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 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*

Quiero comenzar 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.

Dicho sea de paso, debido a esta característica es que algunos consideran que las estructuras son los precursores de las clases, una especie de recurso para definir tipos de objetos, aunque con menos potencia que las clases.

Para definir una estructura se utiliza el comando:

```
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;
    float precio;
    bool suspendido;
};
```

La cantidad de bytes de cada estructura definida es equivalente a la suma de los bytes que ocupan todos los campos definidos en su interior.

Con las estructuras definimos tipos de datos para ser utilizados más abajo en el programa para definir variables de esos tipos. Por ejemplo, los tipos estructurados *coordenadas* y *descripcion* pueden utilizarse para declarar una o más variables en C++, por ejemplo:

```
coordenadas punto;

descripcion algo;
```

Note que la declaración siempre es igual que lo que 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, 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*.

Podemos ver a las variables estructuradas como que contienen diferentes casillas en su interior, y cada casilla también tiene 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 un campo 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 dar 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. También puede utilizarse este formato 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

Otra sintaxis para acceder con un puntero a un campo dentro de un espacio estructurado, con el formato de acceso a miembro, es el siguiente:

(*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 hacemos una asignación directa. Por ejemplo, si las variables *algo1* y *algo2* son del mismo tipo estructurado, podemos escribir:

algo2 = algo1;

Ejemplo:

En este ejemplo se demuestra cómo se declara 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 tuVariable 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.

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 << "Digite las coordenadas del primer punto: ";
   cin >> p1.x >> p1.y;

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

   d = distancia(p1, p2);

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

}

float distancia(coordenadas p1, coordenadas p2)
{
   return sqrt(pow(p2.x - p1.x, 2) + pow(p2.y - p1.y, 2));
}
```

2) Se ha tomado la temperatura en distintos puntos de una placa metálica. Elabore una función que introduzca en un arreglo los datos correspondientes a cada una de las temperaturas y sus coordenadas. Cada casilla del arreglo debe contener los datos sobre la temperatura y sus coordenadas. El arreglo se declara en la función main y se pasa como argumento. Los tipos estructurados a utilizar son: *struct coordenadas y struct datosTemp* son:

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

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

3) Elabore una función que muestre los datos contenidos en el arreglo.

Definición de tipos con *union*

Las uniones permiten también agrupar diferentes elementos, en un solo conjunto. 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;
   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;
    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, 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 número 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{
   char cualEs;
   miUnion dato;
};

int main(void)
{
   int tipoDato;
   miEstructura var;

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

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

   var.cualEs = tipoDato;
   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 usuario 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 ***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 automáticamente 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;
}
```

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.mostrarNombre( ) << " " << pers1.mostrarSalario( ) << endl;
   cout << "Datos de pers2: " << pers2.mostrarNombre( ) << " " << pers2.mostrarSalario( ) << endl;

}
```