1151044 - Programación Orientada a Objetos
Temas adicionales

1. Valores de sólo lectura y constantes

En C++, la palabra `const` sólo indica que una variable es de sólo lectura. El valor de la variable posiblemente no se conocía en tiempo de compilación e incluso puede cambiar de formas indirectas.

```
int n;
std::cin >> n;
const int m = n;    // de antemano no sabemos cuánto valdrá m
m = 5;              // error: no podemos modificar m

int x = 5;
const int& y = x;
std::cout << y;     // 5
y = 6;              // error: no podemos modificar x usando y, porque y es const
x = 6;              // ¡ok¡
std::cout << y;     // ¡imprime 6!
```

Las variables `constexpr` deben recibir un valor conocido desde tiempo de compilación y además son `const`.

```
int n;
std::cin >> n;
constexpr int m = n;            // error: de antemano no podemos saber cuánto vale m
constexpr int tam = 5;          // ok
int a[tam];                     // ok: el tamaño es conocido en tiempo de compilación
```

En algunas situaciones, las variables `const` inicializadas con valores conocidos en tiempo de compilación actúan como si fueran `constexpr` (por ejemplo, para indicar el tamaño de un arreglo) pero no es buena idea confiar en esto.

2. Enumeradores

En la UAM, los alumnos tienen un *estado académico* (inscrito, egresado, titulado, baja, etc) que se representa numéricamente (por ejemplo, el estado 1 es "inscrito", el estado 2 es "no reinscrito", el estado "5" es titulado, etc). Con esto en mente, podríamos imaginarnos algo así:

```
struct alumno {
   std::string matricula;       // por si la matrícula fuera muy larga
   std::string nombre;
   int estado;                  // el estado académico del alumno
   //...
};
```

Ahora imaginemos que queremos imprimir al alumno, pero con la representación textual del estado y no con el valor del `int`, o que queremos revisar si el alumno tiene derecho a reinscribirse el próximo trimestre:

```
void operator<<(std::ostream& os, const alumno& a) {
   //...
   if (a.estado == 1) {
      os << "INSCRITO\n";
   } else if (a.estado == 2) {
      os << "NO REINSCRITO\n";
   } // etc
}

bool puede_reinscribirse(const alumno& a) {
   if (a.estado == 3 || a.estado == 4 || /*etc*/) {   // suspendido, baja, etc
      return false;
   }
   //... revisar alguna otra cosa (por ejemplo, si pagó)
}
```

El uso de constantes enteras "mágicas" que andan regadas por todo el código se vuelve un problema rápidamente, ya que constantemente se necesitaría revisar la documentación oficial para ver qué significa qué. Una alternativa es:

```
constexpr int INSCRITO = 1;
constexpr int NO_REINSCRITO = 2;
constexpr int SUSPENDIDO = 3;
constexpr int BAJA = 4;
//...

void operator<<(std::ostream& os, const alumno& a) {
   //...
   if (a.estado == INSCRITO) {
      os << "INSCRITO\n";
   } else if (a.estado == NO_REINSCRITO) {
      os << "NO REINSCRITO\n";
   } // etc
}

bool puede_reinscribirse(const alumno& a) {
   if (a.estado == SUSPENDIDO || a.estado == BAJA || /*etc*/) {
      return false;
   }
   //... revisar alguna otra cosa (por ejemplo, si pagó)
}
```

Sin embargo, lo siguiente idealmente no debería compilar, pero lo hace.

```
int main( ) {
   alumno a;
   a.estado = -12345;   // no es un estado válido pero compila porque .estado es un int
}
```

Un `enum` es una lista de constantes enteras que tienen su propio tipo. Esta característica del lenguaje permite resolver los problemas anteriormente mencionados.

```
enum estado_academico {
   INSCRITO = 1,
   NO_REINSCRITO = 2,
   SUSPENDIDO = 3,
   BAJA = 4,
   //... en realidad basta numerar INSCRITO y las demás se calculan de forma consecutiva
};

struct alumno {
   std::string matricula;
   std::string nombre;
   estado_academico estado;   // ahora tiene su propio tipo
   //...
};

//... las funciones se pueden programar igual que en el ejemplo anterior

int main( ) {
   alumno a;
   a.estado = -12345;         // ¡error! no se vale guardar un int cualquiera
}
```

