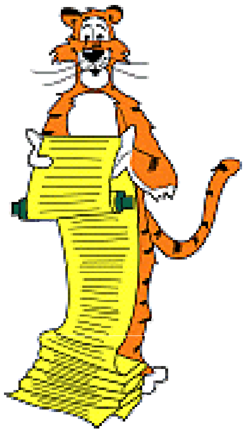
Lecture P5: Abstract Data Types



Overview

Data type:

- Set of values and collection of operations on those values.
- Ex. `short int`
  - set of values between -32,768 and 32,767
  - arithmetic operations: + - * / %

Abstract data type (ADT):

- Data type whose representation is HIDDEN.
- Don't want client to directly manipulate data type.
- Operations ONLY permitted through interface.

Separate implementation from specification.

- INTERFACE: specify the allowed operations
- IMPLEMENTATION: provide code for operations
- CLIENT: code that uses operations

Advantages of ADT's

Different clients can share the same ADT.

Can change ADT without changing clients.

Client doesn't (need to) know how implementation works.

Convenient way to organize large problems.

- Decompose into smaller problems.
- Substitute alternate solutions.
- Separation compilation.
- Build libraries.

Powerful mechanism for building layers of abstraction.

- Client works at a higher level of abstraction.

Complex Number Data Type

Create data structure to represent complex numbers.

- See Sedgewick 4.8.
- Store in rectangular form: real and imaginary parts.

```
typedef struct {
  double re;
  double im;
} Complex;
```

Complex Number Data Type: Interface

Interface lists allowable operations on complex data type.

COMPLEX.h

```
typedef struct {
  double re;
  double im;
} Complex;

Complex COMPLEXadd  (Complex, Complex);
Complex COMPLEXmult (Complex, Complex);
Complex COMPLEXpow  (Complex, int);
Complex COMPLEXconj (Complex);
double  COMPLEXmod  (Complex);
double  COMPLEXreal (Complex);
double  COMPLEXimag (Complex);
Complex COMPLEXinit (double, double);
void    COMPLEXprint(Complex);
```

interface goes in .h file

Complex Number Data Type: Client

Client program uses interface operations to calculate something:

client.c

```
#include <stdio.h>
#include “COMPLEX.h”

int main(void) {

  . . . your favorite application
  . . . that uses complex numbers

  return 0;
}
```

client can use interface

Complex Number Data Type: Implementation

Write code for interface functions.

complex.c

```
#include “COMPLEX.h”

Complex COMPLEXadd (Complex a, Complex b) {
  Complex t;
  t.re = a.re + b.re;
  t.im = a.im + b.im;
  return t;
}

Complex COMPLEXmult(Complex a, Complex b) {
  Complex t;
  t.re = a.re * b.re - a.im * b.im;
  t.im = a.re * b.im + a.im * b.re;
  return t;
}
```

implementation needs to know interface

Complex Number Data Type: Implementation

Write code for interface functions.

complex.c (cont)

```
double COMPLEXmod(Complex a) {
  return sqrt(a.re * a.re + a.im * a.im);
}

void COMPLEXprint(Complex) {
  printf(“%f + %f i\n”, a.re, a.im);
}

Complex COMPLEXinit(double x, double y) {
  Complex t;
  t.re = x;
  t.im = y;
  return t;
}
```


Separate Compilation

Client and implementation both include COMPLEX.h

Can be compiled jointly.

```
%gcc client.c complex.c
```

Can be compiled separately.

```
%gcc -c complex.c
%gcc -c client.c
%gcc client.o complex.o
```


Can Change Implementation

Can use alternate representation of complex numbers.

- Store in polar form: modulus and angle.

$$z = x + i\, y = r\,(\cos\theta + i \sin\theta) = r\, e^{i\,\theta}$$

```
typedef struct {
  double r;
  double theta;
} Complex;
```


Alternate Interface

Interface lists allowable operations on complex data type.

COMPLEX.h

```
typedef struct {
  double r;
  double theta;
} Complex;

Complex COMPLEXadd  (Complex, Complex);
Complex COMPLEXmult (Complex, Complex);
Complex COMPLEXpow  (Complex, int);
Complex COMPLEXconj (Complex);
double  COMPLEXmod  (Complex);
double  COMPLEXreal (Complex);
double  COMPLEXimag (Complex);
Complex COMPLEXinit (double, double);
void    COMPLEXprint(Complex);
```


Alternate Implementation

Write code for interface functions.

complex.c

```
#include "COMPLEX.h"

Complex COMPLEXmod(Complex a) {
  return a.r;
}

Complex COMPLEXmult(Complex a, Complex b) {
  Complex t;
  t.modulus = a.r * b.r;
  t.angle = a.theta + b.theta;
}
```

Some interface functions are now much easier to code up.

Alternate Implementation

Write code for interface functions.

complex.c

```
#include <math.h>

Complex COMPLEXadd(Complex a, Complex b) {
  Complex t;
  double x, y;
  x = a.r * cos(a.theta) + b.r * cos(b.theta);
  y = a.r * sin(a.theta) + b.r * sin(b.theta);
  t.r = sqrt(x*x + y*y);
  t.theta = arctan(y/x);
  return t;
}
```

Others are more annoying.

Multiple Implementations

Usually, there are several ways to represent and implement a data type.

Which is better: rectangular or polar representation of Complex numbers?

