User Defined Data: Product Constructor

Built-in Data Types

Built-in data types of C language are `int`, `float`, `unsigned int`, `char`, `double` etc. The representations and operations corresponding to these data types are decided by the language or by the compiler.

Defined Types

It is essential to define new types of data (object) and perform operations on them in programming related to different applications. A programming language cannot have everything as built-in, but it should provide facility (constructor) to define a new data type, declare an objects of the defined type, and facilities to support operations on them.

A complex Example

A complex number (or its approximation) is not a built-in datatype of C language[a].

[a] Not really - C99 has a data type `_Complex, complex`. See the man page of `complex.h`.

C99 complex in GCC

```
#include <stdio.h>
#include <complex.h>
int main() // gComplex1.c
{
    complex double x, y ;
    x = 1.0 + 2.0i; // real + imaginary part
                    // Operator overloading
    y = ~x;         // complex conjugate
    printf("x: %lf+j%lf\n",creal(x),cimag(x));
    printf("y: %lf - j%lf\n",creal(y),-cimag(y));
    return 0;
}
```

Note

But we shall assume that the data type `complex` is not supported in C language and will define and support as a user defined data.

Data in a Complex Number

In school mathematics book a complex number z is written as $z = a + jb$, where a is the real part and b is the imaginary part. The value of $j = \sqrt{-1}$.

Data in a Complex Number

But the actual data in z may be viewed as an ordered pair of real numbers i.e. $z = (a, b)$, where the first component is the real part and the second component is the imaginary part. The collection of complex numbers is the Cartesian Product of reals i.e. $\mathbb{C} = \mathbb{R} \times \mathbb{R}$.

Note

Viewing $\mathbb{C}$ as $\mathbb{R} \times \mathbb{R}$ is not enough. The data type $\mathbb{C}$ is an algebraic structure equipped with a set of operations and relation e.g. addition, subtraction, equality etc. But right now we are mainly interested about the representation.

Approximation for Representations

The first question is how to represent a complex number in a C program. We have already identified a complex number with a pair of reals. But a real number cannot have exact representation in a computer. It is approximated as a floating-point number (`float` or `double`). So a complex number may be approximated as an ordered pair of floating-point numbers.

Representation: Choice I

We may take two variables of type `double` (or `float`), one holds the real part and the other one holds the imaginary part. We can perform operations on such pairs. But following are the problems of such a representation.

Problems of Choice I

- There is no structural glue between these two variables to call them as a single object.
- It is not possible to pass them as a single parameter, return them as a single value.
- Cannot be assigned by a single assignment or compare them directly.

Representation: Choice II

We may take a two element array of type `double` (or `float`), where the 0^{th} element holds the real part and the 1^{st} element holds the imaginary part. Unfortunately in C a whole array cannot be returned as a value or assigned. But otherwise this is not a bad choice as long as the components of the product are of same type.

Product Constructor in C

C language provides a type constructor for product called a structure. A structure may have data of different types with a tag/name for each component. A structure corresponding to the type *complex* is

```
struct complexType {
        double real, imag ;
};
```

`struct complexType` is a new data type

Note

A data type is a plan for a data object. The defined data type `struct`[a] `complexType` has two components or members each of type `double`.

[a] '`struct`' is a reserve word like `if`, `while`, `int`, `return` etc. They have specific meaning in a language and cannot be used as name of an object.

```
struct complexType {
        double real, imag
}
```

data type

Note

We can declare a variable, an array, a pointer variable of the defined type. A variable can be initialized.
Assignment operator can be used for this type, the address of a variable of this type can be extracted using &. A pointer variable can be dereferenced using *.
A function can take it as a parameter and also return a value of this type.
sizeof() extracts the size of a defined type.

A Good Name to Data Type

A new name can be given to a data type using `typedef` e.g. `typedef int integer` creates the new name `integer` to the data type `int`. We use `typedef` to give a better and shorter name to `struct complexType`.

Type: complex

```
struct complexType {
        double real, imag ;
};
typedef struct complexType complex;
```

or we may have

```
typedef struct {
        double real, imag ;
} complex;
```

Operations

```
complex a = {1.0,2.0}, b, c[5], *p ;
```

The variable `a` of type complex is initialized with `1.0` as the value of the `real` component and `2.0` as the value of the `imag` component.

