

# Abstract Data Types (ADTs)

- **What is data abstraction?**
  - Stack ADT
  - Abstract/algebraic specification
- **Object-orientation**
  - Data abstraction + code sharing through inheritance
- **Object-oriented design**
  - Examples (in C vs C++)

1

## ADTs

user-defined types  
with associated  
operations       $\implies$       data abstractions       $\implies$       OOPLs

***Data abstraction* - a mechanism which encapsulates or hides the representation and operation details of a datatype from its users. All access is through a defined interface.**

**Appears in C++, Java, many others...**

2

# ADTs

- **Advantages**
  - Can modify the ADT implementation without affecting users of the ADT
  - User's changes can't affect the ADT
  - Encourages modularity in programs enabling construction of larger, more complex codes by more than one programmer
- **Possible disadvantages**
  - Writer of ADT has to put more effort into design of a useful interface of operations

3

# ADTs

- **ADTs are usually designed with some invariant of its behavior in mind**
  - E.g., Stack invariant:  
*top of stack is always the last element added*
    - Need to initialize stack within the ADT
    - Must keep some variables private (*top\_of\_stack*)
    - May keep some operations private to prevent illegal use (e.g., check *isEmpty()* when doing a *pop*)

4

# Stack ADT

## Abstract Specification

- Tells behavior of ADT; offers enough information for use of the ADT
- Operations:
  - *createStack*: Stack
  - *push*: Stack, element  $\rightarrow$  Stack
  - *pop*: Stack  $\rightarrow$  Stack, element
  - *peek*: Stack  $\rightarrow$  element
  - *isEmpty*: Stack  $\rightarrow$  boolean
- Variables:
  - *s*: Stack
  - *x*: element

5

# Stack ADT

## Abstract Specification

- Axioms:
  - *isEmpty(createStack()) = true*
  - *isEmpty(push(s, x)) = false*
  - *peek(createStack()) = error*
  - *peek(push(s, x)) = x*
  - *pop(createStack()) = error*
  - *pop(push(s, x)) = s*

6

# Objects

- **What do objects have that ADTs don't?**
  - **Inheritance**
    - Allows code sharing or reuse between related types
  - **Control over visibility of members**
    - Everything is not necessarily private
  - **Generics (parameterized types, explicit polymorphism)**
    - defining a type as a function of another one  
(a linked list of X, where X = int, struct part, ...)

7

# OO Design

- **Process of designing a collection of classes and objects to simulate the entities in an application domain**
- **Think of a system in terms of its independent components**
  - Begin with the *data* and build programs from the *bottom up*
  - Separate specification from implementation
- **Build interfaces between components that are natural for the way they interact**

8

# Elevator control system

- **OO Example: elevator control system**
  - **What are the objects? elevator, floor, control mechanism**
  - **What are their subcomponents?**
    - **Elevator**
      - Control panel with buttons, lights
      - Door
      - Emergency phone
    - **Floor -**
      - Control panel with buttons, lights
      - Door
      - Indicator lights for floors reached
  - **Control mechanism consists of multiple elevators, floors and a program to optimize traversals**

9

## Example - Key Ideas

- **Parts of the system are independent with limited interfaces**
- **Want to allow implementation of any part of the system to be easily changed**
- **Objects can contain other objects**
  - Elevator has buttons
- **May have same type of object in several other objects**
  - Lights in floor buttons and in elevator buttons

10

## Top-down Design

- **Top-down design of imperative programs (1970's)**
  - Wrote program in very high-level language and *successively refined* these into lower-level descriptions which eventually were code;
  - Idea was correctness of code was maintained by these successive transformations;
  - Often similar pieces of code used in very different parts of the program

11

## OO Design

- **Advantages**
  - Reuse of abstractions
  - Can re-implement abstractions to achieve performance (or other) goals, without affecting the design of the software at a higher level

12

# How to design an OO program?

- **Identify the objects and their behavior**
  - Choose level of abstraction, specific implementation, potential for reuse
  - What services does each object provide?
- **Identify relationships between objects**
  - Does one object specialize another?
- **Create a Java/C++ class public interface to represent these objects**
  - *A class should have access to all and only the data it needs to perform its work!*

13

## Object Relationships

- **Is-A**
  - Type T1 is in the is-a relationship with Type T2, if every entity of type T1 is a member of type T2
    - inheritance (T2 base or superclass, T1 derived or subclass)
    - T1 can be used anywhere T2 is used
- **Has-A**
  - Entity type T1 is in the has-a relationship with an entity of type T2, if T2 is part of T1 (T1 stores an instance of T2)
    - Class level (complete containment)
    - Instance level (shared instance via a reference)

14

# Object Relationships

- **Uses-A**
  - occurs when type T1 uses-a instance of T2 (it uses the object, and then discards it)
  - also occurs when one class instance takes another class instance as a parameter

15

# Stack as an ADT

- **Assume given eltType (type of values to be stored in the stack)**
- **Operations**
  - Create and destroy stacks
  - Push, pop, peek, isEmpty? on stack contents
- **Representation choices**
  - Arrays (statically allocated of max size, or dynamically allocated) or linked lists

16

## Stack ADT in C

```
stack *create();  
bool isempty(stack *);  
bool isfull(stack *);  
void push(element, stack *);  
void pop(stack *);  
element peek(stack *);  
void destroy(stack *);
```

17

## 1<sup>st</sup> Stack type in C: Structure (with static array)

```
typedef int element;  
  
#define MAX 20  
#define EMPTY -1  
  
typedef struct stackinfo {  
    element s[MAX];  
    int top;  
} stack;
```

18

## 2<sup>nd</sup> Stack type in C: Linked list

```
typedef struct Cell {
    element info;
    struct Cell *link;
} CellType;

typedef struct stackinfo {
    CellType *top;
} stack;
```

- Now, need to implement all functions on stacks using this new representation:

```
stack *create();
bool isempty(stack *);
...
```

19

## Implementation independence in C

- Use forward declarations, which means you can only create pointers, and not do any arithmetic

```
#include <stdio.h>

/* Use forward declaration for stack */
struct stackinfo;
typedef struct stackinfo stack;

int main()
{
    . . .
}
```

20

## Issues with C Stacks

- **Can use header file (\*.h) to specify function signatures for Stack operations - good**
- **Need to show all-or-none data representation in header file to achieve independent compilation**
- **User might rely on a certain (visible) implementation of Stacks and write implementation-dependent code**

21

## ADT Functions

- **Must understand what users of the ADT need and then design an interface**
- **Requires operations to**
  - Construct/initialize/destroy ADT instance
  - Observe component of ADT: `get()`
  - Mutate component of ADT: `set()`
  - Compare ADT instances
  - Copying ADT instances
  - ...
- **Implementer must make sure to preserve implementation invariants**

22

# Interface to Stack in C++

## stack.h

```
class Stack {
public:
    Stack();
    ~Stack();
    bool isempty();
    bool isfull();
    void push(element data);
    void pop();
    element peek();

private:
    element s[MAX];
    int top;
}
```

23

# Stack in C++

## stack.cc

```
Stack::Stack() {
    top = EMPTY;
}
Stack::~Stack() {
    for(i=0;i<=top;i++)
        delete s[i];
}
bool Stack::isempty() {
    return top == EMPTY;
}
bool Stack::isfull() {
    return top == MAX-1;
}
```

```
void Stack::push(element data) {
    s[++top]=data;
}
void Stack::pop() {
    top--;
}
element Stack::peek() {
    return s[top];
}
```

*For statically allocated array representation type*

24