Tipos definidos por el programador

Los programadores pueden definir sus propios tipos de datos, de acuerdo a la necesidad que tengan en sus programas, basándose en agrupaciones de los tipos ya existentes. El lenguaje C ya proveía de la posibilidad de crear lo que se dio en llamar estructuras y uniones. Posteriormente, C++ proveyó de lo que en programación orientada a objetos se conoce como clase.

Definición de tipos con *struct*

Iniciemos comparando las estructuras con lo que ya conocemos, los arreglos. Las estructuras permiten también agrupar diferentes elementos en un solo conjunto. Pero a diferencia de los arreglos, posibilitan que cada uno de sus elementos sea del tipo que nosotros queramos. Si necesitamos un recurso que nos permita agrupar elementos y que estos sean de distinto tipo, entonces utilizamos la estructura.

Debido a esta característica es que algunos consideran que las estructuras son los precursores de las clases, que son una especie de recurso para definir tipos de objetos, en el paradigma de programación conocido como programación orientada a objetos, aunque con menos potencia que las clases.

Para definir una estructura se utiliza el comando *struct* y se utiliza la siguiente sintaxis:

```
struct nombre{ … };
```

Dentro de las llaves se colocan "variables" de los tipos fundamentales u otros tipos que se hayan definido previamente. Las variables dentro de una estructura reciben el nombre de **campos**.

Ejemplos de dos estructuras podrían ser los siguientes:

```
struct coordenadas{
    int x;
    int y;
};
```

```
struct descripcion{
    int anio;
    double precio;
    bool suspendido;
};
```

En este punto es buen recordar que, de todos los recursos de C/C++ que vamos aprendiendo, ninguno excluye a otro; es decir, incluso podemos definir arreglos cuyas casillas son estructuradas, o estructuras que en su interior tienen arreglos.

Las estructuras se utilizan como tipos de datos para definir variables de esos tipos más adelante en el programa. Por ejemplo, el tipo estructurado *coordenadas* puede utilizarse para declarar una o más variables en C++, por ejemplo:

```
coordenadas punto;
```

El tipo estructurado descripción puede utilizarse para declarar una o más variables en C++, por ejemplo:

```
descripcion algo;
```

Note que la declaración siempre se escribe igual que como ya conocemos: primero el tipo y a continuación una o más variables separadas por coma.

Si se declararan esas mismas dos variables en C, hubiera tenido que escribirse de esta manera:

```
struct coordenadas punto;

struct descripcion algo;
```

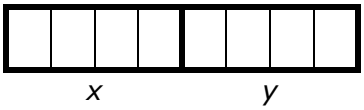
Las variables *punto* y *algo son* variables estructuradas, cada una de un tipo definido por el programador. Puede accederse a cada uno de los campos de ellas, con el formato de acceso a miembro, o formato de acceso a campo, así:

punto.x accede al contenido almacenado en el campo *x* dentro de la variable estructurada *punto*.

algo.precio accede al contenido almacenado en el campo *precio* dentro de la variable estructurada *algo*.

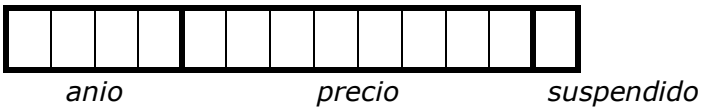
Las variables *punto* y *algo* utilizan la cantidad de bytes que suman todos los campos que se definen dentro de cada una de sus estructuras. La variable *punto* utilizaría ocho bytes en total y la podemos visualizar así en la memoria de la computadora:

punto:



La variable *algo* utilizaría un total de 13 bytes y la podemos visualizar así en la memoria de la computadora:

algo:



