Introducción Rápida a ANSI/ISO C++

Se trata de realizar una introducción rápida al modo de programar de C++. Este texto debe utilizarse complementariamente al de "*Cambios de ANSI/ISO C++ frente a C++ 2.0 de AT&T*", en la página de la asignatura de *Tecnología de Objetos*. Para comprender este rápido "*tour*" por **C++**, es necesario conocer **C**.

Programa Hola, Mundo !

El programa Hola, Mundo ! queda como sigue:

```
#include <iostream>

using namespace std;

void main(void)
{
    cout << "Hola, Mundo !" << endl;
}
```

Respecto a C y a C++, v2.0, destaca la eliminación de la extensión en los ficheros de `#include`, y la utilización de *namespaces*. Los *namespaces* se corresponden, más o menos, con las unidades de *Pascal*. Inicialmente, sólo existe el *namespace std*, para toda la librería estándar de **C++**, incluyendo la que hereda de **C**. Para utilizar las cabeceras de **C**, se antepone una 'c' a su nombre. Por ejemplo `#include <cstdio>`, `#include <cctype>`, `#include <cstring>`. El uso de la *stdio* está considerado como anticuado por el estándar **ANSI** para **C++**. Que esté definido como anticuado indica que es posible utilizarlo por ahora, pero que en sucesivas revisiones del lenguaje lo más probable es que desaparezca.

Otra alternativa posible hubiese sido la siguiente, utilizando el operador de acceso, '::' :

```
#include <iostream>

void main(void)
{
    std::cout << "Hola, Mundo !" << std::endl;
}
```

O también:

```
#include <iostream>

void main(void)
{
    using std::cout;
    using std::endl;

    cout << "Hola, Mundo !" << endl;
}
```

Entrada y Salida

Los *streams* de **C** se han encapsulado en las clases *istream* y *ostream*, con su correspondencia para archivos en *ifstream* y *ofstream*. Existe una clase, la *fstream*, que permite la entrada y salida a un fichero, es decir, con acceso directo. La entrada y salida se produce mediante el operador de extracción (>>) e inserción (<<), respectivamente, lo cuál puede parecer un poco extraño. Estos operadores devuelve en *stream* que los llamó, por lo que es posible encadenarlos (como es posible encadenar '+', ya que devuelve la suma de dos números). Estan sobrecargados para todos los tipos, por lo que no es necesario especificar qué se escribe y/o se lee, a menos que se desee entrada/salida binaria.

```
/*
    Crea un fichero de texto utilizando varios tipos diferentes de datos
    que son formateados a texto automáticamente al escribir con <<
*/

#include <fstream>

using namespace std;

void main(void)
{
    int valor;
    ofstream fout("texto.txt");        // Creado el objeto fout,
                                       // y abierto el fichero para
                                       // escritura, al ser ofstream

    // Leer valor de teclado
    cout << "Creando fichero texto.txt\n"
         << "Introduzca un entero"      << endl;
    cin  >> valor;

    // Escribir y cerrar el fichero
    fout << "Este es un fichero de ejemplo" << '\n'
         << "Valor flotante: " << 5.6       << '\n'
         << "Otro valor: "     << valor     << endl;
    fout.close();

    cout << "Finalizada la escritura del fichero texto.txt" << endl;
}
```

Tipos de Variables y Templates

Enteros

```
char
short int
int
long int
```

A todos estos tipos se les puede preceder de la palabra clave *signed* o *unsigned*. En el caso de los *int*, se gana el bit utilizado para el signo para representar un mayor rango de números, si se declara *unsigned*.

Punto flotante:

```
float
double
long double
```

Cadenas

```
std::string
char * o char [] (heredadas de C)
```

Booleanos

```
bool
```

Declaración de variables

Las variables se declaran poniendo el tipo de la variable, el nombre de la misma, y a continuación, la inicialización, si la hubiera.

```
int x;            // Variable de tipo entero
float f = 0.5;    //  f es de tipo flotante y vale 0.5

std::string s = "Juan";
```

Cuando se declara una variable(objeto) de esta forma, la vida de ésta está limitada a su ámbito. Es decir, si se declara una variable dentro de un método de un objeto o de una función (incluido, main()), la variable será válida hasta que la función termine.
Se puede necesitar que una variable perdure a la función que lo creó mediante el manejo del *heap* que se tenía con el uso en **C** de *malloc*() y *free*(). Los operadores de C++ que realizan esa función se llaman **new** y **delete**.