- Depends on application.
- Rectangular are better for additions and subtractions.
  - no need to evaluate arctangent function
- Polar are better for multiply and modulus.
  - no need to take square roots
- Get used to making tradeoffs.

This example may seem artificial.

- Essential for many real applications. . .

Rational Number Data Type

See Assignment 3.

- You will create data type for Rational numbers.
- Add associated operations (add, multiply, reduce) to Rational number data type.

```
typedef struct {
  int p;    /* numerator   */
  int q;    /* denominator */
} Rational;
```


"Non ADT's"

Is `Complex` data type an ABSTRACT data type?

- NO: Representation in interface.
  - Client can directly manipulate the data type: `a.re = 5.0;`
- Difficult to hide representation with user-defined sets of values.
- Possible to fix (see Sedgewick 4.8 or COS 217).

Are C built-in types like `int` ADT's?

- ALMOST: we generally ignore representation.
- NO: set of values depends on representation.
  - might use `(x & 1)` to test if odd
  - works only if they're stored as two's complement integers
- CONSEQUENCE: strive to write programs that function properly independent of representation.
  - `(x % 2)` is portable way to test if odd
  - also, use `<limits.h>` for machine-specific ranges of int, long

ADT's for Stacks and Queues

Prototypical data type.

- Set of operations (insert, delete) 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.

see Lecture P7

Stack Interface and Client

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

STACK.h

STACK of integers

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

```
#include "STACK.h"
int main(void) {
  int a, b;
  . . .
  STACKinit();
  STACKpush(a);
  . . .
  b = STACKpop();
  return 0;
}
```

client.c

Stack Implementation with Arrays

Push and pop at the end of array.

Demo:

Drawback:

```
#include <stdlib.h>
#include "STACK.h"

int s[1000];
int N;

void STACKinit(void) {
  N = 0;
}

int STACKempty(void) {
  return N == 0;
}

void STACKpush(int item) {
  s[N++] = item;
}

int STACKpop(void) {
  return s[--N];
}
```

stackarray.c

Compilation

Client and implementation both include STACK.h

Can be compiled jointly.

```
%gcc client.c stackarray.c
```

Can be compiled separately.

```
%gcc -c stackarray.c
%gcc -c client.c
%gcc client.o stackarray.o
```

Balanced Parentheses

parentheses.c

```
int balanced(char a[]) {
  int i;
  STACKinit();
  for (i = 0; a[i] != ’\0’; i++) {
    if (a[i] == ’(’)
      STACKpush(a[i]);
    else if (a[i] == ’)’) {
      if (STACKempty()) return 0;
      if (STACKpop() != ’(’) return 0;
    }
  }
  return STACKempty();
}
```

push all left parentheses

check for matching left parenthesis

balanced if stack is empty when no more input

Good: ((( ) ( )))
Bad: (( ))) (( )

Balanced Parentheses

parentheses.c (cont)

```
#include <stdio.h>
#include "STACK.h"
#define NMAX 1000

int main(void) {
  int c, i = 0;
  char a[NMAX];

  while ((c = getchar()) != EOF))
    if (c == '(' || c == ')’)
      a[i++] = c;
  a[i] = '\0';

  if (balanced(a)) printf("balanced\n");
  else printf("unbalanced\n");
  return 0;
}
```

Read from stdin, ignoring non-parentheses

check if balanced

Balanced Parentheses

Check if your C program has unbalanced parentheses.

Unix

```
% gcc parentheses.c stackarray.c
% a.out < myprog.c
% balanced
%
% a.out < parentheses.c
% unbalanced
```

How could valid C program have unbalanced parentheses?

Exercise: extend to handle square and curly braces.

- Good: { ([([ ]) ( )]) }
- Bad: (([ )])

ADT Review

Client can access data type ONLY through implementation.

- Example: STACK implementation.

Representation is HIDDEN in the implementation.

- Implementation uses arrays.
- Client is completely unaware of this.

Can change ADT without changing clients at all.

- In Lecture P7 we implement stack using linked lists instead of arrays.
- Obtain different time / space tradeoffs.

First Class ADT

So far, only 1 stack or queue per program.

```
STACKinit();
. . .
STACKpush(a);
. . .
b = STACKpop();
```

First Class ADT:

- ADT that is just like a built-in C type.
- Can declare multiple instances of them.
- Pass specific instances of them to interface as inputs.
- Details omitted in COS 126 - see Sedgewick 4.8 or COS 226 if interested.

```
Stack s1, s2;

s1 = STACKinit();
s2 = STACKinit();
. . .
STACKpush(s1, a);
STACKpush(s2, b);
. . .
c = STACKpop(s2);
```


Reverse Polish (Postfix) Notation

Practical example of use of stack abstraction.

Put operator after operands in expression.

- Use stack to evaluate.
  - operand: push it onto stack.
  - operator: pop operands, push result.
- Systematic way to save intermediate results.

Example 1.

- **1 2 3 4 5 * + 6 * * 7 8 9 + + * +**

Reverse Polish (Postfix) Notation

Practical example of use of stack abstraction.

Put operator after operands in expression.

- Use stack to evaluate.
  - operand: push it onto stack.
  - operator: pop operands, push result.
- Systematic way to save intermediate results.

Example 2a: convert 97531 from hex to decimal.

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

Example 2b: convert 97531 from hex to decimal.

- 9 16 * 7 + 16 * 5 + 16 * 3 + 16 * 1 +
- Stack never has more than two numbers on it!
- Horner's method (see lecture A3).