Operations

```
complex a = {1.0,2.0}, b, c[5], *p ;
```

The variable `b` of type `complex` is uninitialized. `c[]` is an 1-D array of five element of type `complex`. The variable `p` is of type `complex *` (pointer), can hold an address and its pointer arithmetic is determined by the `sizeof(complex)`.

Operations

```
complex a = {1.0,2.0}, b, c[5], *p ;
```

A field of a structure can be accessed by the '.' (projection) operator. We may write,

```
b.real = 3.0; b.imag = 4.0 ;
a.real = 2.0*b.real + 3.0/b.imag ;
```

etc.

Operations

```
complex a = {1.0,2.0}, b, c[5], *p ;
```

We can write `p = &b` to make `p` point to the object `b`.

We can also use `malloc()` to allocate space for type `complex`.

```
p = (complex *)malloc(sizeof(complex));
```

The returned generic pointer can be casted to `complex *` and assigned.

Operations

```
complex a = {1.0,2.0}, b, c[5], *p ;
p = &b; (*p).real = 5.0 ;
```

The pointer variable `p` is pointing to a `complex` object (`b`), so `*p` is the object (`b`) and we get its `real` component (`b.real`) using the '`.`' operator. But then '`.`' has higher precedence than '`*`', so a parenthesis is essential. A short-hand to access the component of a structure through a pointer is the operator '`->`'.

```
p -> real = 5.0 ;
```

Operations

```
complex a = {1.0,2.0}, b, c[5], *p ;
c[3].real = 7.0, c[0].imag = 8.0;
```

Each element of the array is of type `complex` and a component corresponding to an array element can be accessed using '`[]`' and '`.`' operators.

We can also make `p = c+2;`, so that `p` points to the second element of the array.

Operations

```
complex addComplex(complex x, complex y){
        complex t ;
        t.real = x.real + y.real ;
        t.imag = x.imag + y.imag;
        return t;
}
```

A structure can be passed as a parameter and can be returned as a value.

```
c[1] = addComplex(a,b);
```

C Program

```
#include <stdio.h>
#include <stdlib.h>
typedef struct {
        double real, imag ;
} complex ;
complex addComplex(complex, complex);
void printComplex(complex);
int main() // dataComplex1.c
{
    complex a = {1.0, 2.0}, b, c[5], *p ;

    b.real = 3.0, b.imag = 4.0 ;
```

```
p = (complex *)malloc(sizeof(complex));
(*p).real = 5.0 ;
p -> imag = 6.0 ;
c[0].real = 7.0, c[0].imag = 8.0;
c[1] = addComplex(a,b);
c[2] = addComplex(*p, *p);
printComplex(c[0]);
putchar(' ');
printComplex(c[1]);
putchar('\n');
p = &c[2] ;
printComplex(*p);
putchar('\n');
return 0;
```

```
}
complex addComplex(complex x, complex y){
        complex t ;

        t.real = x.real + y.real ;
        t.imag = x.imag + y.imag;
        return t;
}
void printComplex(complex x){
     printf("%f + j%f", x.real, x.imag);
} // dataComplex1.c
```

Output

```
$ cc -Wall dataComplex1.c
$ ./a.out
7.000000 + j8.000000 4.000000 +
j6.000000
10.000000 + j12.000000
```

Student's Data

Often data types are not standard mathematical entity. We consider data related to every student of a college (say IIT). There may be large number of data items, but for simplicity we only consider the name, roll number, sgpa and grade points in different semesters and the cgpa upto the last semester. The corresponding product data type or the structure may look like the following:

Structure student

```
#define NAME 50
#define ROLL 9
typedef struct {
        char name[NAME], roll[ROLL] ;
        int sem ;
        struct {
          float sgpa ;
          int gp;
        } sgpa[10];
        float cgpa ;
} student ;
```

