FACULTAD DE INFORMÁTICA

Otras características de C++

TEMA

Programación orientada a objetos — Unidad 10

Autor: Luis Hernández Yáñez

FdI UCM

Contenido

FdI UCM

Sobrecarga de insertores y extractores

Como ya sabemos, un insertor es un operador << que sirve para enviar información a un flujo de salida (normalmente, la pantalla). Y un extractor es un operador >> que sirve para recoger información de un flujo de entrada (normalmente el teclado).

Ya hemos usado algunos insertores predefinidos para mostrar información en la pantalla:

`cout << dato`

donde ***`dato`*** puede ser un número, un carácter o una cadena de caracteres. La información que se muestra en la pantalla es el número, el carácter o los caracteres de la cadena.

También hemos usado algunos extractores predefinidos para leer información del teclado:

`cin >> variable`

donde ***`variable`*** puede ser de algún tipo numérico, de tipo carácter o de tipo **`string`**.

FdI UCM

Sobrecarga de insertores y extractores

En las clases que hemos desarrollado hemos incluido normalmente un método **`mostrar()`** que se encarga de mostrar la información del objeto en la pantalla:

```
unaPersona.mostrar();
```

Podría resultar interesante que la información de los objetos se pudiera mostrar en la pantalla usando un insertor, en lugar del método **`mostrar()`**:

```
cout << unaPersona;
```

Así, la visualización de datos se realizaría de forma uniforme por medio de insertores.

Efectivamente, podemos hacerlo. Los insertores se pueden sobrecargar de igual forma que cualquier otro operador.

Sobrecarga de insertores y extractores

Para definir un insertor para una tipo de datos, sobrecargamos el operador << de la siguiente forma:

```
ostream& operator<<(ostream &stream, tipo dato)
{
  // Código para el tipo de datos específico
  return stream;
}
```

El código del operador será el que proceda para el tipo concreto.

Los extractores se definen de forma similar, sobrecargando el operador >> de la siguiente forma:

```
istream& operator>>(istream &stream, tipo& dato)
{
  // Código para el tipo de dato específico
  return stream;
}
```

En ambos casos, **stream** es el flujo (de salida o de entrada).

Sobrecarga de insertores y extractores

Por ejemplo, supongamos que tenemos definido un tipo **TresD**:

```
struct TresD {
  int x, y, z;
}; // Puntos en un espacio tridimensional
```

Un insertor y un extractor adecuados para el tipo **TresD** serían:

```
ostream& operator<<(ostream &stream, TresD punto)
{
  stream << punto.x << "," << punto.y << ","
         << punto.z;
  return stream;
}

istream& operator>>(istream &stream, TresD& punto)
{
  cout << "Introduzca los valores X, Y, Z: ";
  stream >> punto.x >> punto.y >> punto.z;
  return stream;
}
```


Sobrecarga de insertores y extractores

Ahora ya podemos mostrar la información de las variables de tipo **TresD** en flujos de salida (por ejemplo, **cout**) por medio del insertor y leerlas de flujos de entrada (por ejemplo, **cin**) con el extractor:

```
int main()
{
  TresD punto;

  cin >> punto;

  cout << punto;

  return 0;
}
```


Insertores y extractores para clases

Se pueden sobrecargar los insertores y los extractores igualmente para que trabajen con los objetos de las clases.

Sin embargo, hay ciertas cosas que tener en cuenta:

- ✓ El objeto siempre es el operando derecho, por lo que la función operadora no puede ser miembro de la clase.
- ✓ Al no ser una función miembro no puede acceder a los atributos del objeto-argumento.

Por ejemplo, un insertor para los objetos de la clase **Fraccion** podría ser el siguiente:

```
ostream& operator<<(ostream &stream, Fraccion f)
{
  stream << f.numerador() << "/" << f.denominador();
  return stream;
}
```

Funciones amigas

Al no estar definida la función insertora anterior en la clase, no se deja patente que se ha construido para ser utilizada con objetos de esa clase.

Existe una alternativa: declarar la función como *amiga* de la clase.

```
class Fraccion {
...
  friend ostream& operator<<(ostream &stream, Fraccion;
  friend istream& operator>>(istream &stream, Fraccion&);
private:
  ...
};
```

Inconveniente: las funciones amigas no son funciones miembro, por lo que no se pueden definir como virtuales.

