CS 2213
Advanced Programming
Ch 8 – Abstract Data Types

Turgay Korkmaz

Office: SB 4.01.13
Phone: (210) 458-7346
Fax: (210) 458-4437
e-mail: korkmaz@cs.utsa.edu
web: www.cs.utsa.edu/~korkmaz

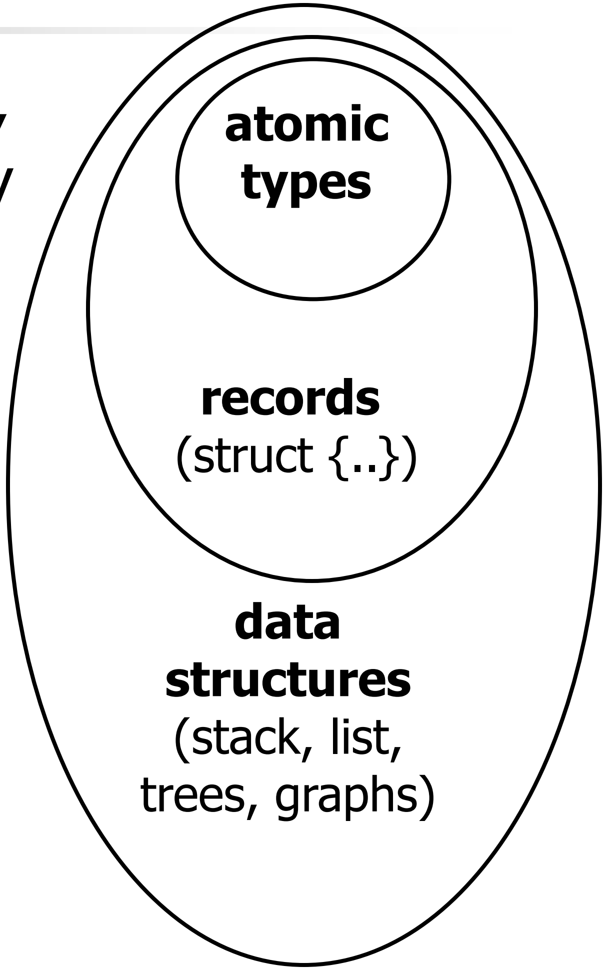
Thanks to Eric S. Roberts, the author of our textbook, for providing some slides/figures/programs.

Objectives

- To appreciate the concept and purpose of **abstract data types**, or ADTs
- To understand both the **abstract behavior** and the **underlying implementation** of the **stack** data type
- To be able to use **incomplete type** mechanism in ANSI C to define ADTs
- To recognize that ADTs provide an attractive alternative to maintaining encapsulated state within a module.
- To understand the design and implementation of a **scanner abstraction** based on ADTs (self-study)

Data Structure hierarchy

- The atomic data types—such as **int, char, double, and enumerated** types—occupy the lowest level in the hierarchy.
- To represent more complex information, we **combine the atomic types** to form larger structures (e.g., struct A {...}).
- These larger structures can then be assembled into even larger ones in an open-ended process using **pointers**.
- Collectively, these assemblages of information into more complex types are called **data structures.**



Abstract data Type (ADT)

- It is usually far more important to know how a data structure **behaves** rather than how it is **represented** or **implemented**
- A type defined in terms of its **behavior** rather than its **representation** is called an **abstract data type (ADT)**
- ADTs are defined by an **interface** (recall ch3)
  - ***Simplicity***. *Hiding the internal representation from the client means that there are* fewer details for the client to understand.
  - ***Flexibility***. *Because an ADT is defined in terms of its* ***behavior****, the lib programmer* who implements one is free to change its underlying representation. As with any abstraction, it is appropriate to change the implementation as long as the interface remains the same so app programmer will not know the changes in lib implementation.
  - ***Security***. *The interface boundary acts as a wall that protects the implementation and* the client from each other. If a client program has access to the representation, it can change the values in the underlying data structure in unexpected ways. Making the data private in an ADT prevents the client from making such changes.

