El lenguaje de consulta de objetos (OQL)

El objetivo fundamental del OQL es proporcionar acceso, mediante un lenguaje de declaración de consultas, a los objetos contenidos en una base de datos de acuerdo con el modelo de objetos de la aplicación definido por las clases descritas en el esquema de objetos de la base de datos.

Una consulta OQL se puede especificar:

- Desde una aplicación C++.
- A través de una línea de comando suministrada *ad hoc* por el usuario desde un entorno de tipo *shell*.
- Desde un entorno basado en un lenguaje de cuarta generación (4GL).
- Mediante una interface gráfica de usuario (GUI).

Los dos últimos proporcionarían una interface amigable para especificar las consultas y ver los resultados.

Las aplicaciones C++ generalmente utilizan la interface de navegación directa a la base de datos:

- En sus inicios, las base de datos orientadas a objetos hicieron mayor hincapié en la integración con C++ que en proporcionar un lenguaje de consultas del tipo del SQL.
- A medida que los desarrolladores comenzaron a aplicar las bases de datos de objetos a un espectro más amplio de problemas reconocieron que disponer de un lenguaje de navegación por la base de datos era importante.

El lenguaje de consulta de objetos (OQL) ...

En un entorno cliente-servidor, puede resultar ventajoso *submitir* una consulta para que la evalúe un servidor independiente (*back-end*) que devuelva los resultados a la aplicación cliente. Esta técnica sólo es efectiva si la arquitectura de la base de datos implementa una cache con soporte de consultas.

- En ocasiones, la navegación puede dar lugar a un trasiego innecesario de objetos desde el servidor al cliente.

  Por ejemplo, supongamos que una aplicación activa un objeto pero sólo está interesada en otros objetos relacionados (accesibles únicamente mediante una navegación multinivel). Los objetos intermedios pueden necesitarse sólo como expresión del recorrido de acceso. Si el cliente submite una consulta especificando la expresión de recorrido, el servidor puede devolver sólo aquellos objetos de interés.
- Por contra, si la aplicación realiza la navegación, todos los objetos intermedios en el recorrido deben ser activados en ésta, aun cuando no sean necesarios.
- Un lenguaje de consulta además proporciona un tipo de acceso asociativo basado en la especificación de ciertas restricciones en el valor de los atributos, lo que redunda en una mejor interface a la base de datos. Si el servidor de la base de datos puede realizar este tipo de consultas sólo transferirá los objetos de interés para la aplicación.
- Una de las ventajas significativas de una base de datos relacional es que cuenta con un lenguaje de consulta relacional estándar – SQL – que puede ser utilizado *ad hoc* por el usuario. Los usuarios requieren frecuentemente este tipo de acceso directo, fácil de usar e independiente de la aplicación del programador.

Criterios de diseño del OQL

El OQL fue diseñado por François Bancilhon, Sophie Cluet y Claude Delobel de la empresa O_2 Technology y fue adoptado por el ODMG como el lenguaje de consulta estándar.

El OQL está basado en el modelo de objetos del ODMG, lo que implica que:

- Soporta el sistema de objetos definido en el esquema de la base de datos.
- Soporta directamente las construcciones del modelo de objetos: encapsulado, herencia y polimorfismo tal y como se han definido en el diseño de la aplicación.
- Proporciona acceso a la base de datos tanto asociativo como navegacional.

El OQL es un superconjunto de la definición mínima del SQL-92, que es la interface común de los desarrolladores de bases de datos relacionales.

Las características y singularidades más importantes del OQL son:

1. A diferencia del SQL, el OQL no pretende ser la interface primaria o única de acceso a la base de datos ni tampoco cuenta con su propio sistema de tipos para representar la información.
   - Opera dentro del contexto de sistema de objetos definido por la aplicación.
   - Esta filosofía contrasta con la utilizada en el esquema relacional: el SQL es la única interface a la base de datos, por lo que es necesaria una traducción entre los tipos conocidos por SQL y los utilizados por el lenguaje de programación de la aplicación.

Criterios de diseño del OQL ...

