# CS 126 Lecture P5: Abstract Data Type

## Outline

- **Introduction**
- Stacks (and queues)
- Stack and queue applications

# Data Type and ADT

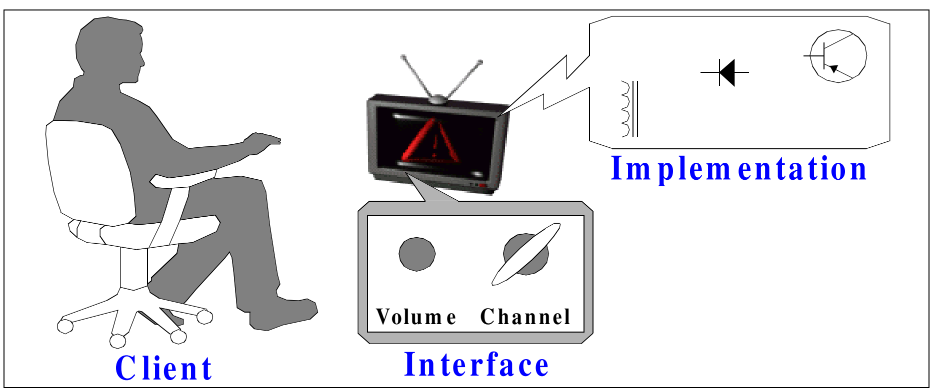
Data type
- set of values
- collection of operations on those values

ex: short int
- set of values between -32768 and 32767
- arithmetic operations + - * /

Abstract data type (ADT)
- data type whose representation is hidden



# Interface, Implementation, and Client




- Separate implementation from specification
  - INTERFACE: specify the allowed operations
  - IMPLEMENTATION: provide code for ops
  - CLIENT: code that uses them

# Advantages of ADT

Representation is hidden in the implementation
Client program does not need to know
*how* the implementation works

- Advantages
  - * different clients can use the same ADT
  - * can change ADT without changing clients
- Convenient way to organize large programs
  - decompose into smaller problems
  - substitute alternate solutions
  - separate compilation
  - build librarys

Powerful mechanism
for building layers of abstraction

Client can work at higher level of abstraction



# "Non-ADTs"

**interface:** `typedef struct {int p; int q;} Rational;`
**client:** `Rational a; a.p = 3;` **Non-ADT**

**ADT**

**interface:**
```
typedef struct {int p; int q;} Rational;
void setRationalP(Rational *r; int x);
```
**implementation:**
```
void setRationalP(Rational *r; int x)
{r->p = x;}
```
**client:**
```
Rational a; setRationalP(&r, 3);
```

- Rational data type (Assignment 3) is NOT an ADT
  representation is in interface

Are C built-in types ADTs ?
ALMOST: we generally ignore representation
NO: set of values depends on representation
YES: good programs use <limits.h> to function
properly independent of representation

# Outline

- ~~Introduction~~
- **Stacks (and queues)**
- Stack and queue applications



# Stack and Queue Definitions

- Prototypical data types
  set of operations on generic data

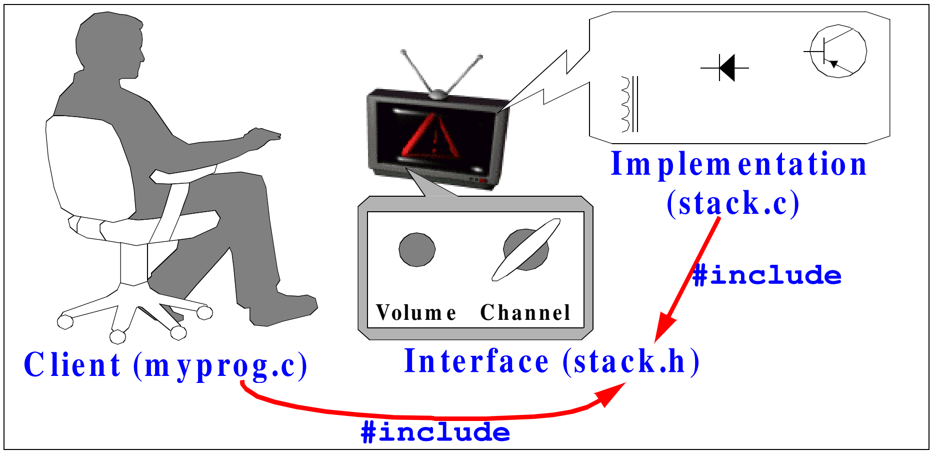
STACK ("last in, first out" or LIFO)
push: add info to the data structure
pop: remove the info most recently added
(initialize, test if empty)

QUEUE ("first in, first out" or FIFO)
put: add info to the data structure
get: remove the info LEAST recently added
(initialize, test if empty)

Could use either
array or linked list
to implement either
stack or queue

Client can work at higher level of abstraction
(stay tuned)

# Interface, Implementation, and Client




- "Client" needs to know how to use the "interface"
- "Implementation" needs to know what "interface" to implement