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

using namespace std;

string *x;
char *y;

y = new char[10];
strcpy(y,"Prueba");

cout << y << endl;

try {
    x   = new string;    // si no hay memoria, new lanza la excepcion badalloc

    *x  = "Otra Prueba";

    cout << *x << endl;

    delete x;               // Borrar el string
    delete[] y;             // Borrar los 10 chars reservados
```

```
}

catch (bad_alloc &x)
{
    cerr << "Error de asignación de memoria." << endl;
}
}
```

El operador **new** lanza una excepción si no es posible reservar más memoria. La excepción bad_alloc está definida en la cabecera *typeinfo*. Véase más adelante la gestión de excepciones.

Conversión entre tipos

La conversión entre tipos (los *cast* de **C**), pueden realizarse utilizando los mecanismos heredados de **C**, sin embargo, es aconsejable utilizar los nuevos conversores, puesto que los *cast* de **C** se consideran anticuados:

- `reinterpret_cast<>()`

Este cast es útil entre punteros, en algunas situaciones, con tipos de datos primitivos. Este cast no realiza (es el único) comprobaciones de ningún tipo, por lo que debe ser utilizado con precaución. Por ejemplo:

```
int *x = reinterpret_cast<int*>(0x4567);
```

Estamos convirtiendo un número a una dirección.

- `static_cast<>()`

Es el cast típico para hacer conversiones entre los tipos ptimitivos, también es posible utilizarlo para realizar casts entre clases derivadas:

```
class a {};
class b:public a {};

a x;
b y;
a* base;
b* deriv;

base  = static_cast<a*>(&y);        // No sería necesario el cast
deriv = static_cast<b*>(base);      // Legal, correcto en todos los sentidos
                                    // el cast es necesario
deriv = static_cast<b*>(&x);        // Legal, pero incorrecto. Errores asegurados.
int z = static_cast<int>(6.5);      // No sería necesario el cast
```

- `const_cast<>()`

Es la **única** forma en C++ posible de *quitarle* el **const** a una variable(u objeto).

```
    const int MAX_SALARIO = 500000;
    int f(int &x) { return x + 100000; };

    // Esto es ilegal, pero se admite.
    // Se supone que el programador sabe lo que hace
    // El comportamiento no está definido
    cout << "Max. Salario es: " << f(const_cast<int>(MAX_SALARIO))
         << endl;
```

- `dynamic_cast<>()`

Este **cast** es un tanto especial, pues forma parte del mecanismo RTTI (*Run Time Type Information*). Su propósito es convertir punteros o referencias a objetos que estén relacionados según la herencia, de unos a otros.

En el caso de manejar punteros, `dynamic_cast<>` devuelve un **NULL** si la conversión no se puede realizar. Si se está tratando de convertir referencias, entonces la excepción *bad_cast* es lanzada (es necesario que exista un gestor de excepciones que la recoja).

Supongamos que tenemos dos objetos, de clases diferentes pero relacionadas por herencia. B hereda de A.

```
#include <iostream>
#include <typeinfo>

using namespace std;

        class A { public: virtual ~A() {}; };
        class B : public A { public: ~B() {}; };
        class C { public: virtual ~C() {}; };

void main(void)
{
        B b;
        A a;
        A *ptr;
        C c;     // C no está relacionada ni con A ni con B

        try {
         cout << (ptr = dynamic_cast<A *>(&b)) // Bien
              << endl;
         cout << (ptr = dynamic_cast<B *>(ptr))// Bien
              << endl;
         cout << (ptr = dynamic_cast<B *>(&a)) // Legal, pero extraño y peligroso.
Falla
              << endl;
         cout << (ptr = dynamic_cast<A *>(&c)) // Ilegal, dynamic_cast<> falla
              << endl;