Funciones amigas

A las funciones amigas se les permite acceder a los atributos de los objetos-argumento (a lo privado de los objetos):

```
ostream& operator<<(ostream &stream, Fraccion f)
{
  stream << f._numerador << "/"
         << f._denominador << endl;
  return stream;
}

istream& operator>>(istream &stream, Fraccion& f)
{
  cout << "Numerador: ";
  stream >> f._numerador;
  cout << "Denominador: ";
  stream >> f._denominador;
  return stream;
}
```


Clases amigas

Se puede hacer que en una clase A se pueda acceder a lo privado de otra clase B haciendo que la clase A sea amiga de la clase B:

```
class B {
friend class A;
public:
  ...
private:
  int _dato;
};

class A {    // Todos los métodos pueden acceder a
public:      // lo privado de los objetos de clase B
  void funcion(B b)
  { cout << b._dato; }
  ...
};
```

Por supuesto, esto sólo se debe hacer de forma muy controlada.

Clases amigas

Si la clase que se quiere hacer amiga se declara anticipadamente, no hay que utilizar la palabra clave `class`:

```
class A; // Declaración anticipada

class B {
friend A;
public:
  ...
private:
  int _dato;
};

class A {    // Todos los métodos pueden acceder a
public:      // lo privado de los objetos de clase B
  void funcion(B b)
  { cout << b._dato; }
};
```

Prohibición de copia

Ya sabemos que para que se puedan realizar copias de objetos convenientemente en un programa, en la clase hay que implementar un constructor de copia y redefinir el operador de asignación.

Hay situaciones, sin embargo, en las que puede no resultar adecuado que se puedan crear copias de objetos (por ejemplo, cuando cada objeto ha de tener asignado un identificador único, identificador que se asigna mediante el constructor por defecto).

Si queremos que NO se puedan realizar copias de objetos en un programa, basta con que hagamos que el constructor de copia y el operador de asignación sean funciones miembro privadas:

```
class Ejemplo {
...
private:
   Ejemplo& operator=(Ejemplo&); // asignación privada
   Ejemplo(Ejemplo&); // constructor de copia privado
};
```


Plantillas de funciones

Supongamos que necesitamos una función que devuelva el valor absoluto de un número entero:

```
int abs(int n)
{
  return (n < 0) ? -n : n;
}
```

Supongamos que necesitamos obtener también el valor absoluto de enteros largos (`long int`) y reales (`float`):

```
long int abs(long int n)
{
  return (n < 0) ? -n : n;
}

float abs(float n)
{
  return (n < 0) ? -n : n;
}
```


Plantillas de funciones

¿Por qué tener que repetir para distintos tipos de datos la implementación de un mismo algoritmo que funciona igual?

C++ nos permite especificar funciones *genéricas* que trabajen con distintos tipos de datos: se especifican con una *plantilla* (`template`) en la que el tipo se indica como un parámetro de la siguiente forma:

```
template<class Tipo>
Tipo abs(Tipo n)
{
  return (n < 0) ? -n : n;
}
```

Cada vez que se haga necesaria la función para un tipo concreto se utiliza la plantilla para generar el código de la función concreta.

Plantillas de funciones

La necesidad de una función concreta se determina por el tipo del argumento con el que se invoque la función:

```
int entero;
long int largo;
float real;
...
cout << abs(entero) << endl;
// Se genera la función abs() para enteros a partir
// de la plantilla, haciendo que "Tipo" sea "int"
cout << abs(largo) << endl;
// Se genera la función abs() para largos a partir
// de la plantilla, haciendo que "Tipo" sea "long int"
cout << abs(real) << endl;
// Se genera la función abs() para reales a partir
// de la plantilla, haciendo que "Tipo" sea "float"
```

Las plantillas de funciones pueden tener varios parámetros genéricos.

Plantillas de clase

El concepto de plantilla se puede aplicar a las clases: en lugar de desarrollar, por ejemplo, una clase **PilaEnt** para enteros y otra clase **PilaLong** para enteros largos podemos desarrollar una clase **Pila** genérica (plantilla) que sirva para esos y otros casos:

```
template<class Tipo>
class Pila {
public:
  Pila() : _tope(0) { }
  ... // constructor de copia, asignación y destructor
  bool pilavacia() { return _tope == 0; }
  bool pilallena() { return _tope == MAX; }
  void push(Tipo);
  Tipo pop();
private:
  enum { MAX = 10 };
  Tipo _elems[MAX];
  int _tope;
};
```

(*continúa*)

Plantillas de clase

```
template<class Tipo>
void Pila<Tipo>::push(Tipo dato)
{
  _elems[_tope] = dato;
  _tope++;
}

template<class Tipo>
Tipo Pila<Tipo>::pop()
{
  _tope--;
  return _elems[_tope];
}
```

Los compiladores (C++ Builder incluido) necesitan que la plantilla completa (estructura **class** e implementaciones externas de los métodos) se encuentre en un único archivo **.h** (sin **.cpp**)

Plantillas de clase

La definición de la clase va precedida de la especificación de plantilla, de la misma forma que se hace con las plantillas de funciones:

```
template<class Tipo>
class Pila {
...
```

La codificación de las funciones miembro insertadas que están implementadas en la clase no se ve aquí alterada.