Podemos ver a las variables estructuradas como que contienen diferentes casillas en su interior y cada casilla también tienen un nombre. Para acceder a las casillas específicas se debe utilizar el formato de acceso a miembro ya que esa casilla está "dentro de" una variable del tipo estructurado. Si una variable dentro de la estructura también es de un tipo estructurado, se utiliza el formato de acceso a miembro para internarse en la subestructura hasta llegar al dato de interés, así:

variable estructurada.campo estructurado. … .dato;

Si es un puntero el que está haciendo referencia a la estructura, se puede acceder a los campos con el operador ->, que simboliza una flecha a modo de decir que el puntero está "apuntando a". Este formato de acceso se puede utilizar si se le da la dirección de una variable estructurada a un puntero y luego se desea acceder a los campos a través de este:

puntero->campo

Este mismo formato también se puede utilizar en los casos que se ha reservado con *new* un espacio de memoria utilizando un tipo estructurado, ya que habrá un puntero apuntando a ese espacio.

Se puede acceder a un campo dentro de un espacio estructurado, con un puntero, con el formato de acceso a miembro, si se utiliza la siguiente sintaxis:

(*puntero).campo

A través de cualquiera de los formatos anteriores se puede acceder, de manera individual, a los campos dentro de una estructura. Pero también podemos manejar la estructura como un todo, por ejemplo, cuando mandamos una variable estructurada como parámetro a una función, cuando una función nos devuelve un objeto de tipo estructurado, o cuando realizamos una asignación. Por ejemplo, si las variables *algo1* y *algo2* son del mismo tipo estructurado, podemos escribir:

```
algo2 = algo1;
```

En este caso C/C++ tomará a *algo1* como un todo y lo asignará en bloque a *algo2*.

Ejemplo:

En este ejemplo se demuestra cómo se define una estructura y cómo se realiza la declaración de una variable estructurada y el acceso a sus campos.

```
#include<iostream>
```

```
using namespace std;

struct estructura{
   int campo1;
   float campo2;
   char campo3;
};

int main(void)
{
   estructura miVariable, tuVariable, *p;

   cout << endl;
   cout << "PRIMER EJEMPLO DE USO DE UNA ESTRUCTURA" << endl << endl;

   p = &miVariable;

   p->campo1 = 5;
   (*p).campo2 = 37.54;
   miVariable.campo3 = 't';

   tuVariable = miVariable;

   cout << "El campo 1 de miVariable es: " << miVariable.campo1 << endl;
   cout << "El campo 2 de miVariable es: " << p->campo2 << endl;
   cout << "El campo 3 de miVariable es: " << (*p).campo3 << endl;

   cout << endl;

   cout << "El campo 1 de tuiVariable es: " << tuVariable.campo1 << endl;
   cout << "El campo 2 de tuVariable es: " << tuVariable.campo2 << endl;
   cout << "El campo 3 de tuVariable es: " << tuVariable.campo3 << endl;

   cout << endl;
```

```
    return 0;
}
```

La corrida de este programa produce la siguiente salida:

```
PRIMER EJEMPLO DE USO DE UNA ESTRUCTURA

El campo 1 de miVariable es: 5
El campo 2 de miVariable es: 37.54
El campo 3 de miVariable es: t

El campo 1 de tuiVariable es: 5
El campo 2 de tuVariable es: 37.54
El campo 3 de tuVariable es: t
```

En este ejemplo se ha definido una estructura y luego dos variables y un puntero de ese tipo. Observe sus declaraciones. Luego se muestran diferentes formas de acceder a los campos de las variables estructuradas. Note que a tuVariable no se puede acceder a través del puntero p, porque este no le apunta a ella.