         B& refb = dynamic_cast<B&>(a);
        }
        catch (bad_cast &x)
```

```
        {
            cout << "Error en cast de referencia." << endl;
        }

        catch (...)
        {
            cout << "Error indeterminado." << endl;
        }

}
```

La salida del programa es:

```
0065FDFC
0065FDFC
00000000
00000000
Error en cast de referencia.
```

Como limitación, `dynamic_cast<>` no puede ser utilizado si la clase sobre la que se realiza la conversión no tiene declarados métodos virtuales (ver más arriba `static_cast<>`, y, más adelante, *Polimorfismo*), de ahí que se hayan declarado los destructores de las clases como virtuales, para poder realizar el ejemplo.

La primera conversión, un puntero de una clase derivada a su clase base, funciona sin problemas. Es una conversión implícita (no necesita *cast*, en realidad), conocida como *downcast*, y que es siempre segura (en una clase derivada siempre se podrá referenciar, como mínimo, su clase base).

En la segunda conversión realizamos un *cast* de un puntero a una clase base a un puntero a una clase derivada. Esto no es siempre seguro, puesto que puede que el puntero apunte realmente a una clase base, y no a la clase derivada que nosotros deseamos. En este caso concreto, la conversión es correcta, puesto que *ptr* apunta a un objeto de clase B, y la conversión se realiza. En otro caso, la conversión devolvería un **NULL**.

La tercera conversión falla, puesto que estoy tratando de convertir un puntero a A, en un puntero a B, cuando realmente el puntero a A apunta a un objeto de clase A. Estamos en el caso indicado anteriormente, y devuelve **NULL**.

El último caso es el más flagrante: se intenta convertir un puntero a un objeto de clase C, en un puntero a un objeto de clase A, cuando las clases C y A no tienen ninguna relación entre sí.

Finalmente, se intenta convertir una referencia a un objeto de clase A en una referencia a un objeto de clase B, cuando el objeto de clase A referencia realmente a un objeto de clase A. La conversión de referencias falla, y se genera una excepción.

Como se ha visto, cuando `dynamic_cast<>` falla, o bien lanza una excepción, `bad_cast`, o bien devuelve un puntero nulo. Ver más adelante el apartado "*Polimorfismo*" en **Clases y Estructuras.**

Gestión de Excepciones

La gestión de excepciones se logra con los bloques *try .. catch()*.

Por ejemplo:

```
  try {
```

```
    int * x = new int[100];
}
catch(bad_alloc &b)
{
    cerr << "Error Reservando Memoria" << endl;
}
```

Las excepciones también pueden generarse. Además, la sentencia *catch(...)* es capaz de capturar cualquier excepción. Véase en este trozo de código:

```
....

ifstream fin("numeros.txt");
int aux,n = 0;
int *p[100];

try {
    do {
        fin >> aux;
        if (aux<0 || aux>100) throw -1;

        p[n]    = new int;
        *p[n++] = aux;
        if (n>100) throw -2;

    } while(!fin.eof());
}
catch (int &x)
{
    cerr << "Error " << x << " Leyendo Archivo" << endl;
}
catch (...)
{
    cerr << "Error Reservando Memoria" << endl;
}

....
```

Las templates

Las templates se utilizan como plantillas donde se permite cualquier tipo; eso sí, una vez admitido, la comprobación de tipos se realiza como si fuese una función o clase normal.

```
#include <iostream>

template <class T>
T media(T &a, T & b)
{
    return a/2;
}

using namespace std;
```

```
void main(void)
{
    int a,b;
    cout << "Introduzca dos valores:" << endl;
    cin   >> a >> b;
    cout << endl << "La media es " << media(a,b) << endl;
                        // El tipo se deduce al ser
                        // a y b ints.
}
```

Ejemplos

