Why Data Structures?

Goal: deal with large amounts of data.
- Organize data so that it is easy to manipulate.
- Time and space efficient.

Basic computer memory abstraction.
- Indexed sequence of bits.
- Address = index.

Need higher level abstractions to bridge gap.
- Scalar (int, double, char).
- Array.
- STRUCT.
- Linked list.
- Binary tree.
- Database.
- ...

Structs

Fundamental data structure.
- HETEROGENEOUS collection of values (possibly different type).
  - Database records, complex numbers, linked list nodes, etc.
- Store values in FIELDS.
- Associate NAME with each field.
- Use struct name and field name to access value.

Built into C.
- To access rate field of structure x use x.rate
- Basis for building "user-defined types" in C.

C Representation of C Students

```c
#include <stdio.h>

struct student {
    char name[16];
    int grade;
};

int main(void) {
    struct student t;
    struct student x = {"Bill Gates", 60};
    struct student y = {"Steve Jobs", 70};
    if (x.grade > y.grade)
        t = x;
    else
        t = y;
    printf("Better student: %s\n", t.name);
    return 0;
}
```
 Typedef
User definition of type name.
- Put type descriptions in one place - makes code more portable.
- Avoid typing struct - makes code more readable.

typedef int Grade;
typedef char Name[16];
struct student {
    Name name;
    Grade grade;
};
typedef struct student Student;


A Data Type: Points
Define structures for points in the plane, and operations on them.

```c
#include <math.h>
typedef struct {
    double x;
    double y;
} Point;

double distance(Point a, Point b) {
    double dx = a.x - b.x;
    double dy = a.y - b.y;
    return sqrt(dx*dx + dy*dy);
}
Point randomPoint(double s) {
    Point p;
    p.x = randomDouble(s);
    p.y = randomDouble(s);
    return p;
}
```

Another Data Type: Circles

```c
#include <math.h>
#define PI 3.1415

typedef struct {
    Point center;
    double radius;
} Circle;

int inCircle(Point p, Circle c) {
    return distance(p, c.center) <= c.radius;
}
double area(Circle c) {
    return PI * c.radius * c.radius;
}
int intersectCircles(Circle c, Circle d) {
    return distance(c.center, d.center) <= c.radius + d.radius;
}
```

Using Data Types: Estimating π

"Monte Carlo" estimate of π.
- Generate N random points in 2 x 2 square.
- Determine fraction that lie in unit circle.
- On average π / 4 fraction should lie in circle.
- Use 4 * fraction as estimate of π.

```c
#define N 100000

int main(void) {
    int i, cnt = 0;
    Point p = {1.0, 1.0};
    Circle c;
    c.center = p; c.radius = 1.0;
    for (i = 0; i < N; i++) {
        p = randomPoint(2.0);
        if (inCircle(p, c)) cnt++;
    }
    printf("pi = %f\n", 4.0*cnt/N);
    return 0;
}
```
Standard Data Type Implementation

Data type:
- Set of values and collection of operations on those values.

Example (built-in): int, double, char.

Example (user defined): complex numbers.
- Set of values: \(4 + 2i, 1.3 - 6.7i, \text{etc.}\)
- Operations: add, multiply, show, etc.

Separate implementation from specification.
- INTERFACE: specifies allowed operations.
- IMPLEMENTATION: provides code for operations.
- CLIENT: uses data type as black box.

Complex Number Data Type

Create data structure to represent complex numbers.
- See Sedgewick 4.8.
- Store in rectangular form: real and imaginary parts.

Complex Number Data Type: Interface

Interface lists allowable operations on complex data type.
- Name interface with .h extension.

