2. Conceptos Avanzados de C/C++

2.1 Tipos Enumerados

Una enumeración, introducida por la palabra reservada `enum` y seguida por el nombre del tipo, es un conjunto de constantes enteras representadas por identificadores. Los valores de estas constantes de enumeración comienzan en 0, a menos que se especifique lo contrario, y se incrementan en 1. Por ejemplo:

```
enum Direccion {NORTE, SUR, ESTE, OESTE};
Direccion flecha;
```

A las variables `Direccion` se les puede asignar sólo uno de los tres valores declarados en la enumeración.

Otra enumeración popular es:

```
enum Meses {ENERO = 1, FEBRERO, MARZO, ABRIL, MAYO, JUNIO, JULIO,
AGOSTO, SEPTIEMBRE, OCTUBRE, NOVIEMBRE, DICIEMBRE };
```

Esta enumeración crea el tipo definido por el usuario Meses con constantes de enumeración que representan los meses del año. El primer valor en la enumeración anterior se establece de manera explícita en 1, de manera que el resto de los valores se incrementa en 1, con lo que resultan los valores del 1 al 12. Se puede asignar un valor entero a cualquier definición de la constante de enumeración, y las constantes de enumeración subsecuentes tendrán un valor mayor en 1 a la constante que le precede en la lista.

2.2 Estructuras

Las estructuras, conocidas generalmente con el nombre de registros, representan un tipo de datos estructurado. Se utilizan tanto para resolver problemas que involucran tipos de datos estructurados, heterogéneos, como para almacenar información en archivos.

Las estructuras pueden contener variables de muchos tipos de datos distintos, en contraste con los arreglos, que contienen sólo elementos del mismo tipo de datos. Este hecho y la mayor parte de lo que digamos sobre las estructuras se aplica también a las clases. La principal diferencia entre las estructuras y las clases en C++ es que los miembros de una estructura tienen de manera predeterminada el acceso `public` y los miembros de una clase tienen como predeterminado el acceso `private`. Veremos cómo declarar estructuras, inicializarlas y pasarlas a funciones.

2.2.1 Definición de estructuras

Consideremos la siguiente definición de una estructura:

```
struct Persona{
      char *nombre;
      int edad;
};
```

La palabra reservada `struct` introduce la definición para la estructura `Persona`. El identificador `Persona` es el nombre de la estructura y se utiliza en C/C++ para declarar variables del tipo de la estructura. Los datos (y posiblemente las funciones, al igual que con las clases) que son declarados dentro de las llaves de la definición de la estructura son los campos de esta. Los campos de la misma estructura deben tener nombres únicos, pero dos estructuras distintas pueden contener campos del mismo nombre sin que haya conflictos. La definición de cada estructura debe terminar con un punto y coma.

La estructura Carta contiene dos campos (uno de tipo `char*` y otro del tipo `int`). Los campos de una estructura pueden ser variables de los tipos de datos básicos (por ejemplo, `int`, `double`, etc.) o agregados tales como arreglos y otras estructuras. Los campos de una estructura pueden ser de muchos tipos de datos. Por ejemplo, una estructura Empleado podría contener cadenas de caracteres para el nombre y los apellidos, una variable `int` para la edad del empleado, un `char` que contenga `'M'` o `'F'` para el sexo del empleado, una variable `double` para el salario por horas del empleado y así, sucesivamente.

Una estructura no puede contener una instancia de sí misma. Por ejemplo, no puede declararse una variable de la estructura Persona dentro de la definición de esta. Sin embargo, puede incluirse un apuntador a una estructura Persona. Una estructura, que contiene una variable que es un apuntador al mismo tipo de estructura, se conoce como estructura autoreferenciada.

La anterior definición de una estructura no reserva ningún espacio en memoria; lo que hace es crear un nuevo tipo de datos que se utiliza para declarar variables de esa estructura. Estas variables se declaran en forma similar a las variables de otros tipos. En las siguientes declaraciones:

```
struct Person unaPersona;
struct Person personas[10];
struct Person *personaPtr;
```

`unaPersona` se declara como una variable de la estructura Persona. `personas` se declara como un arreglo de 10 elementos de tipo Persona y `personaPtr` se declara como un apuntador a una estructura Persona. Las variables de un tipo de datos estructura dado pueden declararse también colocando una lista separada por comas entre la llave de cierre de la definición de la estructura y el punto y coma que termina la definición. Por ejemplo, las declaraciones anteriores podrían haberse incorporado a la definición de la estructura Persona de la siguiente manera:

```
struct Persona{
      char *nombre;
      int edad;
} unaPersona, personas[10], *personaPtr;
```

El nombre de la estructura es opcional. Si la definición de una estructura no contiene nombre, las variables de ese tipo de estructura puede declararse solamente entre la llave de cierre de la definición de la estructura y el punto y coma que termina esa definición.

