Ciencias computacionales

Propedéutico: Programación

Contents

1 Tipos de datos compuestos

1.1 Vídeos sobre tipos de datos compuestos

Vídeo sobre arreglos en C (Parte 1).
Vídeo sobre arreglos en C (Parte 2).
Vídeo sobre unión

1.2 Arreglos

Los arreglos son muy utilizados en C, debido a que no se cuenta con una variable *String* como en Java, en C se almacenan las cadenas de texto carácter por carácter dentro de una posición contigua a otra en memoria.

Los arreglos son un caso especial de estructuras de datos, aun conteniendo un número limitado o fijo de elementos, su tamaño y tipo de elementos son del mismo tipo, es decir, un arreglos es una estructura de datos homogéneos. Los arreglos se enumeran consecutivamente de $0, 1, 2, \ldots, n$, es decir, tienen un indicie por el cual se puede acceder a su información. Por ejemplo, si tenemos un arreglo llamado a y queremos acceder a su primer elemento, se tiene que acceder al indice 0, es decir, *a[0]*. En C, al compilar no se marca error si el indice del arreglo no esta en un rango definido, sin embargo causaría un fallo de programa a la hora de ejecución.

Para utilizar a los arreglos es necesario asignar valores a sus elementos, esto se logra mediante la asignación directa o de manera constante. Por ejemplo, la manera directa de iniciar el arreglo es *a[0] = 10 ;*. De manera constante, se hace de la siguiente manera: *int a[5] = {10, 20, 20, 40, 50}*.

El problema al utilizar arreglos para almacenar variables de distinto tipo para una sola entidad, por ejemplo, el registro de un alumno, el cual necesita distintos campos y variables consiste en la dificultad manejar varios arreglos paralelos, lo cual se convierte en una tarea tediosa [1].

1.2.1 Arreglos multidimensionales

Al contrario a los arreglos unidimensionales (una sola dimensión) los arreglos multidimensionales tienen más de una dimensión y por consiguiente más de un indice. Estos arreglos son comúnmente utilizados para representar matrices o tablas. De manera lógica se representa un arreglo de dos dimensiones de la siguiente manera:

Table 1: Arreglo de 3 x 2

3	1
5	2
8	7

Donde el numero tres representa el número de filas o líneas, y el número dos representa el número de columnas. Digamos que el arreglo tiene como nombre *a*, entonces si nos posicionamos en *a[0]* estaríamos en la fila que contiene a los elementos *(3, 1)*. La fila *a[1]* contendría a los elementos *(5,2)* y así sucesivamente.

Recordemos que en C, la indexación de lleva desde el número *0* hasta *n*. Así, por ejemplo si queremos saber el valor del arreglo en la posición *a[0][0]*, nos devolvería el valor de 3. A continuación se muestra un programa que se encarga de imprimir el valor de la matriz mostrada en la Tabla 1, de igual manera se muestra la misma funcionalidad utilizando apuntadores [2].

```
1 #include <stdio.h>
2 void printarr(int a[][]);
3 void printdetail(int a[][]);
4 void print_usingptr(int a[][]);
5
6 main(){
7         int a[3][2];
8         for(int i = 0;i<3;i++)
9                 for(int j=0;j<2 ;j++){
```

```
10
11                  {
12                  a[i]=i;
13                  }
14          }
15          printdetail(a);
16  }
17
18  void printarr(int a[][]){
19          for(int i = 0;i<3;i++)
20          for(int j=0;j<2;j++){
21          {
22                  printf("value␣in␣array␣%d\n",a[i][j]);
23          }
24          }
25  }
26
27  void printdetail(int a[][]){
28          for(int i = 0;i<3;i++)
29          for(int j=0;j<2;j++){
30          {
31                  printf("value␣in␣array␣%d␣and␣address␣is␣%8u\n", a[i][j
                        ],&a[i][j]) ;
32          }
33          }
34  }
35
36  void print_usingptr(int a[][]){
37
38          int *b;
39          b=a;
40          for(int i = 0;i<6;i++)
41          {
42                  printf("value␣in␣array␣%d␣and␣address␣is␣%16lu\n",*b,b);
43                  b++; // increase by 2 bytes
44          }
45  }
```