La implementación de una función miembro fuera de la clase debe ir precedida de la especificación de plantilla y el nombre de la función debe ir cualificado con el nombre genérico de la clase (el nombre seguido del tipo genérico entre ángulos):

```
template<class Tipo>
void Pila<Tipo>::push(Tipo dato)
{ ...
```


Plantillas de clase

Una vez creada la clase genérica, podemos crear un ejemplar concreto indicando entre ángulos el tipo particular que se desea que utilice la clase:

```
Pila<int> pila1;
Pila<long int> pila2;
Pila<Persona> pila3;
```

Las plantillas también pueden tener definidos parámetros que no sean tipos, sino valores. Por ejemplo, para crear una clase **Lista** genérica puede ser necesario indicar el tamaño deseado:

```
template<class Tipo, int MAX>
class Lista<Tipo,MAX>::push(Tipo dato)
{ ...
```

Y se pueden indicar valores por defecto:

```
template<class Tipo, int MAX = 100>
...
```

Una lista polimórfica genérica

```
// Clase Lista (genérica) - Archivo único "lista.h"
#ifndef lista_h
#define lista_h

#include <iostream>
using namespace std;

template<class X, int MAX = 100> class Lista {
public:        // Parámetro de plantilla con valor por defecto
  Lista();
  Lista(const Lista&);
  Lista& operator=(const Lista&);
  ~Lista();
  bool llena() const;
  bool vacia() const;
  int cont() const;
  bool insertar(X*);
  bool recuperar(int, X*&) const; // la posición, de 1 a cont()
  bool eliminar(int); // la posición, de 1 a cont()
  void mostrar() const;
```

(*continúa*)

Una lista polimórfica genérica

```
private:
  X* _array[MAX];
  int _cont;
};

template<class X, int MAX> Lista<X,MAX>::Lista() : _cont(0) {}

template<class X, int MAX>
Lista<X,MAX>::Lista(const Lista& otra) {
  _cont = otra._cont;
  for(int i = 0; i < _cont; i++)
    _array[i] = otra._array[i]->clon();
}

template<class X, int MAX>
Lista<X,MAX>& Lista<X,MAX>::operator=(const Lista& otra) {
  for(int i = 0; i < _cont; i++) delete _array[i];
  _cont = otra._cont;
  for(int i = 0; i < _cont; i++)
    _array[i] = otra._array[i]->clon();
  return *this;
}
```

(*continúa*)

Una lista polimórfica genérica

```
template<class X, int MAX>
Lista<X,MAX>::~Lista() {
  for(int i = 0; i < _cont; i++) delete _array[i];
}

template<class X, int MAX>
bool Lista<X,MAX>::llena() const {
  return _cont == MAX;
}

template<class X, int MAX>
bool Lista<X,MAX>::vacia() const {
  return _cont == 0;
}

template<class X, int MAX>
int Lista<X,MAX>::cont() const {
  return _cont;
}
```

(*continúa*)

Una lista polimórfica genérica

```
template<class X, int MAX>
bool Lista<X,MAX>::insertar(X* p) {
  if(_cont == MAX) return false;
  _array[_cont] = p;
  _cont++;
  return true;
}

template<class X, int MAX>
bool Lista<X,MAX>::recuperar(int pos, X*& p) const {
  if(pos < 1 || pos > _cont) return false;
  p = _array[pos-1];
  return true;
}

template<class X, int MAX>
bool Lista<X,MAX>::eliminar(int pos) {
  if(pos < 1 || pos > _cont) return false;
  delete _array[pos-1];
```

(*continúa*)

Una lista polimórfica genérica

```
  for(int i = pos; i < _cont; i++)
    _array[i-1] = _array[i];
  _cont--;
  return true;
}

template<class X, int MAX>
void Lista<X,MAX>::mostrar() const {
  cout << "Elementos de la lista:" << endl;
  cout << "----------" << endl;
  for(int i = 0; i < _cont; i++) {
    _array[i]->mostrar();
    cout << "----------" << endl;
  }
}

#endif
```

No hay **.cpp**

Una lista polimórfica genérica

Ahora podemos crear ejemplares de listas polimórficas que sirvan para mantener (punteros a) objetos de las clases de una jerarquía.