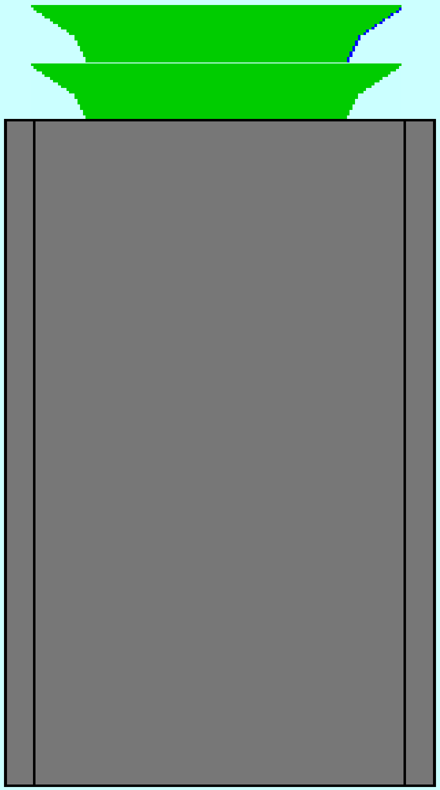
Stacks

- To understand the concept of ADT, we consider a specific data structure, namely **stack**
  - a storage for a collection of data values (**elements**)
  - values are removed from a stack in the opposite order from which they were added, so that the **last** item added to a stack is always the **first** item that gets removed.
  - Adding a new element to a stack is called **pushing**
  - Removing the most recent item from a stack is called **popping**
- So the defining **behavior** of stack is

  "Last in, First out" (LIFO)

The Stack Metaphor

- A **stack** is a data structure in which the elements are accessible only in a **last-in/first-out** order.
- The fundamental operations on a stack are **`push`**, which adds a new value to the top of the stack, and **`pop`**, which removes and returns the top value.
- One of the most common metaphors for the stack concept is a spring-loaded storage tray for dishes. Adding a new dish to the stack pushes any previous dishes downward. Taking the top dish away allows the dishes to pop back up.



Stacks turn out to be particularly useful in a variety of programming applications.

APPLICATIONS OF STACKS

Applications of Stacks

- The primary reason that stacks are important in programming is that **nested function calls** behave in a stack-oriented fashion, recall Fact(n) example
- Pocket calculator example from the textbook
- Compiler example: check if bracketing operators (parentheses, brackets, and curly braces) in a string are properly matched.

Stacks and pocket calculators

- Suppose you want to compute

  50.0 * 1.5 + 3.8 / 2.0

- In a (reverse Polish notation, or RPN) calculator, you will do this as follows:

  50.0 (ENTER) 1.5 (x) 3.8 (ENTER) 2.0 (/) (+)

- ENTER: PUSH the previous value on a stack
- Arithmetic operator (+ - * /):
  - if the user has just entered a value push it on the stack
  - Else
    - POP the top two values from stack
    - Apply the arithmetic OP
    - PUSH the result on the stack

Exercise: Stack Processing

Write a C program that checks whether the bracketing operators (parentheses, brackets, and curly braces) in a string are properly matched. As an example of proper matching, consider the string

```
{ s = 2 * (a[2] + 3); x = (1 + (2)); }
```

If you go through the string carefully, you discover that all the bracketing operators are correctly nested, with each open parenthesis matched by a close parenthesis, each open bracket matched by a close bracket, and so on.

OperatorMatching

```
Enter string: { s = 2 * (a[2] + 3); x = (1 + (2)); }
Brackets are properly nested
Enter string: (a[2] + b[3)
Brackets are incorrect
Enter string:
```

If we have a Stack ADT, it will be easy to solve this problem…

HOW?

- **stack.h** defines the behavior of Stack ADT
  - Export data types
  - Export prototypes for public functions (new_stack, push, pop, etc.)
- **stack.c** implements the public and private functions

DEVELOPING A STACK ADT

stack lib

Data Types in Stack ADT

- Before implementing stack ADT, we need to consider two important types:
  - The type of the element stored in the stack
  - The type of the stack data structure itself
- We must decide whether each type is
  - part of the library implementation or
  - part of the client’s domain/application