1.3 Estructuras

Las estructuras en C permiten crear nuevos tipos de datos, esta capacidad permite crear tipos de datos complejos no proporcionados por el lenguaje, muchas veces se agrupan distintas variables siendo de diferentes tipos. Las estructuras son utilizadas cuando la aplicaciones requieren grandes cantidades de datos, por ejemplo las bases de datos u otras aplicaciones.

A los componentes individuales contenidos en una estructura se les conoce como *miembros*, cada uno de los cuales puede alojar datos diferentes. Como ejemplo, se puede usar una estructura de datos para almacenar información acerca de una persona, como lo es nombre, apellido, edad y fecha de nacimiento.

En resumen una estructura contiene n número de elementos, de los cuales cada uno tiene asociado un nombre único.

1.3.1 Declaración en C de estructuras

Las estructuras deben de ser declaradas por el usuario antes de poder ser utilizadas, el formato puede ser el siguiente:

```
1  struct<struct_name>{
```

```
        <data_type_{1}> <members_name_{1}>
        <data_type_{2}> <members_name_{2}>
        ...
        <data_type_{n}> <members_name_{n}>
};
```

Por ejemplo, queremos representar un número complejo con una estructura, entonces necesitamos la parte real y la parte imaginaria.

```
struct complex_number{
        float real_part, imaginary_part;
};
```

En el ejemplo anterior, el tipo de dato es el mismo. Ahora veamos como podemos representar por ejemplo información de una persona, como puede ser el nombre, estado civil, edad y fecha de nacimiento, claramente debe contener distintos tipos de datos.

```
struct person_info{
        char name [32] ;
        char marital_status [16] ;
        unsigned int age;
        unsigned int birthday;
};
```

Como utilizar las estructuras

Declaremos una estructura para almacenar fechas.

```
struct date{
        int day;
        int month;
        int year;
};
```

Ahora podemos utilizarla declarando una variable de tipo estructura de *date* de la siguiente manera [1]:

```
struct date dob ; // to hold a "date of birth"
```

Para poder utilizar la estructura, nos podemos referir al campo *día* de la siguiente manera *dob.day*, el campo *mes* como *dob.month* y al *año* como *dob.year*. Se puede ver que se accede mediante el uso del punto (.) a los miembros de la estructura.

Para llenar el contenido de la estructura a cada campo se le asigna un valor. Por ejemplo para asignar la fecha "October 24, 2015" se hace de la siguiente manera:

```
        dob.day = 24 ;
        dob.month = 10 ;
        dob.year = 2015;
```

También podemos decirle al usuario que introduzca directamente los valores, de la siguiente manera:

```
        scanf ("%d␣%d␣%d", &dob.day, &dob.month & dob.year) ;
```

Las estructuras se pueden declarar en cualquier parte del programa, o bien se pueden ser parte de la definición, en este caso es necesario dar un valor inicial a cada una de las variables o miembros de la estructura.

Otra de las formas mas comunes para acceder a los miembros de una estructura es mediante el uso del apuntador ($->$). Primero se define una variable de tipo apuntador hacia la estructura que se desea acceder, después se puede usar el operador del apuntador para acceder a cada miembro de la estructura. A continuación se muestra un ejemplo.

```
        struct student{
                char Name [32] ;
                int student_id;
                float grade;
        };
```

Se define un apuntador a la estructura anterior de la siguiente manera:

```
        struct student *ptr_std;
        struct student best;
```

La asignación de valores se hace de la siguiente manera:

```
ptr_std = & best ;
strcpy (ptr_std -> Nombre, "Jonh Smith" ) ;
ptr_std -> student_id = 3215 ;
ptr_std -> grade = 8.5 ;
```