Lo que tenemos que indicar para concretar la lista es la clase raíz de la jerarquía y el número máximo de elementos esperados. Son, respectivamente, el primer y segundo parámetro de la plantilla.

Por ejemplo, una lista de hasta 300 publicaciones:

```
Lista<Publicacion,300> biblioteca;
```

También podemos crear la lista en memoria dinámica:

```
Lista<Publicacion,300>* biblioteca =
                        new Lista<Publicacion,300>();
```

En la unidad que usa el ejemplar de lista, en lugar de poner simplemente **Lista**, ponemos a continuación los valores de los parámetros de plantilla entre ángulos.

Una lista polimórfica genérica

Como el segundo parámetro de la plantilla tiene asignado un valor implícito, podemos crear ejemplares de listas indicando solamente la clase raíz:

```
Lista<Publicacion> biblioteca;
```

Se creará con sitio para 100 (punteros a) **Publicacion**es.

Como otro ejemplo, podemos crear otro ejemplar de la lista genérica para la lista de movimientos de las cuentas bancarias.

En la clase **Cuenta**, cambiamos la definición del atributo **_movs**:

```
class Cuenta {
...
private:
  Persona* _cliente;
  Fecha* _fecha;
  double _saldo;
  Lista<Movimiento,15>* _movs;
};
```


Una lista polimórfica genérica

En la implementación de la clase **Cuenta** simplemente añadimos **<Movimiento,15>** a continuación de **Lista** en el constructor de copia y en el operador de asignación. Y ¡ya está!

Podemos también crear fácilmente una lista de hasta 500 **Cuenta**s:

```
Lista<Cuenta,500> banco;
```

Como se ve, la generalidad resulta muy conveniente.

Pero la generalidad tiene un precio. Todos los ejemplares de la lista genérica que se creen funcionarán de la misma forma. Y para hacerlo, exigen ciertas cosas. Como por ejemplo, que cualquier clase para la que se adapte la lista genérica debe necesariamente tener definidos algunos métodos: **clon()** y **mostrar()** en este caso.

(En la clase **Cuenta** sería necesario añadir el método **clon()**.)

Manejo de errores

Durante la ejecución del código se pueden producir situaciones de error que hay que solucionar.

Un error puede hacer que se interrumpa de forma inmediata la ejecución del programa (*error de ejecución*, como por ejemplo un intento de división por cero) o que la ejecución del programa continúe de forma incorrecta (por ejemplo, un valor incorrecto fuera del intervalo de valores válidos), pudiéndose producir igualmente un error de ejecución posterior o simplemente produciendo resultados inesperados.

La única forma que tenemos de manejar los errores en los programas es la anticipación; una vez que se produce el error, o bien se interrumpe la ejecución o continúa de forma incorrecta.

Se debe detectar la causa del error antes de que se produzca, de forma que se pueda evitar y solucionar.

Manejo de errores

Una vez detectada la posibilidad de error, podemos actuar de distintas formas. Al detectar la existencia de un error, lo que hagamos dependerá del error concreto que se haya detectado: volver a pedir la entrada al usuario, cancelar la operación en curso, establecer valores por defecto, devolver un indicador de error (en una función) y un largo etcétera.

No hacer nada supondría dejar en un estado incorrecto al programa, por lo que normalmente al detectar la posibilidad del error, a continuación se ejecutará algún código adecuado.

Con los mecanismos de los que disponemos de momento, el código de resolución del error estaría intercalado entre el código "normal", haciendo que el código en su conjunto sea menos legible.

C++ nos proporciona los mecanismos de manejo de excepciones para separar el código de procesamiento de errores del código de procesamiento normal.

Manejo de errores

Fijémonos en la operación de obtención de un elemento de una **Pila** de enteros:

```
int Pila::pop() {
  _tope--;
  return _elems[_tope];
}
```

Si la pila está vacía, no se debe devolver elemento alguno, por lo que hay que proteger la función anterior frente a intentos de invocación cuando la pila está vacía.

Podemos pensar en simplemente no hacer nada si la pila está vacía:

```
int Pila::pop() {
  if(_tope > 0) {
    _tope--;
    return _elems[_tope];
  }
}
```


Manejo de errores

Pero el código anterior ni siquiera compila, ya que no todos los caminos de ejecución devuelven un valor.

Podemos pensar en devolver un valor especial que resulte inusual:

```
int Pila::pop() {
  if(_tope > 0) {
    _tope--;
    return _elems[_tope];
  }
  else return -9999;
}
```

Pero podríamos no encontrar tal valor inusual. Por ejemplo, si la pila está preparada para que pueda guardar *cualquier* entero.