Las únicas operaciones válidas que pueden realizarse en variables del tipo estructuras son: asignar una variable de un tipo a otra del mismo, tomar la dirección (&) de una variable, utilizar los campos de una variable y utilizar el operador `sizeof` para determinar el tamaño de una estructura. Al igual que con las clases, la mayoría de los operadores puede sobrecargarse para trabajar con variables de un tipo de estructura.

Los campos de una estructura no se almacenan necesariamente en bytes consecutivos de memoria. Algunas hay "agujeros" en una estructura, debido a que algunas computadoras almacenan tipos de datos específicos bloques de memoria de diferente tamaño, como los de media, una o doble palabra. Una palaba es la unidad estándar de memoria utilizada para almacenar datos en una computadora: generalmente 2 o 4 bytes. Consideremos la siguiente definición de una estructura en la que se declaran las variables muestra1 y muestra2 de tipo Ejemplo:

```
struct Ejemplo{
      char c;
      int i;
} muestra1, muestra2;
```

Una computadora con palabras de 2 bytes podría requerir que cada uno de los campos de Ejemplo se alinee en bloque del tamaño de 2 bytes (el tamaño de la palabra depende del equipo). En la siguiente figura se muestra un ejemplo de almacenamiento para una variable de tipo Ejemplo al que se le ha asignado el carácter 'a' y el entero 97 (se muestran las representaciones de los valores en bits).

Si los campos se almacenan usando bloque de tamaño de 2 bytes, hay un agujero de 1 byte (el byte 1 de la figura) en el almacenamiento para los objetos de tipo Ejemplo. El valor en ese agujero de un byte está indefinido. Si los valores de los campos de muestra1 y muestra2 son iguales, al compararse estas estructuras no son necesariamente iguales, ya que es muy poco probable que los agujeros indefinidos de 1 byte contengan valores idénticos.

2.2.2 Inicialización de estructuras

Las estructuras pueden inicializase mediante el uso de listas inicializadoras, como se hace con los arreglos. Por ejemplo, la declaración:

```
struct Persona unaPersona = {"Juan", 21};
```

Crea la variable `unaPersona` de tipo Persona e inicializa el campo nombre a "Juan" y el campo edad con 21. Si hay menos inicializadores en la lista que campos en la estructura, el resto de los campos se inicializa con 0. Las variables de una estructura que se declaren fuera de la definición de una función (es decir en forma externa) se inicializan con 0 si no se inicializan explícitamente en la declaración externa. Las variables de estructuras pueden también inicializar en instrucciones de asignación, asignándoles una variable de estructura del mismo tipo o asignándoles valores a los campos individuales de la estructura.

2.2.3 Uso de estructuras con Funciones

Hay dos maneras de pasar la información en las estructuras a funciones. Pueden pasar toda la estructura completa o los campos individuales de la misma. De manera predeterminada, los datos se pueden pasar por valor. Las estructuras y sus campos pueden también pasarse por referencia, pasando ya sea referencias o apuntadores.

Para pasar una estructura por referencia, se debe pasar la dirección de la variable estructura o una referencia a la misma. Los arreglos de estructuras, al igual que todos los demás tipos de arreglos, se pasan por referencia.

Sin embargo, es posible pasar un arreglo por valor mediante el uso de una estructura. Para hacer eso, se debe crear una estructura (o clase) con el arreglo como campo y después pasar una variable del tipo de esa estructura (o clase) a una función por valor. Como la variable se pasa por valor, el campo arreglo se pasa también por valor.

2.3 Typedef

La palabra reservada `typedef` cuenta con un mecanismo para crear sinónimos (o alias) para los tipos de datos previamente definidos. Los nombres para los tipos de las estructuras se definen comúnmente con `typedef` para crear nombre de tipos más cortos o legibles. Por ejemplo, la instrucción:

```
typedef struct {
      char c;
      int i;
} Persona;
```

Define un nuevo nombre de tipo, Persona, como sinónimo para una estructura.

Al crear un nuevo nombre con `typedef` no se crea un nuevo tipo de datos: `typedef` simplemente crear un nuevo nombre de tipo que puede entonces utilizarse en el programa como un alias para un nombre de tipo existente.

