Lecture 18: Program Design

Sarah Nadi
nadi@ualberta.ca
Department of Computing Science
University of Alberta

CMPUT 201 - Practical Programming Methodology

[With material/slides from Guohui Lin, Davood Rafei, and Michael Buro.
Some content taken from K.N. King's slides based on course text book]

UNIVERSITY OF ALBERTA

Agenda

- Modules
- Information Hiding
- Encapsulation
- Abstract Data Types

Readings

- Textbook Chapter 19

Overview

- We will learn how to design programs that are:
  - modular
  - easily maintained
  - reusable

Modules

Modules

- A *module* is a collection of services, some of which are made available to other parts of the program (the *clients*)

Modules

- A *module* is a collection of services, some of which are made available to other parts of the program (the *clients*)

- Each module has an *interface* that describes its available services

Modules

- A *module* is a collection of services, some of which are made available to other parts of the program (the *clients*)
- Each module has an *interface* that describes its available services
- The details of the module, including the source code for the services themselves, are stored in the module's *implementation*

Modules

- A *module* is a collection of services, some of which are made available to other parts of the program (the *clients*)

- Each module has an *interface* that describes its available services

- The details of the module, including the source code for the services themselves, are stored in the module's *implementation*

- Dividing your program into modules allows for better abstraction, reusability, and maintainability

Modules in C

Modules in C

- The module “services” are typically functions, but may also be values

Modules in C

- The module “services” are typically functions, but may also be values

- The interface of a module is a header file containing the prototypes for the functions that will be available to clients (e.g., think of the C file with the main function as one of your clients)

Modules in C

- The module "services" are typically functions, but may also be values
- The interface of a module is a header file containing the prototypes for the functions that will be available to clients (e.g., think of the C file with the main function as one of your clients)
- The implementation of a module is a source file that contains definitions of the module's functions

Recall Our Calculator Example

```c
#include <stdbool.h>

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

stack.h

interface of the stack module

```c
#include "stack.h"

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

calc.c

client of the stack module

```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)
{ … }
```

stack.c

implementation of the stack module

The C Library

- The C library is itself a collection of modules
- Each header in the library serves as the interface to a module
    - <stdio.h> is the interface to a module of I/O functions
    - <string.h> is the interface to a module of string-handling functions

Abstraction

Abstraction

- *Abstraction* is a design concept where you design a module in a way such that your clients know only what is needed to use the module

Abstraction

- *Abstraction* is a design concept where you design a module in a way such that your clients know only what is needed to use the module
- For example, as a client of stdio.h, I need to know that it gives me a print functionality, but I don’t care how it actually implements it

Abstraction

- *Abstraction* is a design concept where you design a module in a way such that your clients know only what is needed to use the module

- For example, as a client of stdio.h, I need to know that it gives me a print functionality, but I don't care how it actually implements it

- If the C library developers decide to change how the printf function is implemented, that does not affect my code

Abstraction

- *Abstraction* is a design concept where you design a module in a way such that your clients know only what is needed to use the module

- For example, as a client of stdio.h, I need to know that it gives me a print functionality, but I don’t care how it actually implements it

- If the C library developers decide to change how the printf function is implemented, that does not affect my code

- Thanks to abstraction, it is not necessary to understand how the entire program works in order to change one part of it. This makes it easier for several members of a team to work on the same program

Reusability

- Separating functionality into modules allows for reusability
- I can reuse the stack module in a completely different program that balances parentheses for example
- Always design modules with reusability in mind

Maintainability

Maintainability

- If your program is divided into modules, it is often easier to locate and fix bugs since they can be narrowed down to a single module's implementation

Maintainability

- If your program is divided into modules, it is often easier to locate and fix bugs since they can be narrowed down to a single module's implementation

- Rebuilding the program would require only recompiling that changed module

Maintainability

- If your program is divided into modules, it is often easier to locate and fix bugs since they can be narrowed down to a single module's implementation
- Rebuilding the program would require only recompiling that changed module
- If you decide that you want to completely change a module, you can simply replace it without affecting the rest of your program (as long as the new module adheres to the same interface)

Design Principles for Modules

Design Principles for Modules

- Modules should have two properties:

Design Principles for Modules

- Modules should have two properties:
  - *High cohesion*: the “services” of each module should be closely related to one another. High cohesion makes it easier to understand a give module.

Design Principles for Modules

- Modules should have two properties:
  - *High cohesion*: the “services” of each module should be closely related to one another. High cohesion makes it easier to understand a give module.
  - *Low coupling*: modules should be as independent of each other as possible. Low coupling makes it easier to modify the program and reuse modules.