Ejercicios:

1) Elabore una función que reciba las coordenadas de dos puntos e indique cual es la distancia entre ambos. Recordar que la distancia entre dos puntos del plano cartesiano se obtiene mediante la aplicación de la fórmula:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

```
#include <iostream>
#include <cmath>

using namespace std;

struct coordenadas{
```

```
    float x, y;
};

float distancia(coordenadas, coordenadas);

int main(void)
{
    coordenadas p1, p2;
    float d;

    cout << endl;
    cout << "DISTANCIA ENTRE DOS PUNTOS"  << endl << endl;
    cout << "Digite las coordenadas del primer punto:" << endl;
    cout << "x1: ";
    cin >> p1.x;
    cout << "y1: ";
    cin >> p1.y;

    cout << "Digite las coordenadas del segundo punto:" << endl;
    cout << "x2: ";
    cin >> p2.x;
    cout << "y2: ";
    cin >> p2.y;

    d = distancia(p1, p2);

    cout << "La distancia entre los dos puntos es: " << d << endl;

    cout << endl;
    return 0;
}

float distancia(coordenadas punto1, coordenadas punto2)
{
```

```
    return sqrt(pow(punto2.x - punto1.x, 2) + pow(punto2.y - punto1.y, 2));
}
```

Una corrida de este programa es:

```
DISTANCIA ENTRE DOS PUNTOS

Digite las coordenadas del primer punto:
x1: 1
y1: 1
Digite las coordenadas del segundo punto:
x2: 2
y2: 2
La distancia entre los dos puntos es: 1.41421
```

2) Se han tomado n datos de temperatura en distintos puntos de una placa metálica. Elabore una programa que lea los datos correspondientes a cada una de las temperaturas y sus coordenadas, a continuación muestre todos los datos leídos, luego busque la mayor y la menor temperaturas e imprímalas en pantalla. Para cada dato de temperatura de la placa metálica utilice la estructura datostemp, que incluye en su interior un campo estructurado de coordenadas, tal como se muestra a continuación.

```
struct coordenadas{
        float x, y;
};

struct datosTemp{
    float temp;
    struct coordenadas;
};
```

```
#include <iostream>
using namespace std;
```

```
struct coordenadas{
    float x, y;
};

struct datosTemp{
    float temp;
    coordenadas punto;
};

void leerTemperaturas(datosTemp *, int);
void mostrarTemperaturas(datosTemp *, int);
void buscarExtremos(datosTemp *, int, datosTemp *, datosTemp *);
void mostrarExtremos(datosTemp, datosTemp);

int main(void)
{
    int n;
    datosTemp tempMenor, tempMayor;

    cout << endl;
    cout << "DATOS DE TEMPERATURA EN UNA PLACA" << endl << endl;

    cout << "¿Cuantos datos de temperatura ingresará? ";
    cin >> n;
    datosTemp *temperaturas = new datosTemp[n];
    leerTemperaturas(temperaturas, n);
    mostrarTemperaturas(temperaturas, n);
    cout << "Los valores extremos son:" << endl;
    buscarExtremos(temperaturas, n, &tempMenor, &tempMayor);
    mostrarExtremos(tempMenor, tempMayor);

    cout << endl;
    return 0;
```

```
}

void leerTemperaturas(datosTemp *datos, int n)
{
   int i;

   cout << "Lea los valores de temperatura de la placa:" << endl;
   for(i = 0; i < n; i++){
      cout << "Introduzca x: ";
      cin >> (datos + i)->punto.x;
      cout << "Introduzca y: ";
      cin >> (datos + i)->punto.y;
      cout << "Introduzca valor de temperatura: ";
      cin >> (datos + i)->temp;
   }
}

void mostrarTemperaturas(datosTemp *datos, int n)
{
   int i;

   cout << "Los valores de temperatura de la placa son:" << endl;
   for(i = 0; i < n; i++){
      cout << "x: ";
      cout << (datos + i)->punto.x << endl;
      cout << "y: ";
      cout << (datos + i)->punto.y << endl;
      cout << "Temperatura: ";
      cout << (datos + i)->temp << endl;
      cout << endl;
   }
}

void buscarExtremos(datosTemp *datos, int n, datosTemp *tMenor, datosTemp *tMayor)
```

```
{
   int i;

   *tMenor = *tMayor = *datos;

   for(i = 0; i < n; i++){
      if((datos + i)->temp < tMenor->temp)
         *tMenor = *(datos + i);
     if((datos + i)->temp > tMayor->temp)
         *tMayor = *(datos + i);
  }
}

void mostrarExtremos(datosTemp menor, datosTemp mayor)
{
   cout << "La temperatura menor es: " << menor.temp << endl;
   cout << "Y se ubica en: (" << menor.punto.x << ", " << menor.punto.y << ")" << endl;
   cout << "La temperatura mayor es: " << mayor.temp << endl;
   cout << "Y se ubica en: (" << mayor.punto.x << ", " << mayor.punto.y << ")" << endl;
}
```