First think about **the type of the element**

- For example, double in calculator, char in bracket matching
- Stack element type belongs to the client's domain/application
- Stack implementation does not need to worry about the type of elements. All it needs to store and return an element with *any type*
- So if C had **any** type, we would simply use `any` for data elements (some languages have this – called polymorphism)
- But, C has no such type and requires specific **types** when the exported functions declare their parameters…
- The closest thing C provides is the type `void *` which is compatible with **any pointer** type
- SO, ONE SOLUTION TO DECIDING THE TYPE OF STACK ELEMENT
  - Define stack element type as `void *`
  - Let client application allocate memory for any element and give its pointer to stack ATD
  - Our Stack ADT will push/pop pointers to/from stack (GREAT FLEXIBILITY) but
  - For some applications dealing with pointers might be to complicated and inefficient!

The type of the element (second solution)

- If you allow **client** to access the source code of stack library or package, you can increase the flexibility and efficiency by using **typedef**
- In stack.h define

```
typedef double StackElementT;
```

- If client wants to use stack ADT with char type, all he/she needs to do change the above definition with

```
typedef char StackElementT;
```

  - - Client will edit stack.h *(violates principle of abstraction)*
  - - Client should have the source code to compile
- There is no optimal design strategy. The best you can do

```
typedef void *StackElementT;
```

The type of the stack itself

- Stack type definitely belongs to the library implementation of stack ADT
- Your implementation should be able to
  - push values of `StackElementT` onto a stack
  - retrieve them in a LIFO order when popped
- To perform these operation you need to choose a representation for stack
- Client should not see the implementation details or internal representations. Why?

Opaque type

- In stack.h we can define an **opaque** type such that its underlying representation is hidden form client (it is later implemented in stack.c)

```
typedef struct nameCDT *nameADT;
```

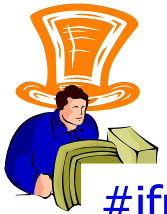
- For Stack ADT:
  - We will have the following **incomplete type** in **stack.h**

    ```
    typedef struct stackCDT *stackADT;
    ```

  - We then define the concrete type in implementation **stack.c**

    ```
    struct stackCDT {
        field declarations
    }
    ```

Defining stack.h Interface



```
#ifndef _stack_h      /* for comments see actual stack.h in the textbook */
#define _stack_h

#include "genlib.h"

typedef double stackElementT;    // char, void *, etc…

typedef struct stackCDT *stackADT;

stackADT        NewStack(void);
void            FreeStack(stackADT stack);
void            Push(stackADT stack, stackElementT element);
stackElementT   Pop(stackADT stack);
bool            StackIsEmpty(stackADT stack);
bool            StackIsFull(stackADT stack);
int             StackDepth(stackADT stack);
stackElementT   GetStackElement(stackADT stack, int index);

#endif
```

As library developer we need to also implement stack.c

For the time being, suppose we implemented stack.c

So stack lib is ready to be used by applications.

A client/driver using stack.h: rpncalc.c

```c
#include "stack.h"  /*  other libraries … */
main()
{
   stackADT operandStack;
   string line;
   char ch;
   operandStack = NewStack();
   while (TRUE) {
      printf("> ");  line = GetLine();
      ch = toupper(line[0]);
      switch (ch) {
        case 'Q': exit(0);
        case 'H': HelpCommand(); break;
        case 'C': ClearStack(operandStack); break;
        case 'S': DisplayStack(operandStack); break;
        default:
          if (isdigit(ch)) {
              Push(operandStack, StringToReal(line));
          } else {
              ApplyOperator(ch, operandStack);
          }
          break;
      }
   }
}
```

```c
void ApplyOperator(char op, stackADT operandStack)
{
   double lhs, rhs, result;

   rhs = Pop(operandStack);
   lhs = Pop(operandStack);
   switch (op) {
     case '+': result = lhs + rhs; break;
     case '-': result = lhs - rhs; break;
     case '*': result = lhs * rhs; break;
     case '/': result = lhs / rhs; break;
     default:  Error("Illegal operator %c", op);
   }
   printf("%g\n", result);
   Push(operandStack, result);
}
static void ClearStack(stackADT stack)
{
   while (!StackIsEmpty(stack)) {
      (void) Pop(stack);
   }
}
static void DisplayStack(stackADT stack)
{
   int i, depth;

   printf("Stack: ");
   depth = StackDepth(stack);
…
}
static void HelpCommand(void)
{…}
```

```
#ifndef _stack_h     /* for comments see actual stack.h in
the textbook */
#define _stack_h

#include "genlib.h"

typedef double stackElementT;    // char, void *, etc…

typedef struct stackCDT *stackADT;


stackADT NewStack(void);
void FreeStack(stackADT stack);
void Push(stackADT stack, stackElementT element);
stackElementT Pop(stackADT stack);
bool StackIsEmpty(stackADT stack);
bool StackIsFull(stackADT stack);
int StackDepth(stackADT stack);
stackElementT GetStackElement(stackADT stack, int index);

#endif
```