2. En una base de datos orientada a objetos, la aplicación y la base de datos comparten un único sistema de objetos. Existe además un alto grado de integración con el lenguaje de programación usado por la aplicación, permitiendo una gran interoperatividad entre la aplicación y el lenguaje de consulta.
3. Por su diseño, el OQL no es un lenguaje computacionalmente completo:
   - Depende del lenguaje de programación utilizado por la aplicación: necesita de la definición de las funciones y los métodos a la hora de especificar una consulta.
   - Las expresiones de consulta se componen de un pequeño conjunto de operadores definidos en el lenguaje.
4. El OQL sólo proporciona operadores de consulta y no dispone de herramientas para modificar objetos.
   - Permitir la actualización directa de los atributos podría considerarse una ruptura de la encapsulación.
   - Por contra, el OQL permite ejecutar sobre los objetos las operaciones definidas por la aplicación. Estas funciones pueden ser tanto funciones miembro de C++ como funciones independientes no asociadas a una clase concreta.
   - Las funciones proporcionan la semántica correcta de actualización sobre la base de la representación e implementación del objeto.
   - Por tanto, OQL soporta la encapsulación definida por el modelo de objetos, invocando las operaciones definidas por el diseñador de la clase para alterar el estado del objeto.
   - Esto es exactamente lo que se espera de un lenguaje de consulta de objetos.

Criterios de diseño del OQL ...

5. Muchos desarrolladores que trabajan en el entorno SQL no desean que los usuarios puedan actualizar la base de datos directamente ya que puede ser el origen de la corrupción de los datos.
   - Normalmente utilizan restricciones semánticas de integridad para evitar las actualizaciones que pueden causar cambios incorrectos o inconsistentes en la información.
   - OQL no cuenta con un operador `update` como en SQL, pero la aplicación puede permitir al entorno de consulta utilizar funciones y métodos que realizan la modificación.
   - Este mecanismo proporciona un método más directo y explícito de asegurar la integridad de los datos que en el entorno SQL.
6. El OQL es un lenguaje de consulta *funcional*.
   - Cada expresión de consulta consiste en un operador y cero o más operandos de un tipo particular y devuelve un valor con tipo (el resultado de una consulta puede ser de cualquier tipo).
   - En contraposición, una consulta SQL devuelve una tabla de tuplas. La habilidad del OQL para devolver un valor permite a los usuarios crear funciones más efectivas.
7. Para construir expresiones de consulta arbitrariamente complejas se utiliza la composición de funciones.
   - Cada consulta válida es una consulta completa que puede utilizarse como operando de otra expresión de consulta.
   - Esta estructura permite especificar tanto consultas simples como elaboradas.

Expresiones y operadores de OQL

El OQL consiste en un conjunto de operadores de consulta (*query operators*) que se pueden combinar mediante composición de funciones.

Como el C++, el OQL es un lenguaje que impone la definición de tipos (*strongly typed language*); cada expresión de consulta espera operandos de un determinado tipo y devuelve un resultado de un tipo concreto. La comprobación de tipos se realiza en el proceso de análisis de la consulta.

Utilizaremos determinados nombres simbólicos para representar el tipo de las expresiones que se utilizan como operandos de un operador de consulta. En la tabla se muestran los símbolos y su tipo asociado.

Símbolo	Expresión OQL
`attr`	Atributo de objeto
`bool`	Expresión booleana
`char`	Expresión carácter simple
`coll`	Una colección
`dict`	Un `dictionary`
`expr`	Expresión OQL arbitraria
`func`	Función
`i, j, k`	Expresión entera
`list`	Una `list`
`num`	Un número entero o punto flotante
`ref`	Referencia a objeto
`str`	Cadena de caracteres
`T`	Tipo arbitrario

Acceso a objetos

El OQL permite el acceso a los objetos de la base de datos al mismo nivel de abstracción al que están definidos en la aplicación. Se puede acceder a las propiedades de los objetos, incluyendo atributos, relaciones y operaciones.

Objectos con nombre

El nombre de un objeto constituye una expresión válida y completa de consulta que se puede utilizar para acceder a un objeto. El tipo del objeto determina el tipo del resultado de la expresión de consulta.

