

Chapter 5: Designing Data Types: Structures

1) Steps In Creating Data Types	5-2
2) Rationale For A New Data Type.....	5-3
3) The File fraction.h.....	5-4
4) Operations On The Fraction Data Types	5-5
5) Implementation Of The Functions	5-6
6) Example Program Using Fractions	5-8
7) Applications With Fractions	5-9
8) Set Notation Examples.....	5-10
9) Creating The Set Type.....	5-11
10) Set Representation Example	5-12
11) Set Representation	5-13
12) Set Function Implementations	5-14
13) A Program Which Uses The Set Data Type	5-16

Steps In Creating Data Types

- New data types are defined using the struct keyword
- The **typedef** facility merely gives a new name for an **existing** data type
- Creating new data types is invaluable toward designing solutions that closely model the problem to be solved
- To design a new data type:
 - ▶ Create a description for the data type structure description
 - Typedef
 - ▶ Create the operations for this this data type
 - Function Prototypes
 - ▶ Write the implementation for the operations
 - Function Definitions

Rationale For A New Data Type

- Suppose we wish to create a **fraction** data type. our goal is to use fractions with the same ease as we use the built in types
- C supports fundamental types such as:
 - ▶ char int double etc,Easy to write programs using these data types
- We would like to use the same constructs for created data types as we do for the *built in* types
- For example, we would like to define, add, and multiply **fractions** just like we do **ints**
 - ▶ We can do all of this but we have to trade operators for functions
 - ▶ We can simulate the + with a function

The File `fraction.h`

- The file `fraction.h` would consist of the pieces shown here
- Create it: (i.e.) decide on a representation

```
struct fraction {  
    int n;  
    int d;  
};
```

Note: other representations are possible.

- Create new name(s) for the new type

```
typedef struct fraction FRACTION, *FPTR;
```

- Specify the operations for the new type

```
FRACTION create(int numerator, int denominator);  
FRACTION input(void);  
void print(FPTR);  
int gcd(int first_dividend, int second_dividend);  
FRACTION add(FPTR, FPTR);  
FRACTION mult(FPTR, FPTR);  
FRACTION divide(FPTR, FPTR);  
FRACTION subtract(FPTR, FPTR);  
etc.
```

Operations On The Fraction Data Types

- `int a,b,c;` // DEFINE INTs
`FRACTION x,y,z;` // FRACTIONS
- `int fun();` // FUNCTION RETURNS INT
`FRACTION fun2();` // RETURNING FRACTION
- `int *p1;` // POINTER TO INT
`FRACTION *p2;` // TO FRACTION
- `a = b + c;` // ADD INTEGERS
`x = y + z;` // ADD FRACTIONS

Implementation Of The Functions

- Each of the functions should now be implemented
 - ▶ A few are shown below. Each requires the inclusion of `fraction.h`

```
#include "fraction.h"
FRACTION input()
{
    FRACTION p;
    printf("input num then denom (n/d) ");
    // "%d/%d" format => allows input of the
    // of the form n/d

    scanf("%d/%d", &p.n, &p.d);
    while( getchar() != '\n')
        ; // flush the newline away
    return(p);
}
```

```
#include "fraction.h"
FRACTION create(int numerator, int denominator)
{
    FRACTION p;
    p.n = numerator;
    p.d = denominator;
    return(p);
}
```

Implementation Of The Functions

```
#include <assert.h>
#include "fraction.h"
void print(FPTR p)
{
    int temp;
    assert(p -> d != 0 );// div by 0
    temp = gcd(p->n,p->d);// reduce
    assert(temp != 0);// sanity check
    p -> n = p -> n /temp;
    p -> d = p -> d /temp;
    if ( 1 == p -> d)// easy reading
        printf("%d\n", p -> n);
    else if ( 0 == p -> n) // easy reading
        printf("0\n");
    else
        printf("%d/%d\n", p -> n, p -> d);
}

#include <fraction.h>
FRACTION add(FPTR f, FPTR s)
{
    FRACTION p;
    p.n = f -> n * s -> d + f -> d * s -> n;
    p.d = f -> d * s -> d;
    return(p);
}
```

