Abstract Data Types

• What is data abstraction?
  – Stack ADT
  – Algebraic specification
  – Generics

• Object-orientation
  – Data abstraction + code sharing through inheritance

• Object-oriented design
  – Examples (in C vs C++)
user-defined types
with associated operations

\[\rightarrow\] data abstractions
\[\rightarrow\] 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 Modula, Ada, CLU, Smalltalk-80, C++, Java
Pizza Story

• Task: design and code an order entry system for Personal Pizza
  – Subtask: represent a pizza order
    • pizzas differ only in what toppings
    • only toppings are mushrooms, anchovies, pepperoni, extra cheese.
  – Other subtasks:
    • … lots of code using datatype pizza
Pizza Story

- Success! Marketing! Focus groups! (Argh!!)
- Personal Pizza expands its menu:
  - a different set of toppings each day
  - Need to fix order entry program (double argh!!)
- New representation for a pizza:
  - array of strings
    - pineapple
    - onions
Pizza Story

- **Problem:** all that code that used old representation has to be fixed
- **Good news:** we divided all that old code into two pieces
  - A: code that implemented meaningful operations on a pizza (create pizza, add topping)
  - B: code that used functions from part A to do order entry (create order, print in kitchen)
- **Only part A has to be fixed!** (Well, mostly)
• 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
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*)
Stack ADT-Abstract Spec

• 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

CLU specification for ADT operations

pop:
  requires--non-empty stack
  effects--removes top element from stack
  modifies--stack contents
Stack ADT- Abstract Spec

• Axioms:
  – $\text{isEmpty}(\text{createStack}) = true$
  – $\text{isEmpty}(\text{push}(s, x)) = false$
  – $\text{peek}(\text{createStack}) = \text{error}$
  – $\text{peek}(\text{push}(s, x)) = x$
  – $\text{pop}(\text{createStack}) = \text{error}$
  – $\text{pop}(\text{push}(s, x)) = x$
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 (e.g., classes, methods)
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
Elevator control system

• 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
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
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
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
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!
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
    - Public inheritance, T2 base or superclass, T1 derived or subclass

- **Has-A**
  - Entity type T1 is in the has-a relationship with an entity of type T2, if T2 is part of T1 or T1 uses T2 for implementation
    - Class level (complete containment)
    - Instance level (shared instance via a reference)
Object Relationships

• **Uses-A**

  - Occurs when one class instance takes another class instance as a parameter

```
Person
  Is-a
  Student
    Has-a
      Class
  Is-a
  Employee
    Uses-a
      Gym_locker
```
Designing a Solitaire Game

- Establish the entities needed to simulate the game
- Decide on classes needed
- Decide what actions these objects should implement
- Organize placement of actions in class hierarchy to maximize code sharing while retaining understandability of the code

*(more on this in class)*
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
Stack in C

/* STACK in C, as a DYNAMICALLY ALLOCATED DATA STRUCTURE */
typedef int eltType; /* using parameterized genericity */

#define MAX  20
#define EMPTY -1
typedef struct stack {
    eltType s[MAX];             // --the array containing the elements
    int top; } stkType;         // --the index of the topmost element
#define bool int
Stack in C

```c
void create(stkType **stkP)
    {
        *stkP = (stkType *) malloc(sizeof(stkType));
        (*stkP)->top = EMPTY;
    }

bool isempty(stkType *stkP)
    {return (bool) (stkP->top == EMPTY);}$/**no error chk*/

bool isfull(stkType *stkP)
    {return (bool) stkP->top == (MAX - 1);}$/**no err chk*/

void push(eltType data, stkType *stkP)
    {stkP->s[++(stkP->top)] = data;} /*no error check*/

void pop(stkType *stkP)
    {((stkP->top)--);} /*no error check :-(

eltType peek(stkType *stkP)
    {return stkP->s[(stkP->top)];} /*no error check*/

void destroy(stkType *stkP) {free(stkP);}
Stack in C

```c
int main()
{
    stkType *x;
    create(&x); /*call by reference*/
    push(2,x);
    push(3,x);
    printf("%d\n", peek(x));
    pop(x);
    printf("%d\n", peek(x));
    pop(x);
    //printf("%d\n", peek(x)); //should be error
    destroy(x);
} /*output
  3
  2
  */
```
2nd Stack type in C

/* STACK in C, as a LINKED LIST */
/* ---------------STACK_2 def's ---------------*/
typedef struct Cell {
    eltType info;
    struct Cell *link;} CellType;
typedef struct stack{
    CellType *top;} stkType;

Need to rewrite the implementations of all functions on Stacks to operate on this new representation type

create(stkType **stkp)
isempty(sktType *stkP)
void push (eletype data, stkType *stkP)
ETC.
Problems with C Stacks

• Can use header file (*.h) to specify function signatures for Stack operations - good
• Need to show the data representation in header file to achieve independent compilation
• NO ENCAPSULATION
• User might rely on a certain (visible) implementation of Stacks and write implementation-dependent code
ADT Functions

• Must to understand what users of the ADT need and then design an interface
• Requires operations to
  – Construct/initialize/destruct 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
Interface to Stack in C++

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

    private:
        eltType s[MAX]; // static array rep
        int top;
}

#define MAX 20 // static const int MAX
#define EMPTY -1 // static const int EMPTY

A possible improvement
make MAX and EMPTY
static consts so there is
only 1 of each per Stack
class.

Needed in Stack.cc:
    int Stack::EMPTY=−1;
    int Stack::MAX = 20;

This would be stack.h
```
2 versions of Stack in C++

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

int main()
/*STATIC allocation */
{ Stack x; //Stack() invoked
  //automatically
  x.push(400);
  x.push(303);
  cout << "\n" << x.peek();
  x.pop();}

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

int main()
/* DYNAMIC allocation */
{ Stack *ps;
  ps = new Stack();
  ps->push(400);
  ps->push(303);
  cout << "\n" << ps->peek();
  ps->pop();
  ...
  ...
  delete ps;
} // ~Stack() invoked automatically

This would be stack.cc
Stack in C++

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(eltType data) {
    if ( !isfull() ) s[++top]=data;
    else error("stack is full; cannot push\n");
}
void Stack::pop() if ( !isempty() ) top--;
    else error(" stack is empty; cannot pop\n");
eltType Stack::peek() {return s[top];}

For statically allocated array representation type