GUÍA DE APRENDIZAJE PARA CONCURSANTES ICPC Y IOI: *STRUCTS* Y *PAIRS* EN C++

---

1. Introducción

En varios problemas, ejercicios de concursos e incluso en el propio desarrollo de software se es común trabajar con determinadas información que a pesar de ser de diferentes tipos de datos ellas unidas representan un concepto o ente. Un ejemplo de la situación anterior puede ser trabajar con el listado de los estudiantes de un grupo donde de cada estudiante se conoce su nombre y estatura. Por tanto a la hora de definir un estudiante se necesita conocer su nombre y estatura. De como poder trabajar con esas dos informaciones por cada estudiante y mantener el vínculo entre ellas tratará la siguiente guía.

2. Conocimientos previos

2.1. Arreglo

En programación, un arreglo (llamados en inglés array) es una zona de almacenamiento continuo, que contiene una serie de elementos del mismo tipo.

2.2. Constructor

Los constructores se conocen con frecuencia como las funciones miembro esenciales requeridas para inicializar estructuras y objetos de tipo clase. En C++, los constructores se usan para estructuras para crear objetos con un método especial, para evitar un comportamiento indefinido o no inicializado.

2.3. Operador *new*

C++ y Java permite a los programadores controlar la asignación de memoria en un programa, para cualquier tipo integrado o definido por el usuario. Esto se conoce como administración dinámica de memoria y se lleva a cabo mediante el operador new. Podemos usar el operador new para asignar (reservar) en forma dinámica la cantidad exacta de memoria requerida para contener cada nombre en tiempo de ejecución. La asignación dinámica de memoria de esta forma hace que se cree un arreglo (o cualquier otro tipo integrado o definido por el usuario) en el almacenamiento libre (algunas veces conocido como el heap o montón): una región de memoria asignada a cada programa para almacenar los objetos que se asignan en forma dinámica.

3. Desarrollo

Siguiendo con la situación expuesta en la introducción si tenemos N estudiantes y de cada uno de ellos se su nombre (un valor representado por una cadena de caracteres) y estatura (un valor decimal positivo) a la hora de representar computacionalmente podrías hacerlo de una forma muy trivial utilizando dos arreglos uno que almacenes todos los nombres y otro que almacene todas la estaturas con el esquema que en la posición i de ambos arreglos está la información referente al estudiante i. Está solución aunque puede parecer efectiva no siempre es recomendable ya que cuando se puede complejizar cuando se comienza a manipular la información.

Ahora veremos como podemos resolver esto aplicando dos elementos que nos brinda C++ como lenguaje de programación

3.1. Pair

El *pair* se usa para combinar dos valores que pueden ser de diferentes tipos de datos. El *pair* proporciona una forma de almacenar dos objetos heterogéneos como una sola unidad. Básicamente se usa si queremos almacenar tuplas. El contenedor de pares es un contenedor simple definido en el encabezado $< utility >$ que consta de dos elementos u objetos de datos.

- El primer elemento se hace referencia como *first* y el segundo elemento como *second* y el orden es fijo (first, second).
- El *pair* puede ser asignado, copiado y comparado. Un arreglo de objetos asignados en un mapa o hash_map es de tipo *pair* de forma predeterminada en la que todos los primeros elementos son claves únicas asociadas con sus segundo objetos de valor.
- Para acceder a los elementos, usamos el nombre de la variable seguido del operador punto (.) seguido de la palabra clave primero o segundo.