Al igual que cualquier otra variable en C, para conocer el tamaño en memoria que ocupa una estructura se puede hacer mediante el uso de la palabra reservada *sizeof*. Tomemos la siguiente estructura:

```
#include <stdio.h>
struct person{
        char name[32] ;
        int age ;
        float height;
        float weight;
}

void main (){
        struct person p;
        printf ("Sizeof(person): %d \n" , sizeof(p)) ;
}
```

Al momento de ejecutar el programa anterior obtendremos a la salida: *Sizeof(perosn): 42*. Esto se debe a que a la variable *name[32]* se le asignan 32 bytes, a la variable *age* se le asignan 2 bytes y a las variables *height* y *weight* se les asigna 4 bytes y 4 bytes respectivamente.

1.4 Estructuras de estructuras

También se pueden tener casos en los cuales se necesitan tener estructuras de estructuras. Estas son útiles porque se pueden utilizar estructuras similares. La idea es definir miembros comunes unicamente en una estructura y utilizar a dicha estructura como miembro de otra estructura.

Por ejemplo, podemos en el cual se comparten los miembros del domicilio en dos estructuras que tienen distinto fin:

```
struct employee{
        char name_employee[32] ;
        char address [32] ;
        char city [16] ;
        long int zipcode;
        double salary;
};

struct client{
char name_client[32] ;
char address [32] ;
char city [16] ;
long int zipcode;
double balance;
```

```
15 }
```

La forma de reducir o reutilizar las variables comunes de las anteriores estructuras, es crear una que contenga los miembros que tienen en común. Entonces por ejemplo:

```
1 struct info{
2         char address [32] ;
3         char city [16] ;
4         long int zipcode;
5 }
```

La estructura anterior puede ser utilizada dentro de otra estructura:

```
1 struct employee{
2         char name_employee[32] ;
3         struct info address_employee;
4         double salary;
5 };
```

```
1 struct client{
2         char name_client[32] ;
3         struct info address_client;
4         double balance;
5 }
```

1.5 Uniones

Las uniones al igual que las estructuras agrupan varias variables. La diferencia radica en la forma de almacenamiento. Al usar estructuras estas almacenan datos de manera continua en memoria. Al igual que las estructuras, las uniones también cuentan con una palabra reservada la cual es *union*. De manera similar se almacenan distintos tipos de elementos, pero estos se enciman o solapan entre sí, es decir, en memoria ocupan comparten la misma posición.

El tamaño de la unión esta determinado por el tamaño de la variable más grande. Es decir, si tenemos en una unión un *int* y un *float*, el tamaño de la unión correspondería al tamaño de la variable *float*.

La declaración de las uniones es similar a aquella de la estructura. Se puede iniciar una unión de la siguiente manera:

```
1 union name{
2         data_type member1 ;
3         data_type member2 ;
4         ...
5 };
```

Las uniones son comúnmente utilizadas cuando es necesario ahorrar memoria. Por ejemplo, si tenemos tres cadenas de caracteres :

```
1 char c1[100] ;
2 char c2[100] ;
3 char c3[100] ;
```

En realidad estas tres variables ocuparían 300 *bytes* en memoria. Supongamos que no es necesario utilizarlas en todo momento, es decir, no se usan de manera simultanea. Entonces, podemos utilizar una unión, esto nos ocupa un total de 100 *bytes* en memoria. Por ejemplo:

```
1 union mychars{
2         char c1[100] ;
3         char c2[100] ;
4         char c3[100] ;
5 } strings ;
```

References

[1] Noel Kalicharan. *Advanced Topics in C: Core Concepts in Data Structures*. Apress, Berkely, CA, USA, 1st edition, 2013.

[2] PS Deshpande and OG Kakde. *C & Data Structures*. Charles River Media, 2003.