

Abstract Data Types

- Abstract data types are a way to make a program look more like the application and less like the hardware.
 - Makes up for deficiencies of original language (e.g., C is missing strings).
- For each type we usually have ways of
 - *Declaring* variables of that type (e.g., `typedef char * string; string x`)
 - Possibly allocating space to a value of that type (e.g., `string name = (char *) malloc(#letters)`)
 - Writing (some) *constants* (e.g., `"abc"`)
 - Performing operations involving these values (e.g.
 - `strlen: string --> int`
 - `strcat: string x string --> string`
 - `substr: string x int x int --> string`
 - `strsql: string x string --> boolean`
- **Problem:** we get caught up in nasty details (packed vs unpacked in Pascal, `\0` terminator in C,...)

Stack: the 'canonical' ADT

- Assume given `eltType` -- the type of values to be stored on the stack
- `stkType` needs roughly the following operations:
 - create and destroy stacks
 - push, pop, and peek at the stack contents;
 - check if the stack is empty or full, so we don't get errors
- There are several standard ways of implementing stacks:
 - Using an array to hold the value
 - Using a linked list to hold the values
 - In the first case, the array could be allocated *statically* (by the compiler on the execution stack, at "variable elaboration time"), or *dynamically* (by the programmer, on the heap, while the program is executing.)

Stack: the 'canonical' ADT

stkType + prototypes of functions involving it:

```
stkType create ();
void destroy(stkType stack);

void push(eltType val, stkType stack);
void pop(stkType stack);
eltType peek(stkType stack);

bool isEmpty(const stkType stk);
bool isFull(const stkType stk);
```

(Because of call-by-value, if we want ops with side-effects, then either **stkType** parameter must be a pointer to a fixed entity, or we must pass in a pointer instead **stkType ***. In the first case, **const** gives a sign that we do not expect/want the argument to change, even locally. (Mostly of help to compiler)

```
/* STACK in C, as a STATICALLY ALLOCATED DATA STRUCTURE */
#include <stdio.h> //<--
typedef int bool; #define TRUE 1 #define FALSE 0
typedef int eltType;

/* -----STACK.1 def's -----*/
#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

void create(stkType *stkP) //just initialize
    {(*stkP).top = EMPTY;}
bool isempty(stkType stk)
    {return (bool) ((stk).top == EMPTY);}
bool isfull(stkType stk)
    {return (bool) ((stk).top == MAX - 1);}
void push(eltType data, stkType *stkP)
    {(*stk).s[++(*stkP).top]=data;} //no error check :- (
void pop(stkType *stkP)
    {(*stkP).top--;} //no error check :- (
eltType peek(stkType stk)
    {return ((stk).s[(stk).top]);} //no error check :- (
void destroy(stkType *stkP) {} //YES! Compiler does it
```

remember to pass **stkType**
whenever the stack (.t
modified

Using stack1.c:

```
// **** Using the stack ****
int main()
{
    stkType x; //all space allocated here

    create(&x);
    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);
}
```

```
/* STACK in C, as a DYNAMICALLY ALLOCATED DATA STRUCTURE */
/* -----STACK.2 def's -----*/
#define MAX 20 // could now be passed in by creator
typedef struct stack {
    eltType *base; //--ADDRESS of the BASE of the array
                //containing elements;
    eltType *top; //--ADDRESS of the *next* free store ;
    eltType *end} //-- ADDRESS of last place to store
    * stkPtType; //<--already a pointer

void create(stkPtType stkP) {
    stkP=(stkPtType) malloc(sizeof(struct stack));
    stkP->base = malloc(MAX * sizeof(eltType));
    stkP->top = stkP->base;  stkP->end=stkP->base+MAX; }
bool isempty(stkPtType stkP)
    {return (bool) (stkP->top == stkP->base);}
bool isfull(stkPtType stkPt)
    {return (bool) (stkPt->top == stkP->end);}
void push(eltType data, stkPtType stkP)
    {*(++stkP->top)=data;} //no error check :-
void pop(stkPtType stkP)
    {stkP->top--;} //no error check :-
eltType peek(stkPtType stkP)
    {return *(stkP->top - 1) //no error check :-
void destroy(stkPtType stkP)
    {free(stkP->base); free(stkP);} //malloc keeps track of calls
```

