Program Design (2)

Program Design (II)

2022 Spring

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

Quick Recap

- Program Design
  - Why?
- Module
  - Abstraction
  - Reusability
  - Maintainability
  - Cohesion and Couplin
  - Types of Modules
- Information Hiding
  - Security and Flexibility

stack.h

```
#include <stdbool.h>

void make_empty(void);
bool is_empty(void);
bool is_full(void);
void push(int i);
int pop(void);
```

calc.c

```
#include "stack.h"

int main(void)
{
  make_empty();
  …
}
```

stack.c

```
#include "stack.h"

int contents[100];
int top = 0;

void make_empty(void)
{ … }

bool is_empty(void)
{ … }

bool is_full(void)
{ … }

void push(int i)
{ … }

int pop(void)
{ … }
```

Quick Recap

- Please discuss with the other classmate sitting nearby about
  - What is Module to you?
  - Think one or two examples of Modules with your partner

Outline

- Example of information hiding: A Stack Module
- Abstract Data Types
- Encapsulation

A Stack Module

- To see the benefits of information hiding, let's look at **two** implementations of a stack module
  - array
  - linked list
- `stack.h` is the module's **header** file.

```
#ifndef STACK_H
#define STACK_H

#include <stdbool.h>

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

#endif
```

A Stack Module

- `stack1.c` uses an array to implement the stack.
- Use fixed-length array to implement the stack
- The variables that make up the stack (`contents` and `top`) are both declared `static`
- Since there is no need for the rest of the program (other files) to access them directly

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

#define STACK_SIZE 100

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

…
```

A Stack Module

- The `terminate` function is also declared `static`
- This function is not in the module's interface
- The `terminate` function is only used for this module
- Therefore, we will call this kind of function **private** functions

```
…
static void terminate(const char
*message){
  printf("%s\n", message);
  exit(EXIT_FAILURE);
}

void make_empty(void){
  top = 0;
}

bool is_empty(void){
  return top == 0;
}
```

A Stack Module

- The other functions of the module interface can be accessed by other source files/clients
- Therefore, we will call this kind of function **public** functions

```
…
bool is_full(void){
  return top == STACK_SIZE;
}

void push(int i){
  if (is_full()){
    terminate("Error in push:
   stack is full.");}
  contents[top++] = i;
}

int pop(void){
  if (is_empty()){
    terminate("Error…");}
  return contents[--top];
}
```

A Stack Module

- We can use Macros to help us indicate whether a function or variable is “public” (accessible elsewhere in the program) or “private” (limited to a single file)
- The word `static` has more than one use in C, os using `PRIVATE` makes it clear that we’re using it to enforce information hiding.

```
#define PUBLIC  /* empty */
#define PRIVATE static
```

A Stack Module

- The stack implementation redone using `PUBLIC` and `PRIVATE`

```
PRIVATE int contents[STACK_SIZE];
PRIVATE int top = 0;
PRIVATE void terminate(const char *message) { … }
PUBLIC void make_empty(void) { … }
PUBLIC bool is_empty(void) { … }
PUBLIC bool is_full(void) { … }
PUBLIC void push(int i) { … }
PUBLIC int pop(void) { … }
```

A Stack Module

- Now, let's **switch** to a **linked-list** implementation of the stack module.
  - `stack2.c`
- Although we can the way of implementing, we can still use the same interface (`stack.h`)

stack2.c

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

struct node {
  int data;
  ___(1)___;
};

static struct node *top = NULL;
…
```

stack1.c

```
#include <stdio.h>
#include <stdlib.h>
#include "stack.h"
#define STACK_SIZE 100

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

slido

What is the correct content for gap (1) if struct node will be used as the node to form linked list?

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