Manejo de errores

Podríamos pensar entonces en devolver un valor que indique si ha habido o no éxito. En estos casos lo mejor es que la función devuelva el código de éxito/fracaso y el valor a devolver sea una referencia (como ya sabemos, el código de éxito/fracaso se puede devolver siempre, mientras que el valor sólo si hay éxito):

```
bool Pila::pop(int& valor) {
  if(_tope == 0) return false;
  valor = _elems[_tope];
  return true;
}
```

Sea cual sea la forma en la que se detecte y controle el error, siempre habrá un código de detección del error (`if`) y un código de manejo del error.

Manejo de excepciones

Los errores que se detectan en tiempo de ejecución también se denominan *excepciones*.

Los mecanismos de manejo de excepciones proporcionan medios alternativos de implementación de los fragmentos de código que se ocupan de tratar los errores o excepciones que se producen.

Veamos cómo funciona el manejo de excepciones en C++:

- ✓ Cada tipo o clase de error se identifica con un tipo de datos simple (`int`, `float`, ...) o con una clase de objetos-excepciones.
- ✓ Cada fragmento de código susceptible de error se identifica como un bloque `try`.
- ✓ Cuando se detecta un error se *eleva una excepción* del tipo o clase correspondiente con la instrucción `throw`.
- ✓ El código encargado de manejar cada tipo o clase de error se identifica adecuadamente y se coloca aparte (bloque `catch`).

Manejo de excepciones

Las excepciones se *elevan* mediante instrucciones `throw`:

```
throw excepción;
```

excepción es un valor de un tipo simple o un objeto de una clase.

Las excepciones se capturan y manejan con estructuras `try-catch`, que toman la forma:

```
try {
  ... // Código protegido
}
catch(Clase1) {
  ... // Manejador de errores de tipo Clase1
}
...
catch(ClaseN) {
  ... // Manejador de errores de tipo ClaseN
}
```


Manejo de excepciones

Los bloques `try` se pueden anidar, estableciéndose una jerarquía de procesamiento de excepciones; siempre se ejecuta el bloque `catch` que corresponda dentro del bloque `try` en el que se detecta la excepción.

En un bloque `catch` se puede *relanzar* la excepción con una instrucción `throw;` (sin valor/objeto; eso hace que se propague la excepción hacia el siguiente bloque `try`, hacia fuera).

Cuando se produce una excepción (se eleva con `throw`), se ejecuta el bloque `catch` que corresponda (al tipo de excepción) dentro del bloque `try` en el que se encuentra la ejecución. La ejecución se entiende que está dentro de un bloque `try` si el `throw` se encuentra en alguna instrucción del propio bloque o en alguna función cuya invocación se originó desde ese bloque.

Al terminar la ejecución del bloque `catch`, la ejecución continúa con la instrucción inmediatamente siguiente al final del bloque `try` en el que se ha manejado la excepción.

Manejo de excepciones

La excepción se puede elevar con el nombre del tipo o clase utilizado como si fuera una función:

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

int main()
{
  try {
    ...
    throw int();
  }
  catch(int) {
    cout << "Error ..." << endl;
  }
  return 0;
}
```


Manejo de excepciones

Se puede utilizar un valor del tipo (o un objeto de la clase) para elevar la excepción:

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

int main()
{
  try {
    ...
    throw 0; // 0 es de tipo int
  }
  catch(int) {
    cout << "Error ..." << endl;
  }
  return 0;
}
```


Manejo de excepciones

Se puede declarar un parámetro en el bloque `catch`, recibiendo el objeto (o valor) con el que se haya elevado la excepción

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

int main()
{
  try {
    ...
    throw 12; // 12 es de tipo int
  }
  catch(int error) {
    cout << "Error " << error << endl;
  }
  return 0;
}
```


Manejo de excepciones

Normalmente se declara una clase para cada tipo de excepción:

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

class Desbordamiento { };
class DivisionPorCero { };

int main()
{
  try {
    ...
    throw Desbordamiento();
    ...
    throw DivisionPorCero();
  }
  catch(Desbordamiento) { ... }
  catch(DivisionPorCero) { ... }
  ...
```

Si la excepción se puede producir en el código de una clase (en una función miembro), la clase de error se declara en aquella clase:

```
#include <iostream>
using namespace std;
class UnaClase {
public:
  class UnError { }; // Clase de excepciones
  void funcion() { ... throw UnError(); ... }
...
};
int main()
{
   try {
      UnaClase obj1;
      obj1.funcion();
   }
   catch(UnaClase::UnError) { ... }
   ...
```