IMPLEMENTATION OF STACK.C

Concrete Data Type (CDT)

- First provide a concrete representation for abstract type **stackADT** in stack.c
  - Suppose we decided to use ARRAY to hold elements on the stack

```
#define MaxStackSize 100
struct stackCDT {
  stackElementT elements[MaxStackSize];
  int count;
};
```

- We then implement the exported (public) and private functions

stack.c

```
#include “stack.h” /* other libraries … */

#define MaxStackSize 100
struct stackCDT {
  stackElementT elements[MaxStackSize];
  int count;
};

stackADT NewStack(void)
{
    stackADT stack;

    stack = New(stackADT);
    stack->count = 0;
    return (stack);
}

void FreeStack(stackADT stack)
{
    FreeBlock(stack);
}
```

```
void Push(stackADT stack, stackElementT element)
{
    if (StackIsFull(stack)
       Error(“Stack Size exceeds”);
    stack->elements[stack->count++] = element;
}

stackElementT Pop(stackADT stack)
{
    if (StackIsEmpty(stack))
              Error("Pop of an empty stack");
    return (stack->elements[--stack->count]);
}

bool StackIsEmpty(stackADT stack)
{
    return (stack->count == 0);
}

bool StackIsFull(stackADT stack)
{
    return (stack->count == MaxStackSize );
}
```

End of stack.c

```
int StackDepth(stackADT stack)
{
    return (stack->count);
}

stackElementT GetStackElement(stackADT stack,
                              int index)
{
    if (index < 0 || index >= stack->count) {
        Error("Non-existent stack element");
    }
    return (stack->elements[stack->count - index - 1]);
}
```

count	elements	index
[6] ⟶		
[5]	F	0
[4]	E	1
[3]	D	2
[2]	C	3
[1]	B	4
[0]	A	5

Improving stack.c implementation using dynamic array while keeping stack.h as is

```
#include “stack.h” /* and other libraries … */

#define InitialStackSize 100
struct stackCDT {
  stackElementT *elements;
  int count;
  int size;
};
```

```
// instead of
#define MaxStackSize 100
struct stackCDT {
  stackElementT elements[MaxStackSize];
  int count;
};
```

```
/* Prototype for Private functions, NOT exported in stack.h */
static void ExpandStack(stackADT stack);
stackADT NewStack(void)
{
    stackADT stack;

    stack = New(stackADT);
    stack->elements = NewArray(InitialStackSize, stackElementT);
    stack->count = 0;
    stack->size = InitialStackSize;
    return (stack);
}
```

```
void Push(stackADT stack, stackElementT element)
{
    if (stack->count == stack->size) ExpandStack(stack);
    stack->elements[stack->count++] = element;
}
stackElementT Pop(stackADT stack)
{
    if (StackIsEmpty(stack)) Error("Pop of an empty stack");
    return (stack->elements[--stack->count]);
}

bool StackIsEmpty(stackADT stack)
{
    return (stack->count == 0);
}
bool StackIsFull(stackADT stack)
{
    return (FALSE);
}
void FreeStack(stackADT stack)
{
    FreeBlock(stack->elements);
    FreeBlock(stack);
}
```

```
int StackDepth(stackADT stack)
{
    return (stack->count);
}
stackElementT GetStackElement(stackADT stack, int index)
{
    if (index < 0 || index >= stack->count) {
        Error("Non-existent stack element");
    }
    return (stack->elements[stack->count - index - 1]);
}
/* Private functions, NOT exported in stack.h */
static void ExpandStack(stackADT stack)
{
    stackElementT *array;
    int i, newSize;

    newSize = stack->size * 2;
    array = NewArray(newSize, stackElementT);
    for (i = 0; i < stack->size; i++) {
        array[i] = stack->elements[i];
    }
    FreeBlock(stack->elements);
    stack->elements = array;
    stack->size = newSize;
}
```