```
bool x        = false;
while (!x)
  std::cout << "Hola, Mundo!\n";

std::string titulo = "Tecnología de Objetos"; std::cout << titulo;
std::cout << ' ' << titulo[titulo.size()-1];

float      val = 5.6; std::cout << val << "Introduzca "
                         "nuevo valor\n";
std::cin >> val;

const std::string version = "ANSI/ISO C++";
const int max_annos_sin_cambios = 5;
```

La STL

Una parte de la librería estándar de C++ consta de una serie de templates diseñadas para albergar colecciones de datos. Esta parte se llama STL, y contiene templates que contemplan la creación de pilas, listas ... etc. Por ejemplo:

Tipo de Estructura	Nombre del #include	Nombre del template	Ejemplo

Pila	stack	stack<>	#include <stack>
			#include <iostream>
			std::stack<float> pilaf;
			pilaf.push(5.6);
			cout << pilaf.top() <<
			pilaf.size();
			pilaf.pop();

Vector	vector	vector<>	#include <vector>
			#include <iostream>
			std::vector<int> v;
			v.push_back(6);
			v.push_back(7);
			std::cout << v[0] << v[1];

Lista	list	list<>	#include<list> #include <iostream> std::list<std::string *> lcad; lcad.push_back(new std::string("Hola, Mundo!"); std::cout << lcad[0] << std::endl; delete lcad[0];

Cola	queue	queue<>	#include <iostream>
			#include <queue>
			using namespace std;
			std::queue<int> c;
			c.push(5);
			c.push(6);
			cout << c.top();
			c.pop();
			cout << c.top();
			c.pop();
			cout << "Fin" << endl;

Llamadas a procedimiento. Paso de parámetros.

Frente a **C**, **C++** declara un nuevo paso de parámetros: el de referencia. Podemos ver un ejemplo en:

```
void f(int &x)
{
    ++x;
}
```

En este ejemplo, el parámetro que se le pasa a f(), x, es actualizado, y al volver a la función que lo llamó, x tiene el valor actualizado por f(). Es exactamente igual a una declaración en Pascal como:

```
procedure f(var x:integer)
begin
        x = x + 1;
```

```
    end;
```

Pero las referencias, el operador &, puede aplicarse también en el mismo sentido en el que se aplica el operdor *, de punteros. De hecho, una referencia podría interpretarse como un puntero que no sigue la notación de punteros. Veamos un ejemplo:

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

void main(void)
{
    int p;
    int &j = p;

    p = 5;
    cout << p << j << endl;

    j = 6;
    cout << p << j << endl;
}
```

Como se ve en el ejemplo, la incialización de la referencia, es decir, cuando se indica a qué variable va a "apuntar" (y es obligatorio inicializarlas), es totalmente distinta de la asignación. En la asignación, es como si se estuviese asignando valores directamente a p.

También es posible, devolver referencias. Supongamos el siguiente caso:

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

int &f(int &x)
{
    return ++x;
}

void main(void)
{
    int j = 10;

    cout << j << endl;
    cout << f(j) << endl;
}
```

En este ejemplo, se devuelve una referencia a x, y esto puede hacerse porque x, en realidad, es j en el programa principal, es decir, x no va a ser destruida cuando la función f() termine. Recordemos que las referencias son similares a los punteros: si devolvemos una referencia a una variable que va a extinguir su vida cuando la presente función acabe, estaremos referenciando un lugar vacío y/o erróneo del stack del espacio de memoria del programa.

Este ejemplo no es muy intuitivo, pero se aclarará cuando se vean las clases.

Funciones inline

Las funciones `inline`, son aquellas que en lugar de ser llamadas, su código se introduce en el lugar donde teóricamente se produciría la llamada. Es parecido al concepto de macro, en C, o en ensamblador.

Sintácticamente no difieren nada del resto de las funciones, y esto se suele utilizar para hacer más veloz el código en funciones pequeñas, ya que el proceso de llamada a función (uso del stack del programa ... etc) suele ser costosa, en términos de velocidad de ejecución.

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

inline char pon_cr(void)
{
      return '\n';
}

void main(void)
{
    cout << "Hola, Mundo!" << pon_cr();
}
```