Un ejemplo más complejo:

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

class Error {
public:
  Error(string m) : _mensaje(m) { }
  string mensaje() { return _mensaje; }
private:
  string _mensaje;
};
```

Una clase de errores en general

(*continúa*)


```
class Suberror1 : public Error {
public:
  Suberror1(string m) : Error(m) {}
};

class Suberror2 : public Error {
public:
  Suberror2(string m) : Error(m) {}
};

void eleva() throw(Suberror2) {
  throw Suberror2("suberror2");
}
```

Una clase de errores concretos

Otra clase de errores concretos

Especificación de excepciones
Indica las excepciones que eleva esa función

Se eleva la excepción en la función **eleva()**

(*continúa*)


```
int main()
{
  try {
    try {
      eleva();
    }
    catch(Suberror1 obj) {
      cout << obj.mensaje();
    }
    catch(...) {
      cout << "Por defecto";
      throw;
    }
  }
  catch(Error) {
    cout << "General";
  }
  ...
```

Así se indica que este manejador se encarga de cualquier otra excepción

Re-eleva la excepción para que sea tratada por la estructura **try-catch** del nivel anterior (aquella en la que está anidada la actual)

Como **Suberror2** es subclase de **Error**, este manejador también se ejecuta

Cuando hay jerarquías de clases de errores los manejadores de las subclases deben estar antes que los de las superclases

La biblioteca estándar de plantillas (STL)

Como sabemos, la *reutilización de software*, con el ahorro de tiempo y esfuerzo que conlleva, es uno de los objetivos primordiales de la Programación orientada a objetos.

La reutilización se consigue básicamente utilizando clases existentes y derivando nuevas clases a partir de clases existentes (herencia). En resumidas cuentas, aprovechando las clases existentes.

Así, cuanto más fácil sea aprovechar las clases existentes, más se fomentará la reutilización. Y las clases genéricas son adecuadas para muchas más situaciones que las que no son genéricas.

Una biblioteca de clases genéricas es, por definición, mucho más útil que una de clases no genéricas. C++ proporciona una biblioteca de clases genéricas: la Biblioteca estándar de plantillas (STL).

La STL es una colección de componentes reutilizables implementados como clases genéricas (plantillas).

Las clases han sido optimizadas para dotarlas de buena eficiencia.

La biblioteca estándar de plantillas (STL)

En la STL se distinguen tres categorías de componentes reutilizables:

- ✓ Los contenedores (colecciones).
- ✓ Los iteradores (índices *sofisticados* para colecciones).
- ✓ Los algoritmos.

Los *contenedores* son clases que implementan distintos tipos de colecciones con comportamientos particulares (vectores, conjuntos).

Los *iteradores* son los medios de acceso a los distintos elementos de los contenedores, proporcionando distintas formas de acceso.

Los algoritmos de la STL son procesos genéricos que se pueden aplicar en variadas situaciones.

Las clases genéricas de la STL se pueden extender por medio de la herencia, creando "clases genéricas más concretas".

La biblioteca estándar de plantillas (STL)

- ✓ Contenedores de secuencia (o secuenciales) (*De primera clase*)

  La clase **`vector`** implementa secuencias de elementos con acceso directo a los elementos e inserciones/eliminaciones rápidas por el final.

  La clase **`deque`** implementa secuencias de elementos con acceso directo a los elementos e inserciones/eliminaciones rápidas por el principio y final.

  La clase **`list`** implementa listas con inserciones/eliminaciones rápidas en cualquier posición.

- ✓ Contenedores asociativos (*De primera clase*)

  La clase **`set`** implementa conjuntos de elementos con búsquedas rápidas y sin posibilidad de elementos duplicados.

  La clase **`multiset`** implementa conjuntos de elementos con búsquedas rápidas y con posibilidad de elementos duplicados.

  La clase **`map`** implementa conjuntos de asociaciones (pares) con búsquedas rápidas por clave y sin posibilidad de elementos duplicados.

  La clase **`multimap`** implementa conjuntos de asociaciones (pares) con búsquedas rápidas por clave y con posibilidad de elementos duplicados.

La biblioteca estándar de plantillas (STL)

- ✓ Adaptadores de contenedores (*No de primera clase*)

  La clase **`stack`** implementa pilas *(último en entrar, primero en salir).*

  La clase **`queue`** implementa colas *(primero en entrar, primero en salir)*.

  La clase **`priority_queue`** implementa colas con prioridad.

Archivos de cabecera correspondientes:

`<vector>`, **`<deque>`**, **`<list>`**, **`<set>`** (también para **`multiset`**), **`<map>`** (también para **`multimap`**), **`<stack>`** y **`<queue>`** (también para **`priority_queue`**). En todos los casos, **`using namespace std;`**.

Las clases de contenedores se han diseñado cuidadosamente para que proporcionen una funcionalidad similar, por lo que hay muchos métodos que se encuentran definidos en todas las clases de contenedores con la misma funcionalidad.

Pero, obviamente, como se trata de distintas clases, también hay métodos específicos de cada una de las clases.

Métodos comunes a todos los contenedores STL

Por supuesto, todas las clases de contenedores tienen definido su constructor predeterminado, su destructor, su constructor de copia y su operador de asignación (FCO).

Otros métodos generales:

- **`empty()`** indica si el contenedor está o no vacío.
- **`max_size()`** devuelve el tamaño máximo del contenedor.
- **`size()`** devuelve el número de elementos en el contenedor.
- **`swap()`** intercambia los elementos con otro contenedor.

También, los operadores relacionales para comparar contenedores.

Hay métodos específicos de los contenedores de primera clase:

- **`begin()`** devuelve un iterador apuntando al primer elemento.
- **`end()`** devuelve un iterador apuntando al último elemento.
- **`rbegin()`** devuelve un iterador inverso apuntando al último elemento.
- **`rend()`** devuelve un iterador inverso apuntando al primer elemento.
- **`erase()`** elimina uno o más elementos del contenedor.
- **`clear()`** elimina todos los elementos del contenedor.

Uso de contenedores STL

Para poder usar contenedores con distintas clases de elementos, es necesario que el tipo en cuestión disponga de una funcionalidad mínima que resulte adecuada para el contenedor. Concretamente, las clases deberán disponer de constructor de copia y operador **=**.

Para los contenedores asociativos (así como para determinados algoritmos), resulta necesario disponer de dos relacionales: **==** y **<**.

Como ejemplo representativo, vamos a ver ejemplos de uso del contenedor **`vector`**. Además de con el constructor predeterminado:

```
vector<int> vec1; // inicialmente sin elementos
                  // y crece según las necesidades
```

se pueden crear vectores con un determinado número de elementos y proporcionar un valor inicial para los elementos:

```
vector<int> vec2(100, 0); // 100 elem. inicializados a 0
```

Y, por supuesto, también se pueden crear por copia.

Uso de contenedores STL

El acceso a los elementos del vector se puede realizar de varias formas. Podemos utilizar los corchetes (**`vec1[6]`**), pero con ellos no se comprueba la validez del índice proporcionado.

El método **`at()`** eleva una excepción de *fuera de intervalo* si el índice proporcionado no corresponde a una posición válida:

```
cout << vec1.at(6);
```

También están definidos los métodos **`front()`** y **`back()`**, que devuelven el primer y último elemento, respectivamente:

```
cout << vec1.front() << " ... " << vec1.back();
```

Para saber cuántos elementos hay en el vector usamos **`size()`**.

Para insertar un elemento por el final se usa **`push_back(`*`elem`*`)`** y para eliminar el último elemento se usa **`pop_back()`**.

Para entender otras operaciones de los vectores, necesitamos conocer primero los iteradores.

Iteradores STL

Los iteradores se usan con los contenedores de primera clase y son similares a los índices o punteros que usamos para recorrer listas, pero con una funcionalidad (y seguridad) muy superior.

Por ejemplo, para acceder al elemento apuntado por un iterador utilizamos el operador *. Y para obtener un iterador que apunte al siguiente elemento o al elemento anterior del iterador actual usamos los operadores de incremento (++) y decremento (--).

Hay diversos tipos de iteradores que se adaptan a las características específicas de los distintos tipos de contenedores.

Se distinguen cinco tipos de iteradores:

- ✓ Iteradores de entrada: sólo para leer (acceder); se mueven hacia delante.
- ✓ Iteradores de salida: sólo para escribir (poner); se mueven hacia delante.
- ✓ Iteradores hacia adelante: para leer y escribir; se mueven hacia delante.
- ✓ Iteradores bidireccionales: lectura/escritura; hacia delante y hacia atrás.
- ✓ Iteradores de acceso aleatorio: lectura/escritura; acceso directo.

Iteradores STL

Se puede usar un iterador de acceso aleatorio como bidireccional, uno bidireccional como hacia delante y uno hacia delante como de entrada o como de salida.

Cada tipo de contenedor de primera clase admite ciertos iteradores. Por ejemplo, los contenedores **vector** y **deque** admiten iteradores de acceso aleatorio, mientras que **list** y todos los asociativos sólo admiten iteradores bidireccionales.

Los contenedores definen varios tipos para los iteradores:

- ✓ **iterator** es el tipo normal que permite lecturas y escrituras.
- ✓ **const_iterator** es para iteradores de sólo lectura.
- ✓ **reverse_iterator** es para iteradores de lectura/escritura que se desplazan al revés, de forma que con ++ se pasa al elemento anterior (iteradores inversos que recorren la colección en sentido contrario).
- ✓ **const_reverse_iterator** es para iteradores inversos de sólo lectura.

Operaciones definidas para los iteradores STL

Todos los iteradores tienen definido el operador de incremento:

```
++iter      iter++
```

Los iteradores de entrada tienen definido el operador de acceso * (sólo para obtener elementos), el operador de asignación y los operadores relacionales == y !=.

Los iteradores de salida tienen definido el operador de acceso * (sólo para colocar elementos) y el operador de asignación.

Los iteradores hacia delante tienen la funcionalidad de los de entrada y la de los de salida.

Los iteradores bidireccionales tienen definido también el operador de decremento (--).

Los iteradores de acceso aleatorio tienen definidos los demás operadores relacionales (<, >, <= y >=), así como la suma y la resta de posiciones (con sus abreviaturas).

Ejemplo de uso de vectores e iteradores

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

template<class T>
void mostrar(const vector<T> &v);

int main() {
  vector<int> vec;
  cout << "El tamaño inicial de vec es: " << vec.size() << endl;

  vec.push_back(2);
  vec.push_back(3);
  vec.push_back(4);
  cout << "El tamaño de vec ahora es: " << vec.size() << endl;

  cout << "Contenido del vector utilizando un iterador:" << endl;
  mostrar(vec);
```

(*continúa*)

Ejemplo de uso de vectores e iteradores

```
  cout << "Contenido del vector al revés:" << endl;
  vector<int>::reverse_iterator iterinv; // Iterador inverso
  for(iterinv = vec.rbegin(); iterinv != vec.rend(); ++iterinv)
    cout << *iterinv << " ";
  cout << endl;

  cout << "El primer elemento de vec es: " << vec.front() << endl;
  cout << "El último elemento de vec es: " << vec.back() << endl;

  vec[0] = 7; // Pone a 7 el primer elemento
  vec.at(2) = 10; // Pone a 10 el tercer elemento
  cout << "Contenido del vector:" << endl;
  mostrar(vec);

  vec.insert(vec.begin() + 1, 22);
  // Inserta 22 como segundo elemento
  cout << "Contenido del vector:" << endl;
  mostrar(vec);
```

(*continúa*)

Ejemplo de uso de vectores e iteradores

```
    try {
      vec.at(100) = 777; // Fuera de intervalo
    }
    catch(out_of_range) {
      cout << "Intento de acceso a posición inexistente" << endl;
    }

    vec.erase(vec.begin());
    cout << "Contenido del vector después de eliminar el primero:"
         << endl;
    mostrar(vec);

    vec.erase(vec.begin(), vec.end());
    cout << "Ahora el vector " << (vec.empty() ? "está" : "no está")
         << " vacío" << endl;

    return 0;
}
```

(*continúa*)

Ejemplo de uso de vectores e iteradores

```
template<class T>
void mostrar(const vector<T> &v) {
  vector<T>::const_iterator iter; // Iterador de lectura
  for(iter = v.begin(); iter != v.end(); iter++)
    cout << *iter << " ";
  cout << endl;
}
```

Las funciones que procesan vectores se definen como plantillas

Algoritmos de la STL

La STL separa los algoritmos de los contenedores, de forma que los algoritmos de tratamiento de los contenedores no se encuentran en las clases como métodos, sino que están aparte (mayor eficiencia).

Los contenedores se procesan por medio de los iteradores, de forma que los algoritmos que procesan contenedores reciben iteradores de dichos contenedores.

Por ejemplo, si **v** es un **vector** de enteros, para localizar el elemento 100 en el vector podemos utilizar el algoritmo **find()**, que acepta un iterador apuntando al elemento en el que empezar la búsqueda, otro iterador apuntando al elemento en el que terminar la búsqueda (exclusive) y el elemento a buscar. Para buscar en todo el vector:

```
find(v.begin(), v.end(), 100); // Devuelve iterador
```

También podemos usar el algoritmo **sort()** para ordenar el vector (**sort(v.begin(), v.end())**. Hay multitud de algoritmos disponibles que se utilizan de forma similar. `#include <algorithm>`