Types of Modules

Types of Modules

- *Data Pool*
  - A collection of related variables and/or constants (e.g., <float.h> and <limits.h>)

Types of Modules

- *Data Pool*
  - A collection of related variables and/or constants (e.g., <float.h> and <limits.h>)
- *Library*
  - A library is a collection of related functions (e.g., <string.h>)

Types of Modules

- *Data Pool*
  - A collection of related variables and/or constants (e.g., <float.h> and <limits.h>)
- *Library*
  - A library is a collection of related functions (e.g., <string.h>)
- *Abstract object*
  - collection of functions that operate on a hidden data structure (e.g., the stack module we saw on Slide 6)

Types of Modules

- *Data Pool*
  - A collection of related variables and/or constants (e.g., <float.h> and <limits.h>)
- *Library*
  - A library is a collection of related functions (e.g., <string.h>)
- *Abstract object*
  - collection of functions that operate on a hidden data structure (e.g., the stack module we saw on Slide 6)
- *Abstract Data Type (ADT)*
  - a type whose representation is hidden. Client modules can use the type to declare variables, but have no knowledge of the structure of those variables. Clients use the functions in the ADT's interface to perform any operations on variables of that type.

Information Hiding

Information Hiding

- *Information hiding* describes the design principle of deliberately concealing information from the clients of a module

Information Hiding

- *Information hiding* describes the design principle of deliberately concealing information from the clients of a module

- Two advantages of information hiding:

    - *security*: if clients don't know how a stack is stored, they won't be able to corrupt it. They can only use the functions provided by the module

    - *flexibility*: you can change internal details without affecting any other parts of the program

Information Hiding

- *Information hiding* describes the design principle of deliberately concealing information from the clients of a module

- Two advantages of information hiding:

  - *security*: if clients don't know how a stack is stored, they won't be able to corrupt it. They can only use the functions provided by the module

  - *flexibility*: you can change internal details without affecting any other parts of the program

- In C, the major tool for enforcing information hiding is the `static` keyword. A static variable or function can only be used in the same file.

Example: Interface of Stack Module (as an Abstract Object)

```
#ifndef STACK_H
#define STACK_H

#include <stdbool.h>   /* C99 only */

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

#endif
```

stack.h

Stack Module - Implementation 1 using Arrays

stack1.c

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

#define STACK_SIZE 100

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

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

void make_empty(void)
{
  top = 0;
}
```

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

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 in pop: stack is empty.");
  return contents[--top];
}
```

Information Hiding in stack1.c

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

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) { … }
```

Stack Module - Implementation 2 using Linked Lists

Alternative Implementation of the Stack Module

- We will not change anything in the interface of the stack module. Instead, we will simply change its implementation

stack2.c

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

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

static struct node *top = NULL;

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;
}
```

```c
int pop(void)
{
  struct node *old_top;
  int i;

  if (is_empty())
    terminate("Error in pop: stack is empty.");

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

bool is_full(void)
{
  return false;
}

void push(int i)
{
  struct node *new_node = malloc(sizeof(struct node));
  if (new_node == NULL)
    terminate("Error in push: stack is full.");

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

Comments on the Stack Module

- Thanks to information hiding, it doesn't matter whether we use stack1.c or stack2.c
- A client of the stack module sees the same interface and gets the same behavior
- This allows us to switch implementations at any point without affecting any other parts of the program that rely on the stack module

Abstract Data Types

- One problem with abstract objects (similar to the stack module we just saw) is that there's no way to have multiple instances of the object. In the previous example, all functions operated on the same array or linked list
- To be able to have multiple instances, we need to create a new type
- For example, a `Stack` type can be used to create any number of stacks

Example of Intended Behavior

```
Stack s1, s2;

make_empty(&s1);
make_empty(&s2);
push(&s1, 1);
push(&s2, 2);
if (!is_empty(&s1))
  printf("%d\n", pop(&s1));  /* prints "1" */
```

Changing the Module's Interface so it's an ADT

```
#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);
```

Information Hiding & Encapsulation

Information Hiding & Encapsulation

- The Stack on the previous slide isn't an **abstract** data type, since stack.h reveals what the Stack type really is.

Information Hiding & Encapsulation

- The Stack on the previous slide isn’t an **abstract** data type, since stack.h reveals what the Stack type really is.

- Nothing prevents clients from using a Stack variable and directly manipulating it without using the provided functions:

```
Stack s1;

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

Information Hiding & Encapsulation

- The Stack on the previous slide isn't an **abstract** data type, since stack.h reveals what the Stack type really is.

- Nothing prevents clients from using a Stack variable and directly manipulating it without using the provided functions:

```
Stack s1;

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

Information Hiding & Encapsulation

- The Stack on the previous slide isn't an **abstract** data type, since stack.h reveals what the Stack type really is.

- Nothing prevents clients from using a Stack variable and directly manipulating it without using the provided functions:

```
Stack s1;

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

- Beyond potential of corrupting the stack, if we change the way stacks are stored, we might affect client code

Information Hiding & Encapsulation

- The Stack on the previous slide isn't an **abstract** data type, since stack.h reveals what the Stack type really is.