Estructuras y Clases

Se mantienen las structs de C, y se define un nuevo tipo de etructura: la clase. Para amoldar la programación orientada a objetos con C, se ha equiparado a la clase con el tipo, y el objeto con la variable.

```
/* Ejemplo de utilización de estructuras */
#include <iostream>
#include <vector>

struct Persona {
    std::string nombre;
    int edad;
};

typedef std::vector<Persona> Vect_personas;

using namespace std;

void main(void)
{
    Vect_personas agenda;
    Persona aux;

    do {
        cout  << "\nIntroduzca nombre: - FIN terminar - " << endl;
        cin   >> aux.nombre;
        cout  << "\nIntroduzca edad: " << endl;
        cin   >> aux.edad;

        agenda.push_back(aux);
    } while(aux.nombre!="FIN");

    cout << endl;

    for (unsigned int n = 0; n<agenda.size()-1; n++)
                                    // n sólo vive durante el bucle
           cout << "Nombre: " << agenda[n].nombre
```

```
                << "\nEdad: " << agenda[n].edad << endl;
}
```

El mismo ejemplo podría haberse programado utilizando una clase para Persona, en lugar de una estructura. La definición sería:

```
class Persona {
    public:
        std::string nombre;
        int edad;
};
```

Y el resto del programa funcionaría exactamente igual.

Constructores y Destructores

Normalmente, las clases llevan asociados unos procedimientos que se repiten siempre que se crea un nuevo objeto de esa clase, y otro que se ejecuta cuando un objeto termina su vida.

El constructor y el destructor, no son más que. en C++, métodos que se llaman de una forma especial: igual que la clase en el caso del constructor, y con la acento circunflejo (ALT+126) y el nombre de la clase en el caso del destructor. Al igual que cualquier otro método (menos el destructor), el constructor puede estar sobrecargado, ejecutándose uno u otro según los parámetros con los que se cree la clase.

```
class Persona {
    public:

        Persona() {};    // Este constructor no hace nada.
        Persona(std::string &x, int ed = 0) :  nombre(x), edad(ed) {};
                         // Este constructor acepta un nombre o un nombre y
                         // una edad, e incializa
                         // nombre y edad con esos datos.
                         // El cuerpo del constructor está vacío.

        ~Persona() {}; // El destructor no precisa hacer nada pq no hay nada que
                       // liberar. Suele utilizarse el destructor
                       // para liberar memoria que haya
                       // sido reservada dentro del objeto.

        std::string nombre;
        int edad;
};
```

Sobrecarga de Operadores y Funciones

En **C++**, las funciones pueden sobrecargarse. Veamos el ejemplo:

```
  int calcula_area(int x) {return x*x; };    // Área del cuadrado
  int calcula_area(int x,int y) {return x*y; }; // Área del rectángulo
```

También pueden sobrecargarse los métodos de las clases y los operadores. Por ejemplo, un operador sobrecargado, podría ser:

```
struct Persona {
    ...
};

// operador << sobrecargado para salida por stream de "Persona"
// se retorna of para permitir el encadenamiento cout << x << y
// de operadores
ostream &operator<<(ostream &of,Persona &x)
{
    of  << x.nombre << ' ' << x.edad << endl;
    return of;
};

...
Persona p;

p.nombre = "Juan";
p.edad    = 23;

cout << p;
```

Encapsulación

Sin embargo, no es habitual declarar una clase para que se pueda acceder libremente a los atributos de los futuros objetos (como es el caso anterior). También es preferible encapsular los operadores de esa función en la clase. De ahí las etiquetas *public* y *private* necesarias para indicar que partes de los futuros objetos serán libremente accesibles y cuáles no.

La Orientación a Objetos, se apoya sobre tres pilares fundamentales: ecapsulación, herencia y polimorfismo.