La sintaxis para crear un *pair* es bastante sencillo:

```
pair<tipo_dato_1, tipo_dato_2> <nombre_pair>;
```

De igual forma un *pair* se puede crear e inicializar en la misma instrucción cuya sintaxis es:

```
pair<tipo_dato_1, tipo_dato_2> <nombre_pair> (<valor_1>,<valor_2>);
```

Existen otras formas de crear un *pair* las cuales veremos mas adelante. Veamos algunas funciones de los *pair* y como funciona frente a los diferentes operadores:

- **make_pair():** Esta función de plantilla permite crear un par de valores sin escribir los tipos explícitamente. La sintaxis es la siguiente

  ```
  <nombre_pair> = make_pair (valor_1,valor_2);
  ```

- **swap:** Esta función intercambia el contenido de un objeto de *pair* con el contenido de otro objeto de *pair*. Los *pair* deben ser del mismo tipo. Su sintaxis es la siguiente:

  ```
  <nombre_pair_1>.swap(<nombre_pair_2>) ;
  ```

- **tie():** Esta función funciona igual que en las tuplas. Crea una tupla de referencias de lvalue a sus argumentos, es decir, para desempaquetar los valores de la tupla (o aquí el *pair*) en variables separadas. Al igual que en las tuplas, aquí también hay dos variantes del *tie*, con y sin **ignore**. La palabra clave **ignore** ignora un elemento de tupla en particular para que no se descomprima. Sin embargo, las tuplas pueden tener múltiples argumentos, pero los *pair*

solo tienen dos argumentos. Por lo tanto, en el caso de un *pair* de pares, el desempaquetado debe manejarse explícitamente. La sintaxis de dicha función es:

```
tie(<variable_1>, <variable_2>) = <nombre_pair>;
```

- **Asiganción(=):** Asigna un nuevo objeto para un *pair* de objetos. Al primer valor se le asigna el primer valor de y al segundo valor se le asigna el segundo valor.
- **Comparación(==):** Para los dos *pairs* dados, digamos par1 y par2, el operador de comparación compara el primer valor y el segundo valor de esos dos pares, es decir, si par1.first es igual a par2.first o no y si par1.second es igual a par2.second o no.
- **No es igual (!=):** Para los dos pares dados, digamos par1 y par2, el operador != compara los primeros valores de esos dos pares, es decir, si par1.first es igual a par2.first o no, si son iguales, comprueba los segundos valores de ambos.
- **Operadores lógicos ($>=, <=$):** Para los dos pares dados, digamos par1 y par2, el =, >, también se puede usar con pares. Devuelve 0 o 1 comparando solo el primer valor del par. Para pares como $p1 = (1, 20)$ y $p2 = (1, 10)$ $p2 < p1$ debería dar 0 (ya que compara solo el primer elemento y son iguales, por lo que definitivamente no es menor), pero eso no es cierto. Aquí, el par compara el segundo elemento y, si cumple, devuelve 1 (este es solo el caso cuando el primer elemento es igual al usar un operador relacional $>$ o $<$ solamente, de lo contrario, estos operadores funcionan como se mencionó anteriormente)

Los *pair* pueden ser perfectamente almacenados en las estructuras de datos conocida por tanto podemos tener arreglos, matrices, vectores, listas, colas, pilas y árboles que almacenan *pair*. Los *pair* se puede ordenar por defecto se ordena tomando como primer criterio el valor del primer elemento en caso de igualdad se tomará como segundo criterio el valor del segundo elemento del *pair*.

Aunque se vea un tanto recursivo o raro incluso se puede pensar que no se utiliza aunque hay casos de que si. Se puede definir un *pair* donde uno o los dos elementos que lo conforman sean *pair* declarados previamente.

3.2. Struct

Una estructura (*struct*) es una forma de agrupar un conjunto de datos de distinto tipo bajo un mismo nombre o identificador. Supóngase que se desea diseñar una estructura que guarde los datos correspondientes a un estudiante como se planteo en la introdución de esta guía. Esta estructura, a la que se llamará estudiante, deberá guardar el nombre y la estatura. Cada uno de estos datos se denomina miembro de la estructura. El modelo, patrón o sintaxis de una estructura puede crearse del siguiente modo:

```
struct <nombre_struct> {
  <tipo_dato> <nombre_miembro_1>;
  <tipo_dato> <nombre_miembro_2>;
  <tipo_dato> <nombre_miembro_3>;
```

```
  ...
  <tipo_dato> <nombre_miembro_N>;
};
```

