
bool is_empty(Stack s);
bool is_full(Stack s);
void push(Stack s, int i);
int pop(Stack s);

#endif
```

Defining the Interface for the Stack ADT

- Clients that include `stackADT.h` will be able to declare **pointer variables** of type `Stack`, each of which is capable of **pointing to** a `stack_type` structure.
- However, clients can’t access the members of the `stack_type` structure, since that structure will be defined in a separate file.

```c
#ifndef STACKADT_H
#define STACKADT_H

#include <stdbool.h>

typedef struct stack_type *Stack;

Stack create(void);
void destroy(Stack s);
void make_empty(Stack s);
bool is_empty(Stack s);
bool is_full(Stack s);
void push(Stack s, int i);
int pop(Stack s);

#endif
```

Defining the Interface for the Stack ADT

- Clients can then call the **functions** declared in `stackADT.h` to perform operations on stack variables.
- A module generally doesn't need `create` and `destroy` functions, but an ADT does.
  - `create` dynamically allocates memory for a stack and initializes the stack to its "empty" state.
  - `destroy` releases the stack's dynamically allocated memory.

```c
#ifndef STACKADT_H
#define STACKADT_H

#include <stdbool.h>

typedef struct stack_type *Stack;

Stack create(void);
void destroy(Stack s);
void make_empty(Stack s);
bool is_empty(Stack s);
bool is_full(Stack s);
void push(Stack s, int i);
int pop(Stack s);

#endif
```

Defining the Interface for the Stack ADT

- Let’s see how a client can use Stack ADT
- The name of the client is: `stackclient.c`, which can be used to test the stack ADT.
- It declares **two stacks** and performs a variety of operations on them.
- What are the correct content for gap (1) to (4) if the output message is "Popped 2 from s1"?

```
#include <stdio.h>
#include "stackADT.h"
int main(void){
  Stack s1, s2;
  int n;

  s1 = __(1)__();
  s2 = __(1)__();

  __(2)__(s1, 1);
  __(2)__(s1, 2);

  n = __(3)__(s1);
  printf("Popped %d from s1\n", n);

  ___(4)___(s1);
  ___(4)___(s2);
  return 0;
}
```

Implementing the Stack ADT Using a Fixed-Length Array

- There are several ways to implement the stack ADT.
- The **simplest** is to have the `stack_type` structure contain a **fixed-length** array:

```
…

typedef struct stack_type *Stack;

…
```

stackADT.h

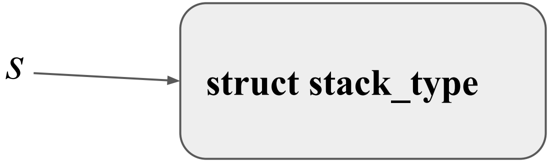
```
#define STACK_SIZE 100
struct stack_type {
        int contents[STACK_SIZE];
        int top;
};
```

```
typedef struct stack_type *Stack;
```

```
#include <stdio.h>
#include <stdlib.h>
#include "stackADT.h"

#define STACK_SIZE 100
struct stack_type {
  int contents[STACK_SIZE];
  int top;
};
static void terminate(const char *message){...}
Stack create(void){
  Stack s = malloc(sizeof(struct stack_type));
  if (s == NULL){
    terminate("Error in create: stack could not be created.");
  }
  s->top = 0;
  return s;
}

void destroy(Stack s){
  free(s);
}
```



```
void make_empty(Stack s){
  s->top = 0;
}
bool is_empty(Stack s){
  return s->top == 0;
}
bool is_full(Stack s){
  return s->top == STACK_SIZE;
}
void push(Stack s, int i){
  if (is_full(s)){
    terminate("Error in push: stack is full.");}
  s->contents[s->top++] = i;
}
int pop(Stack s){
  if (is_empty(s)){
    terminate("Error in pop: stack is empty.");}
  return s->contents[--s->top];
}
```

Changing the Item Type in the Stack ADT

- `stackADT.c` requires that stack items be **integers**, which is too restrictive.
- To make the stack ADT easier to modify for different item types,
- Let's add a type definition to the `stackADT.h` header.
- It will **define** a **type** named `Item`, representing the type of data to be stored on the stack.