Lo habitual (y correcto) es dejar los atributos privados, definiendo métodos de acceso a ellos cuando sea necesario (públicos). También se puede dejar pública una constante, sin necesidad de declarar un método que acceda a ella.

```
/* Ejemplo de utilización de clase */
#include <iostream>
#include <vector>

class Persona {
   private:
    std::string nombre;
    int edad;
   public:
    static const int max_edad;

    Persona() {};       // El constructor y el destructor no hacen nada.
    ~Persona(){};

    const std::string &dev_nombre(void) const { return nombre; };
                        // Devolvemos una referencia
                        // esta función, al estar declarada completamente
                        // en la declaración de la clase, se considera
                        // inline.
```

```
                        // El "const" extra impide que el código que haya
                        // dentro de este método modifique el objeto al
                        // que pertenece.

    const int dev_edad(void) const;

    void pon_edad(const int &x) { if (x<=max_edad) edad = x; };
    void pon_nombre(const std::string &x) { nombre = x; };
            // Paso de parámetro por referencia para que sea más rápido
            // le añadimos el const para que a pesar de ser por
            // referencia no pueda ser modificado dentro de la función.

    friend ostream &operator<<(ostream &of,Persona &x)
        {
            of  << x.nombre << ' ' << x.edad << endl;
            return of;
        };

};

// Este método se declara fuera de la clase pero se le indica que
// va a ser inline.
inline const int Persona::dev_edad(void) const
{
     return edad;
}

const int Persona::max_edad = 130;    // Inicialización de una constante
                                      // Debe hacerse fuera de la clase
typedef std::vector<Persona> Vect_personas;

using namespace std;

void main(void)
{
    Vect_personas agenda;
    Persona aux;
    string nombre;
    int edad;

    do {
        cout  << "\nIntroduzca nombre: - FIN terminar - " << endl;
        cin   >> nombre;
        cout  << "\nIntroduzca edad: " << endl;
        cin   >> edad;

        aux.pon_nombre(nombre);
        aux.pon_edad(edad);
        agenda.push_back(aux);
    } while(aux.nombre!="FIN");

    cout << endl;

    for (unsigned int n = 0; n<agenda.size()-1; n++)
                                    // n sólo vive durante el bucle
```

```
            cout << "Nombre: " << agenda[n].dev_nombre()
                 << "\nEdad: " << agenda[n].dev_edad()
                 << endl;

             // o también, utilizando el op sobrecargado <<,
             // cout << agenda[n] << endl;
}
```

Herencia

La herencia es otro de los conceptos importantes en la OO. Se trata de poder apoyarse en una clase ya realizada para crear otra que tiene alguna relación con la anterior, reutilizando aquellas partes comunes, lo cuál supone un ahorro en cuanto esfuerzo de creación de líneas de código sin errores muy importante. Nótese el uso de la devolución por referencia. Podemos devolver por referencia puesto que los valores a devolver están dentro del objeto. Devolveremos una referencia constante cuando no queramos que se modifique.

```
class Persona_Dir_Completa: public Persona {
    private:
            std::string direccion;
            std::string telef;
            unsigned int codpostal;
    public:
            // El constructor y el destructor no hacen nada
            // El constructor llama al constructor de Persona
            Persona_Dir_Completa() : Persona() {};
            ~Persona_Dir_Completa() {};

            const std::string &dev_direccion(void) const
                                { return direccion; };
            const std::string &dev_telefono(void) const
                                { return telef; };
            const unsigned int &dev_codpostal(void) const
                                { return codpostal; };


            void pon_direccion(const std::string &x)
                                { direccion = x; };
            void pon_telef(const std::string &x)
                                { telef = x; };
            void pon_codpostal(const unsigned int &x)
                                { codpostal = x; };
};
```

La clase *Persona_Dir_Completa*, hereda de la clase *Persona*, con lo cuál toda la parte pública del objeto *Persona* está ahora disponible en para objetos de la clase *Persona_Dir_Completa*.

Por lo tanto, el siguiente código es legal:

```
Persona_Dir_Completa javier;

javier.pon_nombre("Javi");
javier.pon_edad(23);
```

```
javier.pon_direccion("c\Percebe, 13");
javier.pon_codpostal(3456);
javier.pon_telef("980 456 78 90");

cout << "Nombre: "    << javier.dev_nombre()
     << "Edad: "      << javier.dev_edad()
     << "Teléfono "   << javier.dev_telef()
     << "Cod. Postal" << javier.codpostal()
     << endl;
```

Templates en clases

Las templates también pueden ser aplicadas a las clases. Por ejemplo, el siguiente código define una clase que es una template. Es decir, hasta que no se le dice de que tipo van a ser los objetos que se creen, realmente, no existe como clase. En este caso, definimos una clase manejo de conjuntos para cualquier tipo de dato, otras clases incluidas.

```
template <class T>
class Set
{
    private:
        T* vect_set;
        unsigned long int tam;

    public:
        Set() vect_set(NULL) {};
        ~Set() { if (vect_set) delete vect_set; };

        unsigned long int tam(void);
        void mas(const T &x);
        bool pertenece (const T &x);
        void borrar(const T &x);
        std::string listado(void);
};

struct pru {
    float a,b;
};

...

Set<int> setint;

setint.mas(45);
setint.mas(3);

Set<pru> setprus;
```