La sintaxis anterior crea el tipo de dato, pero aún no hay ninguna variable declarada con este nuevo tipo. Obsérvese la necesidad de incluir un carácter (;) después de cerrar las llaves. Para declarar variables de tipo de estructura creado se debe utilizar la sentencia la sintaxis:

```
<nombre_struct> <nombre_variable>;
```

Para acceder a los miembros ya sea para leer o modificar su valor de una estructura se utiliza el operador punto (.), precedido por el nombre de la estructura y seguido del nombre del miembro.

```
<nombre_variable>.<nombre_miembro>
```

Los miembros de las estructuras pueden ser variables de cualquier tipo, incluyendo vectores y matrices, e incluso otras estructuras previamente definidas. Las estructuras se diferencian de los arrays (vectores y matrices) en varios aspectos. Por una parte, los arrays contienen información múltiple pero homogénea, mientras que los miembros de las estructuras pueden ser de naturaleza muy diferente. Además, las estructuras permiten ciertas operaciones globales que no se pueden realizar con arrays.

Se pueden definir también punteros a estructuras y esto permite una nueva forma de acceder a sus miembros utilizando el operador flecha (->), constituido por los signos (-) y (>). El operador flecha debe utilizarse exclusivamente con vectores. El elemento que va a su izquierda, siempre tiene que ser una dirección, ya sea representada por un puntero, o utilizando el operador &.

Al igual que podía crear una estructuras de datos con cualquier tipo de datos en C++, también puede crear estructuras de datos de estructuras. Esto potencia aún más el poder de esta herramienta, ya que le permite crear matrices de dos dimensiones capaces de almacenar una cantidad grande de registros en cada posición del vector, con diferentes campos, los cuales están representados por los miembros de la estructura.

Una estructura llamada *struct* nos permite crear un grupo de variables que consta de tipos de datos mixtos en una sola unidad. De la misma manera, un constructor es un método especial, que se llama automáticamente cuando se declara un objeto para la clase, en un lenguaje de programación orientado a objetos.

Entonces, combinando estas dos metodologías diferentes, podemos decir que cuando estos constructores se definen dentro de una *struct*, estos se denominarían constructores de *struct*. Aprendamos sobre esta funcionalidad en el lenguaje de programación C++.

La sintaxis general para el constructor de Struct se puede definir a continuación:

```
struct <nombre_struct> {
  <tipo_dato> <nombre_miembro_1>;
  <tipo_dato> <nombre_miembro_2>;
```

```
  <nombre_struct>(){
     //dentro del constructor por defecto
  }

  <nombre_struct>(<tipo_dato> <nombre_miembro_1>, <tipo_dato> <
     nombre_miembro_2>){
     //dentro del constructor parametrizado
  }
};
```

Según la sintaxis anterior, una estructura se puede definir utilizando la palabra clave struct, seguida del nombre de la estructura definida por el usuario. Como podemos notar, la declaración es estructura es similar a la declaración de clase.

Después de definir la estructura, se trata de la declaración de todas las variables con diferentes tipos de datos. Entonces habíamos definido el constructor. Se define como una función con el mismo nombre que el nombre de *struct*. En la sintaxis, mostramos la declaración del constructor predeterminado y parametrizado. Una *struct* puede tener tantos constructores como se necesite pero cada constructor debe ser diferente al resto en la cantidad de parámetros que recibe y el orden de los tipos de datos de los parametros que reciben.

Para ordenar una colección de estructuras del mismo tipo basta con sobrecargar el operador (<) o implementar una función que sirva como tercer parámetro de la función *sort* en ambas funciones se debe definir el orden entre dos variables del tipo de dato de la estructura.