```
typedef int Item;
typedef struct stack_type *Stack;
```

`stackADT.h`

```c
#ifndef STACKADT_H
#define STACKADT_H

#include <stdbool.h>
typedef int Item;

typedef struct stack_type *Stack;

Stack create(void);
void destroy(Stack s);
void make_empty(Stack s);
bool is_empty(Stack s);
bool is_full(Stack s);
void push(Stack s, Item i);
Item pop(Stack s);

#endif
```

stackADT.h

Changing the Item Type in the Stack ADT

- The `stackADT.c` file will need to be modified, but the changes are minimal.
- First, we need to update `stack_type` structure
- The item type can be changed by modifying the definition of `Item` in `stackADT.h`.

```
struct stack_type {
     Item contents[STACK_SIZE];
     int top;
};
```

`stackADT.h`

Changing the Item Type in the Stack ADT

- Next, …
- The second parameter of `push` will now have type `Item`.
- `pop` now returns a value of type `Item`.

```
void push(Stack s, Item i){
  if (is_full(s)){
    terminate("Error in push: stack is full.");}
  s->contents[s->top++] = i;
}
Item pop(Stack s){
  if (is_empty(s)){
    terminate("Error in pop: stack is empty.");}
  return s->contents[--s->top];
}
```

`stackADT.h`

Let’s Take a Break.

Implementing the Stack ADT Using a Dynamic Array

- Another problem with the stack ADT: each stack has a fixed maximum size.
- There's no way to have stacks with different capacities or to set the stack size as the program is running.
- Possible solutions to this problem:
  - Store stack items in a dynamically allocated array.
  - Implement the stack as a linked list.
- Let's see how to implement the first solution first.

Implementing the Stack ADT Using a Dynamic Array

- To store stack items in a dynamically allocated array, we need to modifying the `stack_type` structure.
- The `contents` member becomes a *pointer* to the array in which the items are stored
- The `size` member stores the stack's maximum size.

```
#define STACK_SIZE 100
struct stack_type {
      int contents[STACK_SIZE];
      int top;
};
```

```
struct stack_type {
      Item *contents;
      int top;
      int size;
};
```

Implementing the Stack ADT Using a Dynamic Array

- The **`create` function** will now have a **parameter** that specifies the desired **maximum** stack **size**
- When `create` is called, it will first allocate **space for the** `stack_type` **structure**

```
Stack create(int size)
{
  Stack s = malloc(sizeof(struct stack_type));
  if (s == NULL) {terminate("...");}





  return s;
}
```

Implementing the Stack ADT Using a Dynamic Array

- **Then, the second** `malloc` **is called for allocate** an **array** of **length** `size`.
- And the `contents` member of the structure will point to this array.

```
Stack create(int size)
{
  Stack s = malloc(sizeof(struct stack_type));
  if (s == NULL){terminate("...");}
  s->contents = malloc(size * sizeof(Item));
  if (s->contents == NULL) {
    free(s);
    terminate("...");
  }
  s->top = 0;
  s->size = size;
  return s;
}
```

Implementing the Stack ADT Using a Dynamic Array

Given the above `create` function, how `destroy` functon should look like to free all the allocated memory?

```
void destroy(Stack s)
{
    ____(1)______;
    ____(2)____;
}
```

Implementing the Stack ADT Using a Dynamic Array

- The rest modification is in `is_full` function
- We can use size member in **`struct stack_type`** to check if the sack is full

```
bool is_full(Stack s){
  return s->top == s->size;
}
```

Implementing the Stack ADT Using a Dynamic Array

- When a client is trying to use this version of Stack ADT
- The calls of `create` will need to be changed, since `create` now requires an argument.