Sobrecarga de Operadores en Clases

Suele ser útil relacionar operadores con clases determinadas. Por ejemplo, supongamos el ejemplo anterior: sería bastante intuitivo asignar el '+' a la adición de elementos al conjunto, al igual que el menos para borrarlos. Cuando sobrecargamos un operador dentro de una clase, debemos tener en cuenta que uno de los operandos es implícito: es el propio objeto de esa clase que lo está ejecutando.

```
template <class T>
class Set
{
    private:
        T* vect_set;
        unsigned long int tam;

    public:
        Set() vect_set(NULL) {};
        ~Set() { if (vect_set) delete vect_set; };

        unsigned long int tam(void);
        void mas(const T &x);
        bool pertenece (const T &x) const;
        void borrar(const T &x);
        std::string listado(void) const;


        Set<T> &operator+(const T &x)
        {
            mas(x);
            return *this;    // así podemos encadenas varios +
        }

        Set<T> &operator-(const T &x)
        {
            borrar(x);
            return *this;    // así podemos encadenar varios -
        }


};
```

Existe un constructor especial, que es el constructor de copia. Este constructor es el que es utilizado cuando se define un objeto y se le inicializa con otro objeto de su misma clase. Por ejemplo:

std::string x = "Hola";
std::string y = x;

En realidad, no es demasiado especial. Se define como:

```
class clase {
    int x;
    public:
        clase(clase &c) { x = c.dev_x(); };
        int dev_x(void) const { return x; };
};
```

En este caso, la clase sólo tiene un atributo, el entero x, por lo que sólo es necesario copiarlo en el nuestro para que los dos sean iguales. En otros objetos, probablemente el constructor de copia sea algo más complejo (quizá sea necesario reservar espacio, por ejemplo), pero el mecanismo es el mismo.

No es lo mismo, por tanto:

```
clase cl1 = cl0;    // o también clase cl1(cl0); es lo mismo.
```

```
                        // Se llama al operador de copia
    cl1 = cl2;          // Se llama al operador de asignación, operator=
```

Cuando utilizamos nuestras propias clases con containers de la STL, es necesario que tengamos definido el operador de asignación(=), y el constructor de copia. Por defecto, ambos realizan una copia byte a byte del objeto que se estácopiando. Esto puede no ser deseable, por ejemplo, en objetos que contengan punteros a memoria dinámica, puesto que entonces tendremos dos objetos apuntando al mismo sitio del heap. Basta que un objeto sea borrado, y libere su espacio para automáticamente dejar al otro en un estado inconsistente.

Es necesario tener en cuenta que se debe de respetar el significado intuitivo de los operadores existentes. No es razonable asignar la adición de elemntos al conjunto al operador % (módulo), por ejemplo, además de que llevaría a grandes problemas de comprensión del código.

Polimorfismo

Es el tercer gran pilar de la Orientación a Objetos. Básicamente, se trata de hacer lo mismo que con la sobrecarga de operadores o métodos: permitir que un mismo código maneje objetos "diferentes".
Un ejemplo sencillo podría ser:

```
....

class a {
    public:
        virtual void f(void) { cout << "Hola desde A" << endl; };
};

class b : public a {
    public:
        virtual void f(void) { cout << "Hola desde B" << endl; };
};


void handle(a *x)
{
    a->f();
};

void main(void)
{
    a x;
    b y;

    handle(&a);
    handle(&b);
}
```

Las funciones "virtual" son necesarias para poder obtener polimorfismo. En otro caso, en vez de obtener la salida actual:

```
Hola desde A
Hola desde B
```

Se obtendría la salida:

```
Hola desde A
Hola desde A
```

En cambio, al ser la función f() virtual, se hace la llamada apropiada incluso con un puntero (o una referencia) a la clase base.

Es típico el utilizar el polimorfismo a la hora de almacenar objetos en algún *container*.