Supongamos que existe un objeto colección denominado `Deptos` que contiene una referencia a cada instancia de la clase `Departamento` y que en C++ se utiliza la función `set_object_name` para asociar el nombre con el objeto. La expresión:

```
Deptos
```

es una expresión de consulta OQL que devuelve la colección de objetos.

- Si una aplicación accede frecuentemente a un conjunto particular de objetos es posible crear una colección que contenga una referencia a cada instancia.
- La aplicación puede mantener múltiples colecciones con nombre. El conjunto de objetos que debe ser procesado en conjunto se especifica estáticamente mejorando la eficiencia del proceso de consulta (ya que no se requiere evaluación alguna en tiempo de ejecución).

Acceso a los atributos y recorrido de relaciones

- Dada una expresión `ref` que hace referencia a una instancia de tipo T_1 y `attr`, un atributo de T_1 de tipo T_2, las siguientes expresiones:

  ```
  ref->attr
  ```

  y

  ```
  ref.attr
  ```

  se pueden utilizar para acceder al atributo `attr` referenciado por `ref`. Ambas expresiones son de tipo T_2. Los operadores `->` y `.` son intercambiables en OQL.

- OQL no obedece a los especificadores de acceso de C++ `public`, `private` y `protected`. Todos los miembros de una clase son accesibles, lo que permite a los usuarios finales acceder directamente a la implementación profunda de la clase.

  La expresión:

  ```
  Arquitectura_Redes._presupuesto
  ```

  accede al atributo `_presupuesto` de la instancia `Departamento` cuyo nombre es `Aquitectura_Redes`. El tipo de la expresión de consulta es el del atributo (`d_ULong`).

- Si un atributo hace referencia a un objeto de tipo `T`, la expresión de consulta es también del tipo referencia a `T`. La consulta:

  ```
  Arquitectura_Redes._gestor
  ```

  es una expresión de recorrido desde la instancia de `Departamento` a una instancia `Empleado`.

Acceso a los atributos y recorrido de relaciones ...

- Si un atributo es de tipo `T` y `T` tiene sus propios atributos embebidos, es posible aplicar los operadores `->` y `.` repetidamente para acceder a los atributos anidados. Por ejemplo:

  ```
  Arquitectura_Redes->_gestor->_direccion.ciudad
  ```

  es una expresión de consulta que accede al nombre de la ciudad donde vive el gestor (*manager*) del departamento de arquitectura de redes. La clase `Empleado` deriva de la clase `Persona` que contiene el atributo `_direccion` de tipo `Direccion`. La clase `Direccion` contiene un miembro `ciudad` que es de tipo `d_String`. Por tanto, el tipo de la expresión de consulta anterior es `d_String`.

Derreferencia

Supongamos que tenemos una expresión `ref` que es una referencia a un objeto de tipo `T`. El valor de la expresión `ref` es una referencia a un objeto y su representación cambia entre implementaciones, ya que cada una tiene una forma peculiar de implementar las referencias en la base de datos.

Supongamos que la aplicación necesita acceder a los valores de todos los atributos contenidos en el objeto referenciado; es ese caso, la expresión:

```
*ref
```

lleva a cabo la operación de derreferencia. El resultado de la operación es un objeto del tipo `T` y su valor es el valor de todos los atributos de la instancia `T` referenciada. Esta situación es equivalente a utilizar el operador `*` de C++ cuando `ref` es un puntero o una `d_Ref<T>`.

Referencias nulas

Una consulta puede contener una expresión que utiliza una referencia nula a un objeto.

- La palabra clave de OQL `nil` representa una referencia a objeto nula.
- Si una consulta intenta utilizar la referencia nula con los operadores `->` y `.` o con el operador de derreferencia `*`, el resultado no está definido. La palabra OQL `UNDEFINED` se utiliza para representar este valor.
- Es posible utilizar los operadores booleanos de la tabla siguiente para determinar si la expresión no está definida.

Operador	Valor
`is_defined(expr)`	*true* si la expresión está definida, *false* en cualquier otro caso.
`is_undefined(expr)`	*false* si la expresión está definida, *true* en cualquier otro caso.

El resultado de utilizar un valor indefinido como operando de una operación booleana (`=, !=, <, <=, >, >=`) es siempre *false*. En cualquier otra operación se produce una excepción en tiempo de ejecución.