Using stack2.c:

```
// **** Same as using stack 1 but don't have to bother
// with passing names to functions ****
int main()
{
    stkType x; // only allocates space for one pointer
               // on runtime-stack of C;
    create(x); // allocate here the full data structure;
    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);
}
```

```
                /* STACK in C, as a LINKED LIST */
/* -----STACK.3 def's -----*/
typedef struct Cell {
    eltType info;
    struct Cell* link;} CellType;
typedef struct stack{
    CellType *top;
                } * stkType;

void create(stkPtType stkP)
    {stkP->top = NULL;}
bool isempty(stkPtType stk)
    {return (stk->top == NULL);}
bool isfull(stkPtType stk)
    {return FALSE;}
void push(eltType data, stkPtType stkP)
    {CellType* add = (CellType *) malloc(sizeof(CellType ));
    add->info = data; add->link = stkP->top;
    stkP->top = add;}
void pop(stkPtType stkP)
    {stkP->top=(stkP->top)->link;} //no error check
eltType peek(stkPtType stk)
    {return ((stkP->top)->info);} //no error check :-
void delete(stkPtType stkP) {/* walk down to the end of the
    linked list and free cells one by one; */
```

Using stack.3

- Just the same as Stack2 (or Stack1, except for having to pass the name &x, rather than just the stack x.)
- Therefore can switch between implementations at will.

Can we protect stack from vandalism?

- C header file: stack.h

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

void create(stkType *stkP)
bool isempty(stkType *stkP)
bool isfull(stkType *stkP)
void push(eltType data, stkType *stkP)
void pop(stkType *stkP)
eltType peek(stkType *stkP)
```

- C implementation file: stack.c

```
#include "stack.h"
/* CODE FOR THE FUNCTIONS */
void create(stkType *stkP){ . . . }
bool isempty(stkType *stkP) { . . . }
bool isfull(stkType *stkP) { . . . }
void push(eltType data, stkType *stkP) { . . . }
void pop(stkType *stkP) { . . . }
eltType peek(stkType *stkP) { . . . }
```

- Using the stack adt in C

```
// **** Using the stack ****
#include "stack.h" //<-- only new thing
int main()
{
    stkType x;

    setup(&x);
    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}
}
```

- will not give away implementation of operations
- unfortunately, it requires you to show the data structures (in order to have independent compilation)
- this means someone can maliciously/inadvertently mess around with implementation even in user program
- also user can write program relying on some property of implementation

ADT01 Ryder/Borgida

11

Abstract Data Types

- Enforce *encapsulation* (cannot modify implementation)
- Try to *hide information* (don't give user any idea about how ADT is implemented)
- Definer of ADT: must gather needs of potential users
 - decide if type is **immutable**
 - e.g., *STRING*: concatentate(x,y) shouldn't normally affect x or y, anymore than x+y affects x or y; (although in C, strcat does :-())
 - Scheme *lists* are immutable (pure functional lang. => every data type is immutable)
 - **mutable**
 - (e.g., *STACK*: pop(s) modifies s, rather than producing an entirely new stack)
- mutable is more efficient, but dangerous when shared;
- define an interface of functions, constants,...

ADT01 Ryder/Borgida

12

What needs to be done and by whom?

- **Designer of ADT (cont'd):** need at least operations for
 - *constructing/initializing*, destructing values (e.g. `createStk`, `makeStr`)
 - *observing*: let you look at relevant component(s) of the adt (at least display them) (e.g., `peekStk`, `isemptyStk`, `isfullStk`, `getIthCharacter`, `str_length`, `print`)
 - *mutators*: if the type is “changeable/mutable”, ops to change it (e.g., `pushStk`, `popStk`)
 - often operations for *comparing* objects (e.g., `stringCompare`, `stringEqual`)
 - for immutable types, *copying* operations (e.g., `stringCopy`)
- **User of ADT:** can use new type just like old ones
- **Implementor of ADT:**
 - must choose a data structure for each data type
 - implement each operation
 - make sure it is correct: use *implementation invariants*

Ada: a language supporting adt

Ada specification:

```
---ADT specification
package STACKS is
  type Stack is private;
  procedure Push(S: in out Stack; Element: in Integer);
  function Pop(S: in out Stack) returns Integer;
  function Is_empty(S: in Stack) returns Boolean;

  private
    type Stack is
      record
        space: array (1..MAX) of integer;
        top: integer range 0..MAX := 0;
      end record;
end; -- of STACKS
```

Ada implementation:

```
---ADT body (implementation)
package body STACKS is
  procedure Push (S: in out Stack; Element: in Integer);
  begin
    if S.top = 100 ---error endif;
    S.top := S.top +1;
    S.space(S.top) := Element;
  end Push;
  function Pop(S: in Stack) returns Boolean is
  begin
    if Is_empty(S) --error endif;
    S.top := S.top -1;
    return S.space(S.top+1);
  end Pop;
  function Is_empty(S: in Stack) returns Boolean is
  begin
    if S.top = 0 then return True
      else return False;
    end Is_empty;
  end STACKS;
```

Implementation Invariant Example

- RATIONAL NUMBERS :

