Lecture P4: Structs and Data Types

```
struct student {
  char name[20];
  int age;
  double salary;
} Hal, Bill;
```

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.

- Array.
- STRUCT.
- Linked list.
- Binary tree.
- Database.
- . . .

addr	value
0	0
1	1
2	1
3	1
4	0
5	1
6	0
7	0
8	1
9	0
10	1
. . .	. . .
256GB	1

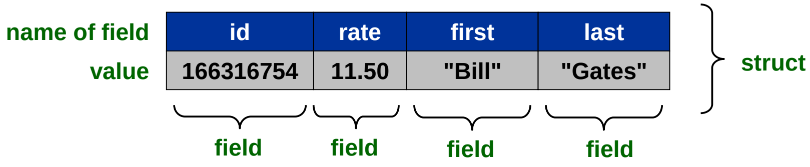
Structs

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-in to C.

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



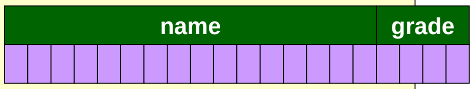
name of field	id	rate	first	last
value	166316754	11.50	"Bill"	"Gates"

C Representation of C Students

student.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;
}
```

struct declaration



can initialize struct fields in declaration

access structs as ordinary variables

%s for string

Typedef

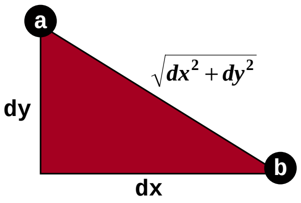
User definition of type name.

- 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;

. . .

Student x = {"Bill Gates", 60};
```

A Data Type: Points

Define structures for points in the plane, and operations on them.



random point with x and y coordinates between 0.0 and s

point data type

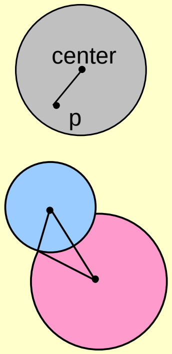
```
#include <math.h>
typedef struct {
  double x;
  double y;
} Point;

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

Define struct for circles, and operations on them.

circle data structure

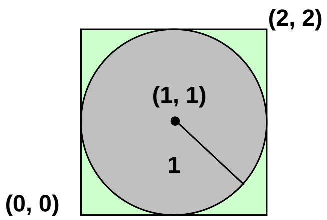
```
#include <math.h>

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 3.14 * c.radius * c.radius;
}

int intersectCircles(Circle c, Circle d) {
   return distance(c.center, d.center) <= c.radius + d.radius;
}
```



Using Data Types: Estimating Pi

Estimate pi.

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



pi.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.

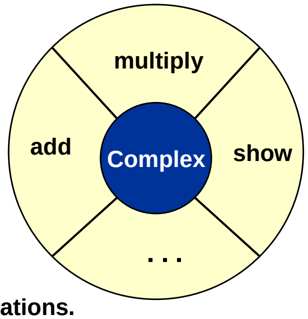
Example (built-in): int, double, char.

Example (user defined): complex numbers.

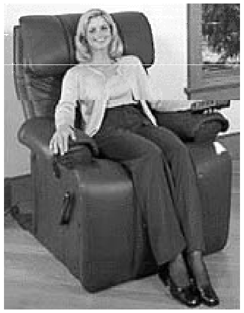
- Set of values: 4 + 2i, 1.3 - 6.7i, etc.
- Operations: add, multiply, show, etc.

Separate implementation from specification.

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



Intuition



Client

Interface

- volume
- change channel
- adjust picture
- decode NTSC, PAL signals

Implementation

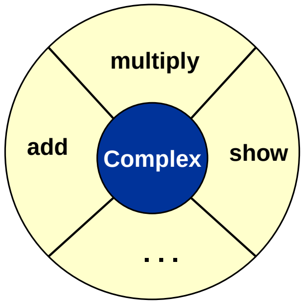
- cathode ray tube
- electron gun
- Sony Wega 36XBR250
- 241 pounds, $2,500

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.

- Name interface with .h extension.

COMPLEX.h

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

Complex COMPLEXadd   (Complex, Complex);
Complex COMPLEXmult  (Complex, Complex);
Complex COMPLEXpow   (Complex, Complex);
Complex COMPLEXconj  (Complex);
double  COMPLEXabs   (Complex);
double  COMPLEXreal  (Complex);
double  COMPLEXimag  (Complex);
Complex COMPLEXinit  (double, double);
void    COMPLEXshow  (Complex);
```

store in rectangular form

can't reuse +, * symbols

function prototypes

Complex Number Data Type: Client

Client program uses interface operations to calculate something:

client.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;
}
```

client can use interface

(5 + 6i) * (-2 + 3i) =
-28 + 3i

Complex Number Data Type: Another Client

Redo Mandelbrot with equivalent complex number update:

```
$r \leftarrow x$, $s \leftarrow y$
while ($r^2 + s^2 \le 4$)
    $r' \leftarrow r^2 - s^2 + x$
    $s' \leftarrow 2rs + y$
```

```
$c \leftarrow z = x + iy$
while ($|c| \le 2$)
    $c \leftarrow c^2 + z$
```

clientmand.c

```
#define MAXIT 255
#include "COMPLEX.h"

int mand(Complex z) {
  int i = 0;
  Complex c = z;
  while (COMPLEXabs(c) <= 2.0 && i < MAXIT) {
       c = COMPLEXadd(COMPLEXmult(c, c), z);
       i++;
  }
  return i;
}
```

Complex Number Data Type: Implementation

Write code for interface functions.

complex.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;
}
```

implementation and client need to agree on interface

Complex Number Data Type: Implementation

Write code for interface functions.

complex.c (cont)

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

void COMPLEXshow(Complex a) {
  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;
}
```

function in math library

Compilation

Client and implementation both include COMPLEX.h

Compile jointly.

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

Or compile separately.

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

Unix

```
% 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` → `#include` **Interface** `"TV.h"` ← `#include` **Implementation** `tvplasma.c`

client needs to know how to use interface

implementation needs to know what interface to implement

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.

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

```
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, Complex);
Complex COMPLEXconj  (Complex);
double  COMPLEXabs   (Complex);
double  COMPLEXreal  (Complex);
double  COMPLEXimag  (Complex);
Complex COMPLEXinit  (double, double);
void    COMPLEXshow  (Complex);
```

polar representation

Alternate Implementation

Write code for interface functions.

complexpolar.c

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

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

Complex COMPLEXmult(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.

complexpolar.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;
}
```

Others are more annoying.

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.

```
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.

Lecture P4: Supplemental Notes

Pass By Value, Pass By Reference

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

```
void mystery(Point a) {
  a.y = 17.0;
}

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

Unix

```
% a.out
 1.0
```

"Pass-by-reference":

- arrays
- function has direct access to array elements

```
void mystery(double a[]) {
  a[1] = 17.0;
}

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

Unix

```
% a.out
17.0
```