Exercise: Proper matching of { ([ ]) }

- Write a C program that checks whether the bracketing operators (parentheses, brackets, and curly braces) in a string are properly matched.
- For example,

```
{ s = 2 * (a[2] + 3); x = (1 + (2)); }
```

is a proper matching,

Suppose you can change stack.h so that it exports
typedef char stackElementT;

```
#ifndef _stack_h
#define _stack_h

#include "genlib.h"

typedef char stackElementT;

typedef struct stackCDT *stackADT;

stackADT NewStack(void);
void FreeStack(stackADT stack);
void Push(stackADT stack, stackElementT element);
stackElementT Pop(stackADT stack);
bool StackIsEmpty(stackADT stack);
bool StackIsFull(stackADT stack);
int StackDepth(stackADT stack);
stackElementT GetStackElement(stackADT stack, int index);

#endif
```

Exercise: Reimplementation of RPN calculator and Proper matching

```
#ifndef _stack_h
#define _stack_h

#include "genlib.h"

typedef void *stackElementT;

typedef struct stackCDT *stackADT;

stackADT NewStack(void);

void FreeStack(stackADT stack);

void Push(stackADT stack, stackElementT element);

stackElementT Pop(stackADT stack);

bool StackIsEmpty(stackADT stack);

bool StackIsFull(stackADT stack);

int StackDepth(stackADT stack);

stackElementT GetStackElement(stackADT stack, int index);
#endif
```

- Suppose you cannot change stack.h and instead of char or double, it currently exports typedef void *stackElementT;
- Given this version of stack.h, implement RPN calc and proper matching
- What will be the main difference
  - Dynamically allocate memory for each value that you push, free when you pop and/or reuse…

Home Exercise

A client/driver using stack.h: rpncalc.c

Modify this

```
#include "stack.h"  /*  other libraries … */
main()
{
    stackADT operandStack;
    string line;
    char ch;
    operandStack = NewStack();
    while (TRUE) {
        printf("> ");  line = GetLine();
        ch = toupper(line[0]);
        switch (ch) {
          case 'Q': exit(0);
          case 'H': HelpCommand(); break;
          case 'C': ClearStack(operandStack); break;
          case 'S': DisplayStack(operandStack); break;
          default:
            if (isdigit(ch)) {
                Push(operandStack, StringToReal(line));
            } else {
                ApplyOperator(ch, operandStack);
            }
            break;
        }
    }
}
```

```
void ApplyOperator(char op, stackADT operandStack)
{
    double lhs, rhs, result;

    rhs = Pop(operandStack);
    lhs = Pop(operandStack);
    switch (op) {
      case '+': result = lhs + rhs; break;
      case '-': result = lhs - rhs; break;
      case '*': result = lhs * rhs; break;
      case '/': result = lhs / rhs; break;
      default:  Error("Illegal operator %c", op);
    }
    printf("%g\n", result);
    Push(operandStack, result);
}
static void ClearStack(stackADT stack)
{
    while (!StackIsEmpty(stack)) {
        (void) Pop(stack);
    }
}
static void DisplayStack(stackADT stack)
{
    int i, depth;

    printf("Stack: ");
    depth = StackDepth(stack);
…
}
static void HelpCommand(void)
{…}
```

SCANNER ADT

STUDY Section 8.5 from the textbook

```
This line contains 10 tokens.
```

This		**line**		**contains**		**10**		**tokens**	**.**

Next homework may use scannerADT

Danger of Encapsulated State

- You can declare a global variable in a module to maintain state between functions (e.g., randword.c, scanadt.h in previous book)
  - static will make it private (encapsulated state)
- The module that uses it can have only one copy of the state information,
  - So we cannot use that module in different parts of the program
  - In case of layered abstractions, client may not know anything about the underlying modules and use them in different places, resulting in error…
- ADT provide a safe alternative to encapsulated state