Una corrida de este programa es:

DATOS DE TEMPERATURA EN UNA PLACA

¿Cuantos datos de temperatura ingresará? 5

Lea los valores de temperatura de la placa:

Introduzca x: 6

Introduzca y: 8

Introduzca valor de temperatura: 21.5

Introduzca x: 3

Introduzca y: 7

Introduzca valor de temperatura: 63.4

Introduzca x: 1

Introduzca y: 6

Introduzca valor de temperatura: 45.3

Introduzca x: 6

Introduzca y: 3

Introduzca valor de temperatura: 28.4

Introduzca x: 9

Introduzca y: 2

Introduzca valor de temperatura: 46.2

Los valores de temperatura de la placa son:

x: 6

y: 8

Temperatura: 21.5

x: 3

y: 7

Temperatura: 63.4

x: 1

y: 6

Temperatura: 45.3

x: 6

y: 3

Temperatura: 28.4

x: 9

y: 2

Temperatura: 46.2

Los valores extremos son:

La temperatura menor es: 21.5

Y se ubica en: (6, 8)

La temperatura mayor es: 63.4

Y se ubica en: (3, 7)

El programa inicia solicitando la cantidad de datos sobre temperatura se van a introducir y de acuerdo a ello reserva un espacio de memoria con la capacidad de almacenar esa cantidad de datos en base a la estructura datosTemp. A partir de allí se llaman cuatro funciones. La función de lectura y llenado del espacio reservado recibe su dirección de inicio y la cantidad de valores que se deben almacenar. La función de despliegue del conjunto de datos recibe los mismos parámetros que la anterior. Para almacenar los datos de las temperaturas extremas se crean dos variables en la función main, así que a la función de búsqueda de estas temperaturas se envía el la dirección de inicio del espacio de memoria, la cantidad de estructuras almacenadas, y las direcciones de memoria de las variables que contendrán los valores extremos. Finalmente estos valores extremos se imprimen en una función que recibe ambas variables por valor.

Definición de tipos con *union*

Las uniones permiten también agrupar diferentes elementos en un solo bloque. Pero, a diferencia de las estructuras, solo uno de ellos es accesible un cualquier momento de la ejecución del programa. Es decir, de todos los campos definidos dentro de una unión, solo uno de ellos es tomado en cuenta: el último asignado.

Para definir una unión se utiliza el comando:

```
union nombre{ … };
```

Dentro de las llaves se colocan "variables" de los tipos fundamentales u otros tipos que se hayan definido previamente Las variables dentro de una unión reciben el nombre de campos.

La diferencia entre la estructura y la unión está en el manejo que se hace de ella en cuanto al uso del recurso de memoria. Por ejemplo, si se declara la siguiente unión:

```
union miUnion{
  int entero;
   long double real;
};
```

La cantidad de memoria que utilizará una variable de tipo *miUnion* es equivalente a la cantidad de bytes del campo más grande. En este caso, dado que el campo entero utiliza 4 bytes, y el campo real utiliza 8 bytes, la unión utilizará 8 bytes de memoria.

Si le agregamos a la unión el siguiente campo:

```
union miUnion2{
  int entero;
   long double real;
  int a[10];
};
```

La cantidad de memoria que utilizará una variable de tipo *miUnion2* es de 40 bytes, ya que el campo más grande es el arreglo de diez enteros, que utiliza esta cantidad de bytes.