```
#include <stdio.h>
#include "stackADT.h"
int main(void){
  Stack s1, s2;
  int n;
  s1 = create(100);
  s2 = create(200);
}
```

Implementing the Stack ADT Using a Dynamic Array

- Implementing the stack ADT using a dynamically allocated array provides more flexibility than using a fixed-size array.
- **However**, the client is still **required** to **specify** a **maximum size** for a stack at the time it's created.
- With a **linked-list** implementation, there **won't** be any limit on the size of a stack.

Implementing the Stack ADT Using a Dynamic Array

- First, we will need to define a `struct node` type
- The linked list will consist of nodes, represented by the following structure

```
struct node {
      Item data;
      struct node *next;
};
```

Implementing the Stack ADT Using a Dynamic Array

- The `stack_type` structure will contain a pointer to the first node in the list

```
struct stack_type {
      struct node *top;
};
```

Implementing the Stack ADT Using a Dynamic Array

- The `stack_type` structure seems useless, since `Stack` could be defined to be `struct node *`

```
// why not? typedef struct node *Stack;
typedef struct stack_type *Stack;
```

Implementing the Stack ADT Using a Dynamic Array

- However, `stack_type` is **needed** so that the **interface** to the stack **remains unchanged** (`stackADT.h`).
- Moreover, having the `stack_type` structure will make it **easier to change** the implementation in the **future**.

```
struct node {
      Item data;
      struct node *next;
};
```

```
struct stack_type {
      struct node *top;
};
```

```c
#ifndef STACKADT_H
#define STACKADT_H

#include <stdbool.h>
typedef int Item;

typedef struct stack_type *Stack;

Stack create(void);
void destroy(Stack s);
void make_empty(Stack s);
bool is_empty(Stack s);
bool is_full(Stack s);
void push(Stack s, Item i);
Item pop(Stack s);

#endif
```

stackADT.h

```c
#include <stdio.h>
#include <stdlib.h>
#include "stackADT.h"

struct node {
  Item data;
  struct node *next;
};

struct stack_type {
  struct node *top;
};
…
```

stackADT3.h (linked list)

```c
Stack create(void){
  Stack s = malloc(sizeof(struct stack_type));
  if (s == NULL){terminate("...")};
  s->top = NULL;
  return s;
}


void destroy(Stack s){
  make_empty(s);
  free(s);
}

void make_empty(Stack s){
  while (!is_empty(s))
    pop(s);
}
```

call `make_empty` before `free()`

```c
struct node {
  Item data;
  struct node *next;
};

struct stack_type {
  struct node *top;
};
```

stackADT3.h (linked list)

slido

Why destroy function calls make_empty before free()?

ⓘ Start presenting to display the poll results on this slide.

```c
struct node {
  Item data;
  struct node *next;
};

struct stack_type {
  struct node *top;
};
```

```c
void push(Stack s, Item i){
  struct node *new_node = malloc(sizeof(struct
   node));
  if (new_node == NULL)
    terminate("...");

  new_node->data = i;
  new_node->next = s->top;
  s->top = new_node;
}
```

stackADT3.h (linked list)

```
Item pop(Stack s){
  struct node *old_top;
  Item i;

  if (is_empty(s))
    terminate("...");

  old_top = s->top;
  i = old_top->data;
  s->top = old_top->next;
  free(old_top);
  return i;
}
```

```
struct node {
  Item data;
  struct node *next;
};

struct stack_type {
  struct node *top;
};
```

stackADT3.h (linked list)

Summary

- A Stack Abstract Data Type
- How to encapsulate the abstract data type, Stack ADT, using incomplete types?
- Implementing Stack ADT using a
  - Fixed Length Array
  - Dynamic Array
  - Linked List
- **`typedef int Item;`** -> specify the type of value stored in the stack