## Stack interface and client

**Interface (STACK.h)**

```
void STACKinit();
int STACKempty();
void STACKpush(int);
int STACKpop();
```

**Client (myprog.c)**

```
#include "STACK.h"
main()
  { int a, b;
     ...
     STACKinit();
     ...
     STACKpush(a);
     ...
     b = STACKpop();
     ...
  }
```

Client uses data type, without regard to how it is represented or implemented

## Array implementation of stack (of ints)

- Push and pop at the end of the array

```
#include <stdlib.h>
#include "STACK.h"
int s[1000];
int N;

void STACKinit()
   { N = 0; }
int STACKempty()
   { return N == 0; }
void STACKpush(int item)
   { s[N++] = item; }
int STACKpop()
   { return s[--N]; }
```

**Post-increment:**

```
s[N] = item;
N+=1;
```

**Pre-decrement:**

```
N-=1;
return s[N];
```

- Client and implementation both include STACK.h
  (could be compiled separately)
  can be compiled with one command

```
cc myprog.c stackArray.c
a.out
```

Problem: have to reserve space for max size
(see Program 4.4 in text for solution)

## Array stack example

Index of 1st empty slot

Time

| | N | t | s[0] | s[1] | s[2] | s[3] |
|---|---|---|---|---|---|---|
| | 0 | | | | | |
| push(3) | 1 | | 3 | | | |
| push(1) | 2 | | 3 | 1 | | |
| t = pop() | 1 | 1 | 3 | | | |
| push(4) | 2 | | 3 | 4 | | |
| push(t) | 3 | | 3 | 4 | 1 | |
| push(5) | 4 | | 3 | 4 | 1 | 5 |
| t = pop() | 3 | 5 | 3 | 4 | 1 | |
| t = pop() | 2 | 1 | 3 | 4 | | |
| t = pop() | 1 | 4 | 3 | | | |

index of

## Linked-list implementation of stack

- Push and pop nodes at the front of the list

```
#include <stdlib.h>
#include <STACK.h>
typedef struct STACKnode* link;
struct STACKnode { int item; link next; };
static link head;
link NEW(int item, link next)
   { link x = malloc(sizeof *x);
     x->item = item; x->next = next;
     return x;
   }
void STACKinit()
   { head = NULL; }
int STACKempty()
   { return head == NULL; }
STACKpush(int item)
   { head = NEW(item, head); }
int STACKpop()
   { int item = head->item;
     link t = head->next;
     free(head); head = t;
     return item;
   }
```

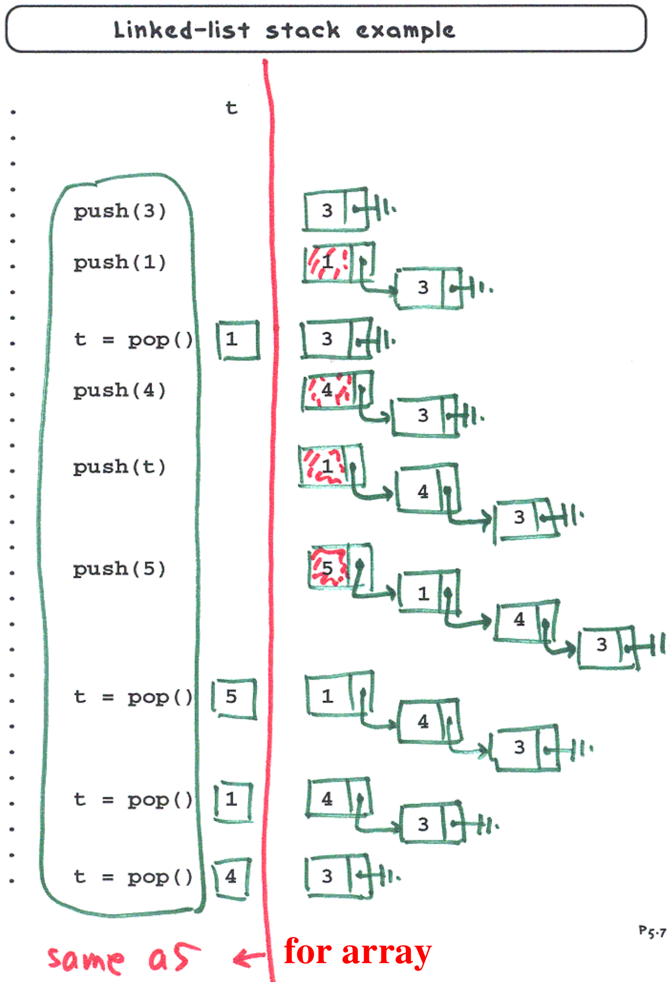
**Opposite of `malloc`: gives memory back to computer**

- Switch implementations without changing interface or client

```
cc myprog.c stackList.c
a.out
```