Supongamos la siguiente estructura de clases:

```
// Clase base pura. Define una función virtual.
// =0 significa que esa fn no se define para esta clase
class Ente {
    public:
        Ente() {};
        virtual ~Ente() {};

        virtual const std::string dev_descr(void) const    = 0;
};


class Persona : public Ente {
    private:
        std::string nombre;
    public:
        Persona() {};
        Persona(const std::string &x) {nombre = x; };
        ~Persona() {};

        const std::string dev_nombre(void) const { return nombre; };
        const std::string dev_descr(void)  const { return dev_nombre();};
};


class Animal: public Ente {
    private:
        std::string nombre;
        enum peligrosidad { poca, mucha };
        peligrosidad peligro;
    public:

         Animal() {};
         Animal(const std::string & x, peligrosidad &p) : nombre(x), peligro(p) {};

          ~Animal() {};


        const std::string dev_nombre(void) const { return nombre; };
        const std::string dev_peligro(void) const  { return (peligro == mucha? "
Peligroso": " No Peligroso");};
        const std::string dev_descr(void) const    { return dev_nombre() +
dev_peligro(); };
};
```

En esta declaraciones hemos definido tres clases, Ente, Persona y Animal. Las dos últimas heredan de la primera, así que tienen que definir obligatoriamente la función `dev_descr()`. Esa es una de las consecuencias de que una función sea virtual.

Supongamos que debemos guardar una colección de estos objetos en un *container*. Es decir, en una *lista*, un *stack*, un *vector* ... etc. Supongamos que es un *vector*. Una primera aproximación podría ser crear dos *vectores*, uno para **Animal**es y otro para **Persona**s.

Sin embargo, lo ideal debería ser poder almacenar en una sola lista objetos de estas dos clases, ya que están relacionadas (tienen una clase padre común). Al recuperarlos, sería cuestión de reconvertirlos de la clase padre a la clase correcta.

Podría definirse el vector de la siguiente manera:

```
typedef std::vector<Ente *> Vect_Entes;
```

Ahora, cada vez que almacenemos un objeto en ese vector, se guardará como puntero a su clase base. Aún así, podremos destruir el objeto prefectamente, a pesar de que no sea directamente de la *clase base* (por eso el **destructor** es *virtual*), y también podemos obtener su descripción completa, ya que éstas son funciones virtuales.

Supongamos un bucle en el programa principal, con las declaraciones anteriores realizadas:

```
Vect_Entes losentes;
// Metemos objetos en el vector
Ente *x;

x = new Persona("Emilio");
losentes.push_back(x);

x = new Animal("Bola de Pelo", Anmal::poca);
losentes.push_back(x);

// Recorremos el vector para publicar las descripciones
for (unsigned int n=0;n<losentes.size();n++)
            cout << losentes[n].dev_descr() << endl;

// Ahora vamos a hacer un listado detallado
try {
      Persona *p;
      Animal   *a;
      for(unsigned in n=0;n<losentes.size();n++)
      {
            p = dynamic_cast<Persona *>(losentes[n]);//Es una persona?
            if (p==NULL)
                if (a = dynamic_cast<Animal*>(losentes[n])!=NULL)
                            // Es animal ?
                {
                    cout << "Animal:\n"
                         << "Nombre: "  << a->dev_nombre() << '\n'
                         << "Peligrosidad: " << a->dev_peligro()
                         << endl;
                }
                else throw -1;   // Problemas
```

```
            else {  // Es Persona
                cout << "Persona:\n"
                     << "Nombre: " << a->dev_nombre() << endl;
            }
        }
    }
catch(...)
{
        cerr << "Error interno de memoria" << endl;
        exit(-1);
}

// Eliminamos a todos los elementos del vector
for (unsigned int n=0;n<losentes.size();n++)
       delete  losentes[n];
```

Este código no es más que un ejemplo para ilustrar como guardar y recuperar objetos en un *container* utilizando **polimorfismo**. Sin embargo, el **Polimorfismo** no sólo se limita a guardar objetos en un *container*. Las aplicaciones posibles dependen sólo del programador.

José Baltasar García Perez-Schofield
Tecnología de Objetos