- Nothing prevents clients from using a Stack variable and directly manipulating it without using the provided functions:

```
Stack s1;

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

- Beyond potential of corrupting the stack, if we change the way stacks are stored, we might affect client code

- We need a way to prevent clients from knowing how the Stack type is represented —> encapsulation and information hiding

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 intent is that an incomplete type will be completed elsewhere in the program

```
struct t; //incomplete type
```

Incomplete Types *Cont'd*

- An incomplete type cannot be used to declare a variable:

```
struct t s; //WRONG
```

- However, it is legal to define a pointer type that references an incomplete type

```
typedef struct t *T; //OK
```

- The above declares a new type name T that is a pointer to a struct called t

- We can now declare variables of type `T`, pass them as arguments to functions, and perform any operations that are legal for pointers

Interface for the Stack ADT

```
#ifndef STACKADT_H
#define STACKADT_H

#include <stdbool.h>   /* C99 only */

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
```

stackADT.h

Interface for the Stack ADT

```
#ifndef STACKADT_H
#define STACKADT_H

#include <stdbool.h>   /* C99 only */

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
```

stackADT.h

A client will know that there is Stack is a pointer to a struct called stack_type but it has no idea what is in stack_type and as a result, cannot directly use any of the members of stack_type

Interface for the Stack ADT

```
#ifndef STACKADT_H
#define STACKADT_H

#include <stdbool.h>   /* C99 only */

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
```

stackADT.h

```
#ifndef STACKADT_H
#define STACKADT_H

#include <stdbool.h>   /* C99 only */

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
```

stackADT2.h

Interface for the Stack ADT

```c
#ifndef STACKADT_H
#define STACKADT_H

#include <stdbool.h>   /* C99 only */

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
```

stackADT.h

```c
#ifndef STACKADT_H
#define STACKADT_H

#include <stdbool.h>   /* C99 only */

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
```

stackADT2.h

For added flexibility, we can even declare the type of items in the stack as a typedef such that we can easily change it later

Implementation Options for StackADT using Array

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

#define STACK_SIZE 100

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


int pop(Stack s)
{
  if (is_empty(s))
    terminate("Error in pop: stack is empty.");
  return s->contents[--s->top];
}

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;
}
...
```

Implementation Options for StackADT using Array

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

#define STACK_SIZE 100

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

 Item
int pop(Stack s)
{
  if (is_empty(s))
    terminate("Error in pop: stack is empty.");
  return s->contents[--s->top];
}

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;
}
...
```

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

Implementing a Stack ADT using a Dynamic Array

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

```
void destroy(Stack s)
{
  free(s->contents);
  free(s);
}
```

```
Stack create(int size)
{
  Stack s = malloc(sizeof(struct stack_type));
  if (s == NULL)
    terminate("Error in create: stack could not be created.");
  s->contents = malloc(size * sizeof(Item));
  if (s->contents == NULL) {
    free(s);
    terminate("Error in create: stack could not be created.");
  }
  s->top = 0;
  s->size = size;
  return s;
}
```

Implementation of StackADT Using Linked Lists

Implementation of StackADT Using Linked Lists

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

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

Implementation of StackADT Using Linked Lists

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

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

```
Stack create(void)
{
  Stack s = malloc(sizeof(struct stack_type));
  if (s == NULL)
    terminate("Error in create: stack could not be created.");
  s->top = NULL;
  return s;
}
```

Implementation of StackADT Using Linked Lists

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

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

```
Stack create(void)
{
  Stack s = malloc(sizeof(struct stack_type));
  if (s == NULL)
    terminate("Error in create: stack could not be created.");
  s->top = NULL;
  return s;
}
```

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

  if (is_empty(s))
    terminate("Error in pop: stack is empty.");

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

Implementation of StackADT Using Linked Lists

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

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

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

  if (is_empty(s))
    terminate("Error in pop: stack is empty.");

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

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

```
Stack create(void)
{
  Stack s = malloc(sizeof(struct stack_type));
  if (s == NULL)
    terminate("Error in create: stack could not be created.");
  s->top = NULL;
  return s;
}
```

Naming Conventions in ADT

- Try to use easy-to-understand names, but also names that are unique to avoid name clashes: `create` can be used by multiple ADTs but `create_stack` is more unique

Error Handling

- You need to think of whether you want to terminate the program every time an error occurs. For example, you can change the push function to return a boolean to indicate whether it succeeded or not, instead of terminating the program.

- Instead of having lots of if conditions in your implementation and then exiting the program, an alternative is to use the `assert` function

`assert`

```
void assert(int expression);
```

- If the expression passed to it is “false” (i.e., 0), `assert` reports an error to stderr and terminates the program To use assert, include <assert.h>
- Example:

```
int i = 5;
assert(i > 0); //passes
assert(i < 0); //fails
char *str = “Hello!”;
assert (strcmp(str, “Hi”) == 0); //fails
assert (strlen(str) > 0)); //passes
```

demo: assert.c