Operaciones sobre objetos

El OQL permite invocar una función C++ definida en la clase de la aplicación. Los usuarios disponen de este modo de una interface que se corresponde directamente con la abstracción definida en el modelo de objetos.
Algunos entornos OQL carecen de esta funcionalidad.

- Dadas una expresión **ref** que referencia a una instancia T y una operación **oper** definida como función miembro sin argumentos del tipo T, cada una de las siguientes expresiones:

  ref->oper ref.oper ref->oper() ref.oper()

  aplican la operación **oper** sobre el objeto referenciado por **ref**.
  - Los paréntesis son opcionales cuando la función no tiene argumentos.
  - El tipo de la expresión depende del tipo de retorno de **oper** y el resultado es el mismo que se obtiene cuando se invoca a la función **oper**.
  - Si la función tiene un tipo de retorno **void**, el valor de la expresión OQL es **nil**.
- Si la operación requiere argumentos y si $\mathbf{expr}_i$ son expresiones cuyo tipo es compatible con los operandos correspondientes requeridos por **oper**, entonces las expresiones:

  $\mathrm{ref}\text{->}\mathrm{oper}(\mathrm{expr}_1,\ \mathrm{expr}_2,\ \ldots,\ \mathrm{expr}_n)$
  $\mathrm{ref}.\mathrm{oper}(\mathrm{expr}_1,\ \mathrm{expr}_2,\ \ldots,\ \mathrm{expr}_n)$

  aplican la operación **oper** sobre el objeto referenciado pasando los argumentos $\mathbf{expr}_i$.

Cambio de tipo de una expresión

Sea una expresión **expr** de tipo T_1 y un tipo T_2 derivado de T_1. La expresión:

$(\mathrm{T}_2)\ \mathrm{expr}$

indica que **expr** es una instancia de la clase T_2.

- Esta técnica es similar a usar dynamic_cast<T_2> en C++. Se utiliza para cambiar el *cast* de una referencia a objeto desde una clase base a una posible clase derivada.
- En tiempo de ejecución, el procesador de consultas valida si **expr** es realmente una instancia de T_2. En caso afirmativo, el resultado será una instancia de T_2, que puede entonces accederse como tal.

Construcción de objetos y estructuras

Es posible construir un objeto en una consulta,pero se tratará de una instancia transitoria que no se insertará en la base de datos. La expresión:

$\mathrm{T}(\mathrm{attr}_1{:}\mathrm{expr}_1,\ \mathrm{attr}_2{:}\mathrm{expr}_2,\ \ldots,\ \mathrm{attr}_n{:}\mathrm{expr}_n)$

define un nuevo objeto T cuyos atributos $\mathbf{attr}_i$ se inicializan mediante las expresiones $\mathbf{expr}_i$.

- No es necesario especificar todos los atributos; los que no se especifiquen tomarán los valores por omisión.
- Los atributos tampoco tienen por qué ser suministrados en el mismo orden que se declaran en C++ ya que no se invoca al constructor de la clase.
- El usuario puede inicializar un objeto que viola las restricciones de la clase, aunque siempre es posible crear una función que invoque al constructor de la clase y hacerla accesible al entorno de consulta.

Construcción de objetos y estructuras ...

La siguiente expresión OQL construye una instancia de `Persona`:

```
Persona (
  _fname: "Juan",
  _lname: "Lopez",
  _direccion: Direccion (
    calle: "Vall d'Uxo, 12-7",
    ciudad: "Valencia",
    provincia: "VA",
    codigo_postal: "46018"
  )
  _telefono: "963984545"
)
```

También es posible definir nuevas estructuras en OQL. Por ejemplo:

struct(attr$_1$:expr$_1$, attr$_2$:expr$_2$, ..., attr$_n$:expr$_n$)

define una estructura que contiene los atributos `attr`$_i$ inicializados con los valores `expr`$_i$.

El tipo del atributo viene determinado por el tipo de la expresión `expr`$_i$. La estructura definida sería similar a la definida en C:

```
struct {
  T1   attr1;
  T2   attr2;
  ...
  Tn   attrn;
} value = { expr1, expr2, ..., exprn };
```

Estas estructuras se definen dinámicamente y no se corresponden con ninguno de los tipos definidos en el lenguaje de la aplicación. Generalmente se utilizan en expresiones de consulta para producir resultados multivalor.