```
static void terminate(const
char *message){
  printf("%s\n", message);
  exit(EXIT_FAILURE);
}

void make_empty(void){
  while (!is_empty()){
    pop();
  }
}

bool is_empty(void){
  return top == NULL;
}
```

stack2.c

```
static void terminate(const char
*message){
  printf("%s\n", message);
  exit(EXIT_FAILURE);
}

void make_empty(void){
  top = 0;
}

bool is_empty(void){
  return top == 0;
}
```

stack1.c

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

int pop(void){
  struct node *old_top;
  int i;
  …//remove the first node
}
```

stack2.c

```
bool is_full(void){
  return top == STACK_SIZE;
}

void push(int i){
  if (is_full()){
    terminate("...");}
  contents[top++] = i;
}

int pop(void){
  if (is_empty()){
    terminate("...");}
  return contents[--top];
}
```

stack1.c

A Stack Module

- Note that the `is_full` function in `stack2.c` returns `false` everytime, because linked list has no limit on its size!
- So the stack will never be full (unless running out of computer memory)

```
bool is_full(void){
  return false;
}
```

A Stack Module

- Thanks to information hiding, it doesn't matter whether we use `stack1.c` or `stack2.c` to implement the stack module.
- Both versions match the module's interface, so we can switch from one to the other without having to make changes elsewhere in the program.

Let’s Take a Break.

Quick Recap of Linked List

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

1. `top = new_node;`

2. `new_node->next = top;`

3. `new_node->data = i;`

slido

What is the correct order of statements for push() function?

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

Quick Recap of Linked List

```
int pop(void){
  struct node *old_top;
  int i;
  …//remove the first node
}
```

```
1.  top = top->next;
```

```
2. old_top = top;
   i = top-<data;
```

```
3. free(old_top);
   return i;
```

slido

What is the correct order of statements for pop() function?

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

Abstract Data Types

- A module that serves as an abstract object (like the previous stack moudule) has a serious disadvantage
- There's no way to have multiple instances of the object.
  - we can only have one stack no matter we use `stack1.c` or `stack2.c`
  - since the **declaration** of stack is written in the source files

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

struct node {
  int data;
  ___(1)___;
};

static struct node *top = NULL;
```

```
#include <stdio.h>
#include <stdlib.h>
#include "stack.h"
#define STACK_SIZE 100

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

Abstract Data Types

- To accomplish this, we'll need to create a new *type*.
- For example, we can define a `Stack` abstract data type (ADT) so that we can use `Stack` to create any number of stacks in multiple files.

Abstract Data Types

- Here is an example of a program fragment that uses two stacks
- We don't really need to know what `s1` and `s2` are (structures or pointers?)
- To clients, `s1` and `s2` are *abstractions* that can be handled by **certain** operations (`make_empty`, `is_empty`, `is_full`, `push`, and `pop`).

```
Stack s1, s2;

make_empty(&s1);
make_empty(&s2);
push(&s1, 1);
push(&s2, 2);
```

Abstract Data Types

- To achieve the prior example by defining a `Stack` abstract data type (ADT),
- we need to modify the `stack.h` header so that it provides a `Stack` type, where `Stack` is a structure
- Doing so will require adding a `Stack` (or `Stack *`) parameter to each function.
- Here is the modified header file (Changes to `stack.h` are shown in **bold**)

```
#define STACK_SIZE 100

typedef struct {
     int contents[STACK_SIZE];
     int top;
} Stack;

void make_empty(Stack *s);
bool is_empty(const Stack *s);
bool is_full(const Stack *s);
void push(Stack *s, int i);
int pop(Stack *s);
```

```
#ifndef STACK_H
#define STACK_H

#include <stdbool.h>

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

#endif
```

original header file

```
#define STACK_SIZE 100

typedef struct {
     int contents[STACK_SIZE];
     int top;
} Stack;

void make_empty(Stack *s);
bool is_empty(const Stack *s);
bool is_full(const Stack *s);
void push(Stack *s, int i);
int pop(Stack *s);
```

Abstract Data Types

- The stack parameters to `make_empty`, `push`, and `pop` need to be pointers, since these functions modify the stack.
- Passing these functions a `Stack` *pointer* instead of a `Stack` *value* is done for **efficiency**, since the latter would result in a structure being copied.

```
#define STACK_SIZE 100

typedef struct {
    int contents[STACK_SIZE];
    int top;
} Stack;

void make_empty(Stack *s);
bool is_empty(const Stack *s);
bool is_full(const Stack *s);
void push(Stack *s, int i);
int pop(Stack *s);
```