4. Implementación

4.1. Pair

```
#include <bits/stdc++.h>
using namespace std;

int main(){
   pair<string, double> estudiante1("Alejandro Perez", 1.80);
   pair<string, double> estudiante2, estudiante3("Jose Perez", 2.05);
   pair<string, double> estudiante4(estudiante1);

   estudiante2 = make_pair("Perico Ramirez", 1.76);

   cout << estudiante1.first << " ";
   cout << estudiante1.second << endl;

   cout << estudiante2.first << " ";
   cout << estudiante2.second << endl;

   cout << estudiante3.first << " ";
```

```
    cout << estudiante3.second << endl;

    cout << estudiante4.first << " ";
    cout << estudiante4.second << endl;

    cout << endl;
    estudiante1.swap(estudiante2);
    tie(ignore, estudiante2.second) = estudiante3;

    cout << estudiante1.first << " ";
    cout << estudiante1.second << endl;

    cout << estudiante2.first << " ";
    cout << estudiante2.second << endl;

    cout << estudiante3.first << " ";
    cout << estudiante3.second << endl;

    cout << estudiante4.first << " ";
    cout << estudiante4.second << endl;

    return 0;
}
```

4.2. Struct

```
#include <bits/stdc++.h>
using namespace std;

struct Estudiante {
    string nombre;
    double estatura;

    Estudiante(){
        nombre="Andres Valido";
        estatura = 1.75;
    }

    Estudiante(string _n, double _e){
        nombre=_n;
        estatura = _e;
    }
};

int main() {

    Estudiante e1,e4,e5,e6;
    Estudiante * e2,*e3;
```

```
    e1.nombre="Alejandro Perez";
    e1.estatura=1.89;

    e2 = new Estudiante();
    e3 = new Estudiante("Juana Carrasco", 1.69);
    e4 = Estudiante();
    e5 = Estudiante("Jorge Padron",1.78);
    e2->nombre="Perico Perez";
    e2->estatura=2.04;

    cout<<e1.nombre<<" "<<e1.estatura<<endl;
    cout<<e2->nombre<<" "<<e2->estatura<<endl;
    cout<<e3->nombre<<" "<<e3->estatura<<endl;
    cout<<e4.nombre<<" "<<e4.estatura<<endl;
    cout<<e5.nombre<<" "<<e5.estatura<<endl;
    cout<<e6.nombre<<" "<<e6.estatura<<endl;

    return 0;
}
```

5. Aplicaciones

Uno de los usos más comunes del *pair* es con los contenedores de biblioteca de plantillas estándar (STL). std::map y std::multimap almacena sus elementos en forma de parejas. De hecho, un mapa STD es esencialmente una colección de *pair*, donde el primer elemento de cada *pair* es una clave y el segundo elemento es el valor correspondiente.

El *pair*, en su simplicidad, ofrece mucha versatilidad y funcionalidad que pueden simplificar su experiencia de codificación de muchas maneras diferentes. Es una herramienta que merece un lugar en cada kit de herramientas del programador de C++.

Comprender cómo y cuándo usar *pair* puede ser fundamental para escribir código C++ limpio y eficiente. Es un testimonio del hecho de que a veces, las herramientas más simples pueden ser las más poderosas.

Las estructuras constituyen uno de los aspectos más potentes del lenguaje C. En esta sección se ha tratado sólo de hacer una breve presentación de sus posibilidades. C++ generaliza este concepto incluyendo funciones miembro además de variables miembro, llamándolo clase, y convirtiéndolo en la base de la programación orientada a objetos.

Una de las características más interesantes de las estructuras es su enorme potencial para facilitar el trabajo del programador, al permitir referirse a un grupo grande de variables mediante un único nombre. Esto es especialmente útil cuando es necesario pasar un número grande de argumentos a una función. De la forma clásica, si tuviéramos veinte variables, habría que pasar las veinte variables como veinte argumentos distintos a una función, lo cual en más de una ocasión, provocaría errores. Gracias al uso de estructuras, en cambio, bastará con pasar a la función el nombre de la variable estructura que agrupa todos los argumentos necesarios para la función.

6. Complejidad

La complejidad temporal de las funciones de *pair* es asi como las estructuras es O(1). Ahora la complejidad espacial de ambas va depender de los tipos de datos los elementos que la componen.