Con `typedef` pueden crearse sinónimos para los tipos de datos integrados. Por ejemplo, un programa que requiere enteros de 4 bytes podría utilizar el tipo `int` en un sistema y el tipo `long int` en otro sistema que tenga enteros de 2 bytes. Los programas diseñados para ser portables pueden utilizar

`typedef` para crear un alias tal como `Integer` para los enteros de 4 bytes. Este alias puede entonces utilizarse para `int` en sistemas con enteros de 4 bytes y puede utilizare con `long int` en sistemas con enteros de 2 bytes, en donde los valores `long int` ocupan 4 bytes. Así, para escribir programas portables el programador simplemente necesita declarar a todas las variables de enteros de 4 bytes como del tipo `Integer`.

2.4 Uniones

La unión (definida con la palabra reservada `union`) es una región de memoria que, con el tiempo, puede contener objetos de una variedad de tipos. Sin embargo y en cualquier momento una unión solo puede contener un máximo de un objeto, ya que los campos de una unión comparten el mismo espacio de almacenamiento. Es responsabilidad del programa asegurar que se haga referencia a los datos en una unión con el nombre de un campo del tipo de datos apropiados.

En distintas ocasiones durante la ejecución de un programa, algunos objetos podrían no ser relevantes mientras que otros sí; por lo tanto, una unión comparte el espacio en vez de desperdiciar almacenamiento con objetos que no se están usando. El número de bits utilizados para almacenar una unión debe ser al menos suficiente como para almacenar al campo más grande.

Una unión se declara con el mismo formato que usa `struct` o `class`. Por ejemplo:

```
union Numero {
      int x;
      double y;
};
```

Lo anterior indica que `Numero` es del tipo unión con los campos `int x` y `double y`. La definición de la unión debe ir antes de todas las funciones en las que se vaya a utilizar.

Las únicas operaciones integradas válidas que se pueden realizar con una unión son: asignar una unión a otra unión del mismo tipo, tomar la dirección (`&`) de una unión y utilizar sus campos mediante el operador miembro de estructura (`.`) y el operador apuntador de estructura (`->`). Las uniones no pueden compararse.

Una unión es similar a una clase en cuanto a que puede tener un constructor para inicializar a cualquiera de sus miembros. Una unión que no tiene constructor puede inicializarse con otra unión del mismo tipo, con una expresión del tipo del primer campo de la unión o con un inicializador (encerrado entre llaves) del tipo del primer campo de la unión. Las uniones pueden tener otras funciones miembro tales como destructores, pero las funciones miembro de una unión no pueden declararse como virtual. Los campos de una unión son `public` de manera predeterminada.

Una unión no puede utilizarse como una clase base en herencia (es decir, no pueden derivarse clases de las uniones). Las uniones pueden tener objetos como campos solamente si éstos no tienen un constructor, destructor o un operador de asignación sobrecargado. Ninguno de los campos de una unión puede declararse `static`.

Una unión anónima es una unión sin nombre de tipo que no trata de definir objetos o apuntadores antes del punto y como que termina su definición. Dicha unión no crea un tipo, pero si crea un objeto

sin nombre. Los campos de una unión anónima pueden utilizarse directamente en el alcance en el que se declara esta unión anónima, de la misma forma en que se utiliza cualquier otra variable local; no hay necesidad de utilizar los operadores punto (`.`) o flecha (`->`).

Las uniones anónimas tienen ciertas restricciones. Sólo pueden contener campos de datos (no funciones). Todos los campos de una unión anónima deben ser `public`. Además, una unión anónima declarada en forma global (es decir, con alcance a nivel de archivo) debe declararse explícitamente como `static`.

2.5 Operadores a Nivel de bits

C/C++ cuenta con herramientas extensas de manipulación de bits para los programadores que necesitan sumergirse hasta el nivel denominado como de "bits y bytes". Los sistemas operativos, el software de prueba de equipos, de red y muchos otros tipos de software requieren que el programa se comunique "directamente con el hardware". En esta sección hablaremos sobre las herramientas de manipulación de bits de C/C++. Veremos cada uno de los diversos operadores a nivel de bits de C/C++ y cómo ahorrar memoria mediante el uso de los campos de bits.

Las computadoras representan internamente todos los datos como secuencias de bits. Cada bit se puede asumir el valor de 0 o 1. En la mayoría de los sistemas, una secuencia de 8 bits forma un byte: la unidad estándar de almacenamiento para una variable de tipo `char`. Otros tipos de datos se almacenan en números más grandes de bytes. Los operadores a nivel de bits se utilizan para manipular los bits de los operandos integrales (`char`, `short`, `int` y `long`; tanto `signed` como `unsigned`). Los enteros sin signo se utilizan normalmente con los operadores a nivel de bits.