```
deftype struct{ int num,denom } //rational with operations
create_rat(top,bottom), add_rat(r1,r2), mult_rat(r1,r2),
equal_rat(r1,r2)
```

- How do you implement `bool equal_rat(rational x,y);` ?
(`??? x.num==y.num && x.denom==y.denom`)???

Problem : Invariant to solve it

- (3,0) : `denom != 0`
- (0,3) vs (0,4) : `if num==0 then denom==1`
- (-1,2) vs (1,-2) : `denom > 0`
- (1,2) vs (3,6) : `greatest_common_divisor(num,denom)=1 or 0`

- Invariants are contracts agreed to by all operations (e.g.,) so that each can make some assumptions about the implementation data structure

Genericity: a problem for ADTs based on other types

- C stack was really for keeping integers.

- How do we make a stack of strings?

Change

```
typedef int eltType;
```

to

```
typedef char * eltType;
```

- How do we have BOTH a stack of integers and one of strings?

Only by making two copies of the stack ADT, using `eltType1` and `eltType2` ! Unfortunately, this requires copying everything because you need different types for the stacks (`stack1` and `stack2`), and then every operation needs a copy with argument `stack1` and another with argument `stack2`.

¥ Ada allow `type` parameter to be used in defining ADTs (or even procedures)

Ada Generic Stack

generic

```
max : natural;          ---a natural number
type Item is private;  ---Item is the base type of
the abstraction
package STACKS is
  type Stack is limited private
    --- means that assignment
    ---and comparison operations are not defined
    ---unless they are given in the package body
  procedure Push (S: in out Stack; Element: in Item);
  function Pop(S: in out Stack) returns Item;
  function Is_empty(S: in Stack) returns Boolean;
private
  type Stack is record
    space : array(1..max) of Item;
    top: integer range 0..max := 0;
  end record;
end;
```

Ada Generic Stack (cont'd)

¥ This is how we can define particular instances of the generic:

```
declare package My_stack is new STACKS (100, Real); ---
    this instantiates the Stack ADT to be a Stack of REALs
declare package Your_stack is new STACKS(100, Integer); ---
    this instantiates the Stack ADT to be a Stack of integers
```

¥ Once we do this, we can use `My_stack` as a stack of reals

```
x:My_stack;
Push (x, 2.5);
```

Summary: advantages of ADTs

- **Designer of ADT can modify implementation (usually, to improve some kind of performance)**
- **Users cannot change the ADT values, except through the "exported" interface**
- **Encourages modularity in program, and thus supports the creation of larger, more complex systems**
- **Gives the programmer some of the facilities that used to be only available to language designers.**

C++: an extension of C with ADTs +...

- **Basic idea:** (Java-like) *classes* are just
 - structures that are allowed to have function members +
 - visibility rules (public/private)

```
typedef struct
{int val;
 void bump();}
Counter
```

```
class Counter
{public
 int val;
 void bump();}
```

- **Every class has at least one (standardly named) constructor and destructor** (just for intialization, or allocation too)

```
class Counter
{public
 int val;
 void bump();
 Counter();
 ~Counter();}
```

- **Information hiding:**
 - Functions are associated with classes (and not all must be, as in Java) are referred to with qualified names: `Counter::bump()`
 - Only member functions of a class have access to the private parts (usually data) of that class

```
class Counter
{private:
 int val;
public:
 void bump();
 int getValue();}
```

- **+ Object orientation:**
 - Every function member has special (implicit) parameter, **this**, which allows access to all other members of the class:

```
void bump(){this.val +=1};
void bump(){val +=1}; // can be left out
```
 - **OO-like syntax for function invocation:**

```
Counter ct;
ct.bump();
```

Stack in C++

stack.h

```
#define MAX 20
class Stack {
private:
    eltType s[MAX];
    int top;
public:
    void create();
    bool isempty();
    bool isfull();
    void push(eltType data);
    void pop();
    eltType peek();
}
```

stack.cc

```
void Stack::create() {top = EMPTY;}
bool Stack::isempty()
    {return top == EMPTY;}
bool Stack::isfull()
    {return top == (MAX - 1);}
void Stack::push(eltType data)
    {s[++top]=data;}
void Stack::pop()
    {top--;}
eltType Stack::peek()
    {return s[top];}
) void Stack::destroy() {}
```

ADT01 Ryder/Borgida

23

Using stack.cc vs. stack2.c:

C++

```
int main()
{
    Stack x;

    x.push(2)
    x.push(3)
    cout<<\n<<x.peek();
    x.pop();
    cout<<\n<<x.peek();
    x.pop();
}
```

C

```
int main()
{
    stkType x;
    create(x);
    push(2,x);
    push(3,x);
    printf("%d\n", peek(x) );
    pop(x);
    printf("%d\n", peek(x) );
    pop(x);
    destroy(x);
}
```

Implicit
invocations



ADT01 Ryder/Borgida

24