Structure student

```
#include <stdio.h>
#include <stdlib.h>
#define NAME 50
#define ROLL 9
typedef struct {
        char name[NAME], roll[ROLL] ;
        int sem ;
        struct {
                float sgpa ;
                int gp;
        } sgpa[10];
        float cgpa ;
```

```
} student ;
student *readStudent(int *);
void calcCgpa(student *, int);
void insertionSort(student *, int);
void writeStudent(student *, int);
int main() // dataStudent.c
{
    int n;
    student *sP ;
    sP = readStudent(&n);
//    printf("Raw data:\n");
//    writeStudent(sP, n);
    calcCgpa(sP, n);
    insertionSort(sP, n);
```

```
    printf("Merit List:\n");
    writeStudent(sP, n);
    return 0;
}
student *readStudent(int *nP){
        int i ;
        student *sP ;
        scanf("%d", nP);
        sP = (student *)malloc(*nP*sizeof(student));
        for(i=0; i<*nP; ++i){
            int m, j;
            scanf("%s %[^0-9]%d",
               sP[i].roll, sP[i].name, &sP[i].sem);
            m = sP[i].sem - 1;
```

```
            for(j=0; j<m; ++j)
                scanf("%d%f", &sP[i].sgpa[j].gp,
                              &sP[i].sgpa[j].sgpa);
        }
        return sP;
} // dataStudent.c
void calcCgpa(student *sP, int n){
     int i, j, gpSum;
     float sgpaSum;

     for(i=0; i<n; ++i){
        gpSum = 0, sgpaSum = 0.0;
        for(j=1; j<sP[i].sem; ++j){
            gpSum += sP[i].sgpa[j-1].gp;
```

```
            sgpaSum += sP[i].sgpa[j-1].gp*sP[i].sgpa[j-1].s
        }
        sP[i].cgpa = sgpaSum/gpSum;
    }
} // dataStudent.c
void insertionSort(student *sP, int n){
    int i, j ;
    student temp ;

    for(i=0; i<n; ++i){
        temp = sP[i] ;
        for(j=i-1; j>=0; --j)
            if(sP[j].cgpa < temp.cgpa) sP[j+1] = sP[j];
            else break ;
```

```
        sP[j+1] = temp;
    }
} // dataStudent.c
void writeStudent(student *sP, int n){
    int i, j;

    for(i=0; i<n; ++i){
        printf("%s %s\n", sP[i].roll, sP[i].name);
        for(j=1; j<sP[i].sem; ++j)
            printf("\tSem %d sgpa: %4.2f\n", j, sP[i].sgpa[
        printf("\t\tcgpa: %4.2f\n", sP[i].cgpa);
    }
} // dataStudent.c
```

Input Data: stdData

```
7
08CE1012 Sukesh Jain 3 20 8.5 25 8.0
08CS1020 Ansuman Roy 3 22 8.0 23 7.5
08PH1010 Rusha Ahamed 3 25 9.0 20 8.5
08NA1002 Karan Rao 3 20 8.0 25 7.5
08EE1007 Simranjit S Mann 3 23 8.5 22 8.5
08EC1023 P V Verma 3 20 8.0 25 9.0
08CH1016 P V Verma 3 20 8.0 25 9.0
```

Variation of Structure student

```
#define NAME 50
#define ROLL 9
struct sgpa {
        float sgpa;
        int gp;
};
typedef struct {
        char name[NAME], roll[ROLL] ;
        int sem ;
        struct sgpa sgpa[10];
        float cgpa ;
} student, Student ;
```

Note

The type name `struct sgpa` does not clash with the field name `sgpa[10]`.

More than one type name can be given using `typedef`.

If a variable is declared with a type name defined by `typedef`, the defined type name will be obscured in the scope of the variable.

Name Clash

```
#include <stdio.h>
typedef struct complexType {
                double real, imag;
} complex ;
int main() // typedef1.c
{
    int complex; // Obscures type complex in main()
//    complex a = {1.0, 2.0}, b = {2.0, 3.0};

    return 0;
}
complex addComplex(complex x, complex y){ // complex is typ
```

```
        complex t;
        t.real = x.real + y.real ;
        t.imag = x.imag + y.imag ;
        return t;
}
```