Abstract Data Types

- However, this version of header file is **not good enough!**
- Unfortunately, `Stack` isn't an *abstract* data type, since `stack.h` reveals what the `Stack` type really is.
- Nothing prevents clients/users from using a Stack variable as a structure.
- For example, we can write the following statements in other files to directly access the member of `Stack s1`

```
Stack s1;
s1.top = 0;
s1.contents[top++] = 1;
```

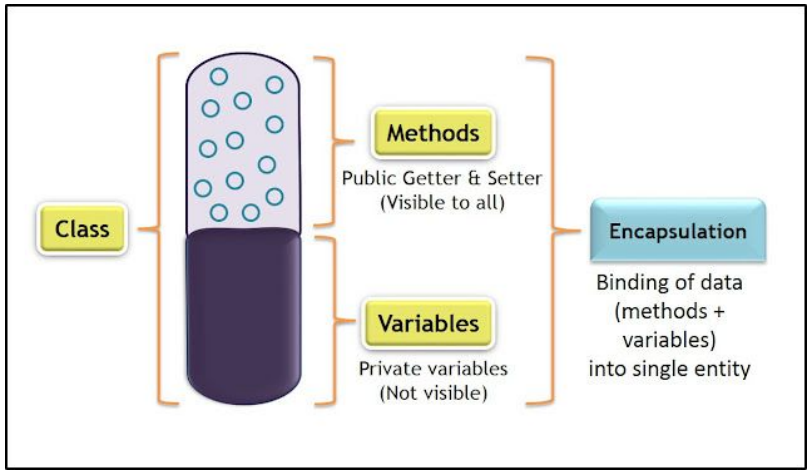
Abstract Data Types

- Why this is not good?
- Because clients/users will not always operate Stack ADT correctly…
- For example, providing access to the `top` and `contents` members allows clients to **corrupt** the stack.
  - forgot to initialize `top` to zero

```
Stack s1;
s1.top = 0;
s1.contents[top++] = 1;
```

Encapsulation

- What we need is a way to prevent clients from knowing how the `Stack` type is represented.
- C has only limited support for ***encapsulating*** types in this way.
- Newer C-based languages, including C++, Java, and C#, are better equipped for this purpose.



https://corejavasea.blogspot.com/2020/05/encapsulation.html

Incomplete Types

- The only tool that C gives us for encapsulation is the ***incomplete type.***
- Incomplete types are “types that describe objects but lack information needed to determine their sizes.”
- The following example tells the complier that t is a structure tag but doesn’t describe the members of the structure.
- So, the compiler doesn’t have enough information to determine the size of such a structure

```
struct t;  /* incomplete declaration of t */
```

Incomplete Types

- The intent is that an incomplete type will be completed elsewhere in the program.
- As long as a type is incomplete, its uses are limited.
- For example, since the compiler doesn't know the size of an incomplete type, an incomplete type can't be used to declare a variable like `s` below

```
struct t;  /* incomplete declaration of t */
struct t s;   /*** WRONG ***/
```

Incomplete Types

- However, it's legal to define a pointer type that references an incomplete type!
- The type definition states that a variable of type `T` is a **pointer to a structure** with **tag** `t` (type: `struct t`)

```
struct t;  /* incomplete declaration of t */
typedef struct t *T;
```

Incomplete Types

- We can now declare variables `s` of type `T`, pass them as **arguments** to functions, and perform other operations that are legal for pointers.
- Since the size of a pointer doesn't depend on what it points to.
- What we can't do is using `->` operator to s, because complier knows nothing about the members of a t structure

```
struct t;  /* incomplete declaration of t */
typedef struct t *T;
T s;
s-> ?; //WRONG!
```

Summary

- Example of information hiding: A Stack Module
  - array
  - linked list
- Abstract Data Types
  - How to improve the header file to set up a better stack ADT?
- Encapsulation
  - Incomplete Types