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

Complex COMPLEXadd (Complex a, Complex b);
Complex COMPLEXmult (Complex a, Complex b);
Complex COMPLEXpow (Complex a, Complex b);
Complex COMPLEXconj (Complex a);
double COMPLEXabs (Complex a);
double COMPLEXreal (Complex a);
double COMPLEXimag (Complex a);
Complex COMPLEXinit (double x, double y);
void COMPLEXshow (Complex a);
```
Complex Number Data Type: A Sample Client

Client program uses interface operations to calculate something:

```c
#include <stdio.h>
#include "COMPLEX.h"

int main(void) {
    Complex a, b, c;
    a = COMPLEXinit(5.0, 6.0);
    b = COMPLEXinit(-2.0, 3.0);
    c = COMPLEXmult(a, b);
    COMPLEXshow(a);
    printf(" * ");
    COMPLEXshow(b);
    printf(" = ");
    COMPLEXshow(c);
    printf("\n");
    return 0;
}
```

(5.0 + 6.0i) * (-2.0 + 3.0i) = -28.00 + 3.00i

Complex Number Data Type: Implementation

Write code for interface functions.

```c
#include <stdio.h>
#include <math.h>
#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;
}

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

void COMPLEXshow(Complex a) {
    printf("(%f + %f i)", a.re, a.im);
}

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

Complex Number Data Type: Implementation (cont)

function in math library

Compilation

Client and implementation both include COMPLEX.h

Compile jointly.

```bash
%gcc client.c complex.c -lm
```

Or compile separately.

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

Unix

```bash
% gcc126 client.c complex.c
% a.out
(5.00 + 6.00 i) * (-2.00 + 3.00 i) = (-28.00 + 3.00 i)
```
Client, Interface, Implementation

- Client: couchpotato.c
- Interface: "TV.h"
- Implementation: tvplasma.c

Implementation and client need to agree on interface ahead of time.

Can Change Implementation

- Can use alternate representation of complex numbers.
  - Store in polar form: modulus and angle.
  
  \[ z = x + iy = r(\cos\theta + i\sin\theta) = re^{i\theta} \]

Alternate Interface

Interface lists allowable operations on complex data type.

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

Complex COMPLEXadd (Complex a, Complex b);
Complex COMPLEXmult (Complex a, Complex b);
Complex COMPLEXpow (Complex a, Complex b);
Complex COMPLEXconj (Complex a);
double COMPLEXabs (Complex a);
double COMPLEXimag (Complex a);
Complex COMPLEXinit (double x, double y);
void COMPLEXshow (Complex a);
```

Alternate Implementation

Write code for interface functions.

```c
#include "COMPLEX.h"
#include <math.h>
#include <stdio.h>

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

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

Some interface functions are now faster and easier to code.
Alternate Implementation

Write code for interface functions.

```c
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;
}
```

Multiple Implementations

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

How to represent complex numbers: rectangular vs. polar?
- Depends on application.
- Rectangular are better for additions and subtractions.
  - no need for arctangent
- Polar are better for multiply and modulus.
  - no need for square root
- Get used to making tradeoffs.

This example may seem artificial.
- Essential for many real applications.
- Crucial software engineering principle.

Rational Number Data Type

See Assignment 3.
- You will create data type for Rational numbers.
- Add associated operations to Rational number data type.

```c
typedef struct {
    int num;
    int den;
} Rational;
```

- Simple version relatively easy to implement.
- Improved implementation staves off overflow by:
  - reducing fractions
  - order of computation

Conclusions

Basic computer memory abstraction.
- Indexed sequence of bits.
- Address = index.

Need higher level abstractions to bridge gap.
- Array.
  - homogeneous collection of values
- Struct.
  - heterogeneous collection of values

Data type.
- Set of values and collection of operations on those values.

Client-interface-implementation paradigm.
- Consistent way to implement data types.
Arrays and structs are passed to functions in very DIFFERENT ways.

**Pass-by-value:**
- int, float, char, struct
- a COPY of value is passed to function

```cpp
void mystery(Point a) {
    a.y = 17.0;
}
Point a = {1.0, 2.0};
mystery(a);
printf("%4.1f\n", a.y);
```

**"Pass-by-reference":**
- arrays
- function has direct access to array elements

```cpp
void mystery(double a[]) {
    a[1] = 17.0;
}
double a[] = {1.0, 2.0};
mystery(a);
printf("%4.1f\n", a[1]);
```