Expresiones literales atómicas

El ODMG usa el término *literal atómico* para referirse a los tipos de dominio simples.

Tipos de dominio primitivos

OQL soporta los siguientes tipos de dominio primitivos:

- Referencia a objeto nula: `nil`.
- Booleano: *true* y *false*.
- Enteros de 32 bits.
- Números en punto flotante, que consisten en una mantisa y un exponente opcional.
- Carácter, que se especifican mediante comillas simples. Por ejemplo `'D'`.
- Cadena de caracteres; secuencia de caracteres limitada por comillas simples o dobles. Por ejemplo: `"Hola mundo"`. OQL permite utilizar ambos tipos de comillas por compatibilidad con SQL. Sin embargo, esto conduce a cierto tipo de ambigüedad por lo que se utiliza el contexto de la consulta para determinar si el tipo es cadena de caracteres o carácter simple.

Expresiones aritméticas

En la tabla siguiente se muestran los operadores aritméticos unarios y binarios reconocidos por el OQL.

Operadores ariméticos unarios

Operador	Descripción
`+ num`	El valor de `num`.
`- num`	`num cambiado de signo`.
`abs num`	Valor absoluto de `num`.

Operadores aritméticos binarios

Operador	Descripción
$\texttt{num}_i$ `+` $\texttt{num}_j$	Suma.
$\texttt{num}_i$ `-` $\texttt{num}_j$	Substracción.
$\texttt{num}_i$ `*` $\texttt{num}_j$	Producto.
$\texttt{num}_i$ `/` $\texttt{num}_j$	División.
$\texttt{num}_i$ `mod` $\texttt{num}_j$	Módulo.

Operadores de comparación

Los operadores binarios de comparación devuelven un valor booleano. Los operadores se muestran en la tabla adjunta y se pueden aplicar sobre todos los dominios de tipo primitivos. Los operandos $\texttt{expr}_i$ y $\texttt{expr}_j$ deben ser del mismo tipo o de tipos compatibles (entero y punto flotante se consideran compatibles).

Operador	Descripción
$\texttt{expr}_i$ `=` $\texttt{expr}_j$	*True* si son iguales, *false* en otro caso.
$\texttt{expr}_i$ `!=` $\texttt{expr}_j$	*True* si son distintos, *false* en otro caso.
$\texttt{expr}_i$ `<` $\texttt{expr}_j$	*True* si $\texttt{expr}_i$ es menor que $\texttt{expr}_j$.
$\texttt{expr}_i$ `<=` $\texttt{expr}_j$	*True* si $\texttt{expr}_i$ es menor o igual que $\texttt{expr}_j$.
$\texttt{expr}_i$ `>` $\texttt{expr}_j$	*True* si $\texttt{expr}_i$ es mayor que $\texttt{expr}_j$.
$\texttt{expr}_i$ `>=` $\texttt{expr}_j$	*True* si $\texttt{expr}_i$ es mayor o igual que $\texttt{expr}_j$.

Operadores booleanos

Los operadores booleanos unarios y binarios reconocidos por OQL se muestran a continuación ordenados por prioridad decreciente. El operador `not` tiene la mayor prioridad que `and`, que a su vez tiene mayor prioridad que `or`. Por ejemplo, las siguientes operaciones son equivalentes:

```
not a and b or c    equivale a:  ((not a) and b) or c
a or not b and c    equivale a:  a or ((not b) and c)
```

Operador	Descripción
`not bool`	*True* si `bool` es *false*, `false` si `bool` es `true`.
$\texttt{bool}_i$ `and` $\texttt{bool}_j$	*True* si ambos son *true*, *false* en cualquier otro caso.
$\texttt{bool}_i$ `or` $\texttt{bool}_j$	*True* si uno es *true*, *false* en cualquier otro caso.

Expresiones cadena de caracteres

El OQL soporta operaciones sobre cadenas de caracteres. En la tabla adjunta se muestran las operaciones soportadas.