Las uniones, por tanto, son útiles cuando se utilizará un tipo de dato, de un conjunto de tipos de datos posibles. Un ejemplo podría ser una celda de una hoja de cálculo, a la que se le pudieran introducir nada más números enteros y reales. Si se define el tipo de la celda con una unión, nos ahorraríamos algún consumo de memoria, ya que en una celda solo puede existir un dato por vez.

Ejemplo de unión:

```
union miUnion{
  int x;
   float y;
};
```

Al declarar una variable de este tipo, escribimos:

```
miUnion var1;
```

Ejemplo:

```
#include <iostream>

using namespace std;

union miUnion{
    int entero;
    float real;
    char car;
};

struct miEstructura{
    int cualEs;
    miUnion dato;
};

int main(void)
{
    miEstructura var;

    cout << endl;
    cout << "EJEMPLO CON UNION" << endl << endl;

    cout << "¿Desea un (1) entero / (2) real / (3) carácter?: ";
    cin >> var.cualEs;

    switch(var.cualEs){
        case 1:
            cout << "Digite el entero: ";
            cin >> var.dato.entero;
            break;
```

```
        case 2:
            cout << "Digite el real: ";
            cin >> var.dato.real;
            break;
        case 3:
            cout << "Digite el carácter: ";
            cin >> var.dato.car;
    }

    cout << endl << endl;

    cout << "El dato almacenado es: ";
    switch(var.cualEs){
        case 1:
            cout << var.dato.entero << endl;
            break;
        case 2:
            cout << var.dato.real << endl;
            break;
        case 3:
            cout << var.dato.car << endl;
    }

    cout << endl;

    return 0;
}
```

Definición de clases con *class*

En el paradigma de programación estructurada un tipo definido por el programador con el comando *struct* se conoce precisamente como **tipo de dato**.

En el paradigma de programación orientada a objetos, cuando utilizamos *class* para definir un tipo, a esto se le llama un **tipo de objeto**.

Las clases, definidas con el comando *class*, comparten muchas características con *struct*. Con el comando *class* también podemos agrupar un conjunto de campos, que en la jerga de C++ se conocen como ***miembros dato*** y en términos de programación orientada a objetos se conocen como ***atributos***. Pero no solo eso, sino que con el comando *class* también podemos agrupar un conjunto de funciones, que en la jerga de C++ se conocen como ***funciones miembro*** de la clase y en términos de programación orientada a objetos se conocen como ***comportamientos*** o ***métodos***.

La sintaxis del comando *class* es:

class nombre de la clase{ definición de atributos y métodos };

Donde el nombre de la clase, por convención, suele iniciar con letra mayúscula, aunque esto le es indiferente al lenguaje C++. Otros lenguajes, como java, son más estrictos en cuanto a respetar este convenio.

Los objetos de una clase solo comenzarán a existir cuando sean definidos (o declarados). Esto puede hacerse de la siguiente manera:

nombre_de_la_clase nombre_del_objeto;

Al declarar los miembros de la clase -miembros dato y funciones miembro- también se especifica el tipo de acceso de cada uno. Estos tipos de acceso se definen con las etiquetas: *public* y *private*. Otros lenguajes de programación definen más tipos de acceso, pero en este curso solo utilizaremos estos dos. Si un atributo está especificado bajo la etiqueta *private*, solo puede ser accedido por las funciones miembro de la misma clase. Si un atributo está especificado bajo la etiqueta *public*, también puede ser accedido por funciones que estén fuera de la clase.

Ejemplo:

```
class Persona{

   private:
```

```
        char nombre[35];
        float salario;

    public: // Solo tenemos la declaración forward (prototipos) de los métodos.
        Persona(char *, float);
        char *mostrarNombre(void);
        float mostrarSalario(void);
};
```

En este ejemplo los atributos *nombre* y *salario* no pueden ser accedidos por funciones externas u otras funciones miembro de otros objetos. Solo pueden ser accedidos por las funciones miembro del mismo objeto.