Existe una variante de `enum` que se llama `enum class` que, entre otras cosas, obliga a usar el prefijo del tipo (es decir, en lugar de mencionar `INSCRITO` a secas, se mencionaría `estado_academico::INSCRITO`). Esto puede ser útil si varios `enum` que declaran constantes con nombres similares deben poder coexistir en el mismo programa.

3. Funciones y tipos amigos

Una función o tipo declarado como `friend` tiene acceso a las variables y funciones miembro privadas de otro tipo:

```
struct s {
   friend void f(s v);
   friend struct t;
private:
   int n = 5;
};

void f(s v) {
   std::cout << v.n;       // ¡ok! f es una función amiga
}

struct t {
   void g(s v) {
      std::cout << v.n;    // ¡ok! t es un tipo amigo
   }
};
```

Normalmente es un error de diseño usar `friend`, ya que tener que otorgar este tipo de priviligios suele ser señal de que la funcionalidad pública de nuestro tipo de dato es insuficiente para todos los usos válidos e interesantes que éste pueda tener. Sin embargo, si queremos evitar tener que programar esa funcionalidad pública extra y sólo queremos resolver alguna situación pequeña y específica, puede ser buena idea.

4. Grupos de tipos en Java (recordatorio)

En Java, los tipos normales como `int`, `char`, `float`, etc. reciben el nombre de *tipos primitivos*. Las variables de tipo `class`, de tipo arreglo como `int[]` o `String[]`, o de tipo `record` sí son objetos y tienen semántica de apuntador.

5. Conversión de objetos a cadenas en Java

Supongamos que tenemos la siguiente clase en Java:

```
class Fraccion {
   int num, den;

   Fraccion(int n, int d) {
      num = n;
      den = d;
   }
}

void main( ) {
   var f = new Fraccion(1, 2);
   System.out.println(f);   // ¡imprime algo raro! similar a imprimir un apuntador
}
```

Toda clase hereda de `Object` y este tipo tiene una función llamada `toString` que devuelve un `String`. Ésta es la función que usa Java al intentar convertir un objeto a cadena, que es lo que ocurre al usar `System.out.println` que espera una cadena (no se pueden añadir métodos a `System.out`, mientras que en C++ sí podíamos agregar nuestros propios `operator<<`). En Java, en general todo es virtual, así que podemos intentar redefinir esa función:

```
class Fraccion {
   //...
   public String toString( ) {
      return String.valueOf(num) + "/" + String.valueOf(den);
   }
}

void main( ) {
   var f = new Fraccion(1, 2);
   System.out.println(f);   // 1/2
}
```

6. Tipos `record` en Java

Supongamos que tenemos la siguiente clase en Java:

```
class Ejemplo {
   private int x, y;

   Ejemplo(int x, int y) {
      this.x = x;
      this.y = y;
   }

   int suma( ) {
      return x + y;
   }

   int x( ) {       // cosa rara: Java permite nombrar una función igual que una variable
      return x;     // permitiremos que consulten el valor del int x,
   }                // pero no ofreceremos una función que pueda modificarlo

   int y( ) {
      return y;     // lo mismo
   }
}
```

Los `record` permiten simplificar lo anterior:

```
record Ejemplo(int x, int y) {
   int suma( ) {
      return x + y;
   }
}

void main( ) {
   var r = new Ejemplo(5, 7);
   System.out.printf("%d %d %d", r.x( ), r.y( ), r.suma( ));
   r.x = 10;      // error: no se pueden modificar las variables del record
}
```

Normalmente los `record` se usan para tipos muy simples que no modelan ningún concepto demasiado interesante. Es similar a un `struct` de C++ que es `const` pero que deja todo público.