P5.6

# Demo Linked List Stack




# Outline

- ~~Introduction~~
- ~~Stacks (and queues)~~
- **Stack and queue applications**

## Reverse polish (postfix) notation

### Practical example of use of stack abstraction

- Put operator after operands in expression
- Use stack to evaluate
  - operand: push it on stack
  - operator: pop operands, push result
- Systematic way to save intermediate results

infix: a + b
postfix: a b +
infix: 9*16^4 + 7*16^3+5*16^2 + 3*16^1

Ex: convert 97531 from hex to decimal

```
9 16 16 16 16 * * * * 7 16 16 16 * * *
    5 16 16 * * 3 16 * 1 + + + +
9
9 16
9 16 16
9 16 16 16
9 16 16 16 16
9 16 16 256
9 16 4096
9 65536
589824
589824 7
...
589824 28672 1280 48 1
589824 28672 1280 49
589824 28672 1329
589824 30001
619825
```

22+
4

Ex: alternate implementation

```
9 16 * 7 + 16 * 5 + 16 * 3 + 16 * 1 +
```

- Stack never has more than two numbers on it!
  HORNER's METHOD (see lecture A1)



## Postfix expression evaluation in C

Classic pushdown stack client
[Sedgewick, Program 4.2]

```
#include <stdio.h>
#include <string.h>
#include "Item.h"
#include "STACK.h"
main(int argc, char *argv[])
  { char *a = argv[1]; int i, N = strlen(a);
    STACKinit(N);
    for (i = 0; i < N; i++)
      {
        if (a[i] == '+')
          STACKpush(STACKpop()+STACKpop());
        if (a[i] == '*')
          STACKpush(STACKpop()*STACKpop());
        if ((a[i] >= '0') && (a[i] <= '9'))
          STACKpush(0);
        while ((a[i] >= '0') && (a[i] <= '9'))
          STACKpush(10*STACKpop()+(a[i++]-'0'));
      }
    printf("%d ", STACKpop());
  }
```

number of inputs to main
array of input strings
number of characters in string
for each character in the string
pop two items, add them up, and push result back

deal with digits:
pop the previous digits,
multiply the number by 10,
add the new digit,
and push the partial answer back.

```
% a.out "2 2 +"
4
% a.out "12 24 +"
36
% a.out "5 9 8 + 4 6 * * 7 + * "
2075
```

# Demo Postfix Calculator



## PostScript

postfix language, abstract stack machine

Ex: convert 97531 from hex to decimal

```
9 16 mul 7 add 16 mul 5 add
      16 mul 3 add 16 mul 1 add
```

(Horner's method)

Stack:

- operands for operators
- arguments for functions
- return values for functions

Coordinate system: rotate, translate, scale, ...
Turtle commands: moveto, lineto, rmoveto, rlineto
Graphics commands: stroke, fill, ...
Arithmetic: add, sub, mul, div, ...
Stack commands: copy, exch, dup, currentpoint, ...
Control constructs: if, ifelse, while, for, ...
Define functions: /XX { ... } def

- Everyone's first program: draw a box

```
%!
50 50 translate
0 0 moveto 0 512 rlineto
512 0 rlineto 0 -512 rlineto -512 0 rlineto
stroke
showpage
```

# “First Class” ADTs

```
{
   ...
   stackInit();
   ...
   stackPush(5);
}
```

```
{
   Stack s1, s2;
   s1 = stackInit();  s2 = stackInit();
   ...
   stackPush(s1, 5);  stackPush(s2, 8);
}
```

- So far, only one stack (or queue) per program
- “First Class” ADTs
  - An ADT that is just like a built-in C type
  - Can declare multiple instances of them
  - Pass specific instances of them to the interface functions as inputs



## "WAR" using abstract data types

- Need "first class" queue ADT
  - declare variables of type queue
  - use them as arguments to functions
  - hide representation from clients
- Interface, implementation?
  ["object-oriented programming"; stay tuned]

"peace" client code, FYI
("QUEUE -> "Q" in identifiers, for brevity)

```
int play(Q deck)
   { int Aval, Bval, i, cnt = 0; Q T = Qinit();
     deal(deck);
     while  ((!(Qempty(A)) && (!Qempty(B))))
       { cnt++;
         Aval = Qget(A); Bval = Qget(B);
         if (randI(2))
              { Qput(T, Aval); Qput(T, Bval); }
         else { Qput(T, Bval); Qput(T, Aval); }
         if (Aval % 13 > Bval % 13)
           while (!Qempty(T)) Qput(A, Qget(T));
         else
           while (!Qempty(T)) Qput(B, Qget(T));
       }
     return cnt;
   }
```

Take one element from the T queue at a time, add it to the A queue, and repeat until the T queue is empty.

(Not most efficient.)

Advantage: avoid details of linked lists
Disadvantage: add details of interface

# Conclusion

- ADT is one of the most important concepts for managing software engineering complexity
- Learn to identify the possible use of ADTs in a program
- Learn the proper decomposition and encapsulation using interface and implementation files