Puede notarse también que las funciones miembro con las que se accederá a los dos atributos anteriores están declaradas también dentro de la clase. El acceso a ellas es público, para que puedan ser invocadas desde otras funciones desde afuera de la clase. Cuando se invoca una función miembro de una clase desde otra función, a esto se le llama **enviar un mensaje** al objeto.

Los prototipos de las funciones miembro en la parte pública de la clase reciben el nombre de *interfaz de la clase*. La clase completa se define en un archivo de encabezado que termina con la extensión *.h*, es decir, este archivo contiene la interfaz de la clase.

Pero la implementación de las funciones miembro se establece por aparte, en otro archivo que tiene el mismo nombre que el archivo que contiene la clase, pero que termina con la extensión *.cpp*. En este ejemplo, las funciones miembro que retornan los valores de los atributos del objeto creado son bastante sencillas. Su código fuente es el siguiente:

```
char *Persona::mostrarNombre(void)
{
    return nombre;
}

float Persona::mostrarSalario(void)
{
```

```
    return salario;
}
```

Note que cada una de ellas solo devuelve el valor del atributo al que hace referencia. También observe que los nombres de las funciones están precedidos por el nombre de la clase y el operador de resolución de ámbito: *Persona::*.

Si las funciones miembro no estuvieran precedidas de *Persona::*, el compilador se podría confundir al considerar que son otras funciones externas a la clase. Al especificar las funciones precedidas por *Persona::* el compilador comprende que el programador se ha querido referir a las funciones miembro de la clase.

Recordar que la definición de la clase suele guardarse en un archivo aparte, con la extensión *.h*. Luego, las funciones miembro se construyen en un archivo de código fuente por separado, que por convención tiene el mismo nombre que el archivo que contiene la clase, pero termina con la extensión *.cpp*. También hay que recalcar que la ***función cliente***, *main*, se definirá en un tercer archivo que tendrá un nombre cualquiera. En la jerga de C++ este archivo se conoce como el programa cliente o programa controlador.

La función miembro constructor

Dentro de la definición de la clase anterior puede ver que se define una función miembro que tiene el mismo nombre que la clase: *Persona(char *, float)*. A esta función se le llama *constructor*. Notar que no devuelve ningún valor y que tampoco se le coloca *void* a la izquierda. En este ejemplo recibe dos argumentos, pero incluso no es obligatorio que los constructores reciban argumentos. La función constructor se escribe en el mismo archivo fuente que las demás funciones miembro.

Las funciones constructoras se invocan cuando se crea el objeto por medio de su declaración. Para nuestro ejemplo, el archivo fuente completo es el siguiente:

```
#include <iostream>
#include <cstring>

#include "ejemploClasePersona.h"

Persona::Persona(char *nom, float sal)
```

```
{
   strcpy(nombre, nom);
   salario = sal;
}

char *Persona::mostrarNombre(void)
{
   return nombre;
}

float Persona::mostrarSalario(void)
{
   return salario;
}
```

El código fuente del programa cliente

El código fuente del programa cliente se guarda en un tercer archivo. Puede tener cualquier nombre. El código fuente del programa cliente contiene la creación de los objetos, su inicialización y su manipulación. Para nuestro ejemplo, se muestra lo siguiente:

```
#include <iostream>
#include "ejemploClasePersona.cpp"
using namespace std;

int main(void)
{
   Persona pers1("Guille", 100.25);
   Persona pers2("Blanqui", 150.43);

   cout << "Datos de pers1: " << pers1.verNombre( ) << " " << pers1.verSalario( ) << endl;
```

```
   cout << "Datos de pers2: " << pers2.verNombre( ) << " " <<
pers2.verSalario( ) << endl;


}
```