Example Program Using Fractions

- An entire program using the **fraction** type
 - ▶ Fill an array and sum **max** many fractions

```
#include "fraction.h"
#define MAX 5
main()
{
    int i;
    FRACTION array[MAX];
    FRACTION s ;
    s = create(0,1);
    printf("enter %d fractions\n", MAX);
    for (i = 0; i < MAX; i++) {
        printf("input fraction # %d ", i + 1);
        array[i] = input();
    }
    for ( i = 0; i < MAX; i++)
        s = add(&s,&array[i]);
    print(&s);
}
```

Applications With Fractions

- Any applications with fractions would include the header file **fraction.h**. Fractions could then be:

- ▶ Defined with

- `FRACTION a,b,c;`

- ▶ Initialized with

- `a = create(2,3);`
- `b = create(4,5);`

- ▶ Added with

- `c = add(&a, &b);`

- Complex calculations could proceed as

```
c = (a + b) * (a / b);
```

```
FRACTION a, b, c;
```

```
c = mult(add(&a, &b), divide(&a, &b));
```

NOTE THAT **add** MUST RETURN AN **FPTR** RATHER THAN A **FRACTION** TO SATISFY **mults** ARGUMENTS

Set Notation Examples

- Mathematicians use the following set notation

$$A = \{0, 3, 5, 2, 4\}$$

$$B = \{3, 4, 6, 1\}$$

- Set intersection

 \wedge

$$A \wedge B = \{3, 4\}$$

- Set union

 \cup

$$A \cup B = \{0, 1, 2, 3, 4, 5, 6\}$$

- Set difference

 $-$

$$A - B = \{0, 5, 2\}$$

- Compliment of a set

With respect to (say the set of digits)

$$\sim A = \{1, 6, 7, 8, 9\}$$

Creating The Set Type

- Sets are collections of unordered objects
- Many operations are defined on sets including

Intersection: those elements in **A** and in **B**

Union: those elements in **A** or in **B**

Difference: those elements in **A** and not in **B**

Complement of a set: those not in the set

Complement of a set must be with respect to a universal set (say the set of all digits)

Set Representation Example

- The following pieces would be placed in **set.h**
- Representation

```
#define SIZE 100
struct set {
    int array[SIZE];
    int howmany;
};
```

- Create the new names for the type
`typedef struct set SET, *SETP;`
- Create operations for the new type

```
SETP create(void);
void add(int, SETP);
void print(SETP);
SETP setunion(SETP, SETP);
```

Set Representation

- There are many representations of sets
 - ▶ Abstract sets have no bound
 - ▶ Good candidate for dynamic representation
- Instead we choose the following
 - ▶ An array
 - ▶ A particular size
- Following the method of the **fraction** data type, we use **typedef** to create some new names
- Next we choose the operations to be implemented

Set Function Implementations

```
#include <stdlib.h>
#include "set.h"
SETP create()
{
    SETP temp;
    temp = (SETP) malloc(sizeof(SET));
    if(temp == NULL) {
        printf("malloc: no more room\n");
        exit(1);
    }
    temp -> howmany = 0;
    return(temp);
}
void add(int new, SETP p)
{
    if( p -> howmany == SIZE ) {
        printf("set overflow\n");
        exit(2);
    }
    p -> array[p -> howmany++] = new;
}
void print(SETP p)
{
    int i;
    for ( i = 0; i < p -> howmany; i++)
        printf("%d\n", p -> array[i]);
}
```

Set Function Implementations

```
SETP setunion(SETP a, SETP b)
{
    SETP c;
    int i, j;
    c = create();
    for ( i = 0; i < a -> howmany; i++)
        add(a -> array[i], c);
    for ( i = 0; i < b -> howmany; i++) {
        for ( j = 0; j < a -> howmany; j++)
            if(b -> array[j] == a -> array[i])
                break;
        if ( j == a -> howmany)
            add(b -> array[i], c);
    }
    return(c);
}
```

A Program Which Uses The Set Data Type

```
#include "set.h"
main()
{
    SETP c,a,b;
    a = create();
    b = create();
    add(10,b);           // ADD ELEMENT 10 TO SET b
    add(5,b);
    add(5,a);           // ADD ELEMENT 5 TO SET a
    add(6,a);
    print(a);
    print(b);
    c = setunion(a,b); // TAKE UNION OF a AND b
    print(c);
}
```

Exercises For Chapter 5

1. Implement the fraction **add** function with the following function prototype.

```
FPTR add(FPTR, FPTR);
```

2. Create the type **COMPLEX** and provide operations such as **init**, **add**, **mult**, and **print**.
3. Implement **set** intersection.

This Page Intentionally Left Blank

Evaluation