Los operadores a nivel de bits son: AND a nivel de bits (`&`), OR inclusivo a nivel de bits (`|`), OR exclusivo a nivel de bits (`^`), desplazamiento a la izquierda (`<<`), desplazamiento a la derecha (`>>`) y complemento (`~`). Los operadores AND, OR inclusivo y OR exclusivo a nivel de bits comparan dos operandos bit por bit. El operador AND a nivel de bits hace que cada bit en el resultado sea 1 si el bit correspondiente en ambos operadores es 1. El operador OR inclusivo a nivel de bits hace que cada bit en el resultado sea 1 si el bit correspondiente en cualquier operador (o en ambos) es 1. El operador OR exclusivo a nivel de bits hace que cada bit en el resultado sea 1 si el bit correspondiente en sólo uno de los operadores es 1.

El operador de desplazamiento a la izquierda desplaza los bits de su operando izquierdo a la izquierda, según el número de bits especificado en su operando derecho. Los lugares vacíos de la derecha se reemplazan con 0s; los bits que se desplazan más allá del último lugar a la izquierda se pierden.

El operador de desplazamiento a la derecha desplaza los bits en su operando izquierdo a la derecha, según el número de bits especificados en su operando derecho. Al realizar un desplazamiento a la derecha sobre un entero `unsigned` los lugares vacíos a la izquierda se reemplazan con 0s; los bits que se desplazan más allá del último lugar a la derecha se pierden.

El operador de complemento a nivel de bits hace que todos los bits que sean 0 en su operando sean 1 en el resultado, y hace que todos los bits que sea 1 en su operando sean 0 en el resultado; lo que también se conoce como "tomar el complemento a uno del valor".

Cada operador a nivel de bits (excepto el operador de complemento) tiene un operador de asignación correspondiente. Estos operadores de asignación a nivel de bits se muestran en la siguiente figura y se utilizan de forma similar a los operadores de asignación aritméticos.

Operadores de asignación a nivel de bits	
&=	Operador de asignación AND a nivel de bits
!=	Operador de asignación OR inclusivo a nivel de bits.
^=	Operador de asignación OR exclusivo a nivel de bits.
<<=	Operador de asignación de desplazamiento a la izquierda.
>>=	Operador de asignación de desplazamiento a la derecha, con extensión de signo.

2.6 Campos de Bits

C/C++ ofrece la habilidad de especificar el número de bits en los que se almacena un tipo integral o miembro de tipo `enum` de una clase o estructura. Dicho miembro se conoce como un campo de bits. Los campos de bits permiten una mejor utilización de memoria al guardar los datos en el mínimo número de bits requeridos. Los campos de bits deben declararse como un tipo de datos integral o `enum`.

Considere la definición de la siguiente estructura:

```
struct Carta {
      unsigned cara : 4;
      unsigned palo : 2;
      unsigned color : 1;
};
```

La definición contiene tres campos de bits `unsigned`: cara, palo y color, utilizados para representar una carta de un juego de 52 cartas. Para declarar un campo de bits se pone después de un campo de tipo integral o `enum` si el signo de dos puntos (`:`) y una constante entera que representa la anchura del campo (es decir, el número de bits en los que se almacena ese campo). La anchura debe ser una constante entera entre cero y el número total de bits utilizados para almacenar un `int` en el sistema.

La anterior definición de una estructura indica que el campo `cara` se almacena en 4 bits, el campo palo se almacena en 2 bits y el campo color en 1. El número de bits se basa en el rango de valores para cada miembro de la estructura. El campo `cara` almacenará valores entre 0 (as) y 12 (rey): 4 bits pueden almacenar un valor entre 0 y 15. El campo almacenará valores entre 0 y 3 (diamante, corazones, tréboles, espadas): 2 bits pueden almacenar un valor entre 0 y 3. Finalmente, el campo color almacenará ya se 0 (rojo) o 1 (negro): 1 bit puede guardar 0 o 1.

```
struct Ejemplo {
      unsigned a : 13;
      unsigned   : 3;
      unsigned b : 4;
};
```

Es posible especificar un campo de bits sin nombre, en cuyo caso el campo se utiliza como relleno en la estructura. Por ejemplo, la siguiente definición de una estructura utiliza como relleno un campo de 3 bits sin nombre; nada puede almacenarse en esos tres bits. El campo b se almacena en otra unidad de almacenamiento.

Un campo de bits sin nombre con una anchura de cero se utiliza para alinear el siguiente campo de bits en el límite de una nueva unidad de almacenamiento. Por ejemplo, la definición de la estructura:

```
struct Ejemplo {
      unsigned a : 13;
      unsigned   : 0;
      unsigned b : 4;
};
```

Utiliza un campo de 0 bits sin nombre para omitir los bits restantes (todos los que haya) de la unidad de almacenamiento en la que se encuentra a y así alinear b con el límite de la siguiente unidad de almacenamiento.