Operador	Resultado	Descripción
$\texttt{str}_i$ `||` $\texttt{str}_j$	string	Concatenación.
$\texttt{str}_i$ `+` $\texttt{str}_j$	string	Concatenación.
`char in str`	booleano	*True* si `char` está contenido en `str`.
`str[n]`	char	El carácter de `str` en posición `n+1`.
`str[m:n]`	string	La subcadena de `str` desde la posición `m+1` a `n+1`.
`str like` $\texttt{str}_j$	booleano	*True* si `str` coincide con el patrón definido en $\texttt{str}_j$.

El operador `like` devuelve *true* si el patrón de caracteres especificado existe en la cadena de caracteres. El patrón debe ser un literal de tipo cadena de caracteres que puede incluir los caracteres comodín mostrados en la tabla inferior. El caracter \ se puede poner delante de estos caracteres para anular su efecto como comodín.

Dada la cadena de caracteres **''un ODBMS ofrece muchas ventajas''**, almacenada en la variable `str`, la siguiente expresión se evaluaría como *true*:

```
str like "un ODBMS ofrece*ventaja?"
```

Comodín	Interpretación
`?` o `_`	Cualquier carácter.
`*` o `%`	Cualquier subcadena, incluyendo la vacia.

Colecciones

El OQL soporta colecciones de elementos de tipos arbitrarios. Las colecciones y su nombre en C++ se muestran en la tabla siguiente. Una colección puede ser un objeto persistente independiente, puede estar embebida como miembro de una clase o creada mediante una expresión OQL.

Colección OQL	Nombre C++	Características
`set`	`d_Set<T>`	Desordenada sin duplicados.
`bag`	`d_Bag<T>`	Desordenada con duplicados.
`list`	`d_List<T>`	Ordenada con duplicados.
`array`	`d_Varray<T>`	Ordenada con duplicados.
`dictionary`	`d_Dictionary<K,V>`	Desordenada sin duplicados.

Una colección literal puede crearse en una consulta. En este caso, es transitoria, carece de identidad y no puede almacenarse en la base de datos.

Colecciones ...

- **set**: Dadas las expresiones $\texttt{expr}_1$, $\texttt{expr}_2$, ..., $\texttt{expr}_n$ de tipo `T`, la expresión:

  $\texttt{set}(\texttt{expr}_1,\ \texttt{expr}_2,\ \ldots,\ \texttt{expr}_n)$

  define un `set` que contiene a estos elementos.

- **bag**: Dadas las expresiones $\texttt{expr}_1$, $\texttt{expr}_2$, ..., $\texttt{expr}_n$ de tipo `T`, la expresión:

  $\texttt{bag}(\texttt{expr}_1,\ \texttt{expr}_2,\ \ldots,\ \texttt{expr}_n)$

  define un `bag` que contiene a estos elementos.

- **list**: Dadas las expresiones $\texttt{expr}_1$, $\texttt{expr}_2$, ..., $\texttt{expr}_n$ de tipo `T`, la dos expresiones:

  $\texttt{list}(\texttt{expr}_1,\ \texttt{expr}_2,\ \ldots,\ \texttt{expr}_n)$
  $(\texttt{expr}_1,\ \texttt{expr}_2,\ \ldots,\ \texttt{expr}_n)$

  definen una `list` que contiene a estos elementos.
  Si `min` y `max` son dos expresiones de tipo entero o carácter y suponiendo que `min < max`, entonces las siguientes dos expresiones:

  $\texttt{list}(\text{min}\ ..\ \text{max})$
  $(\text{min}\ ..\ \text{max})$

  definen una `list` que contiene los siguientes elementos:

  $\texttt{list}(\text{min},\ \text{min}+1,\ \ldots,\ \text{max}-1,\ \text{max})$

- **array**: Dadas las expresiones $\texttt{expr}_1$, $\texttt{expr}_2$, ..., $\texttt{expr}_n$ de tipo `T`, la expresión:

  $\texttt{array}(\texttt{expr}_1,\ \texttt{expr}_2,\ \ldots,\ \texttt{expr}_n)$

  definen una `array` que contiene a estos elementos.

Operaciones comunes sobre colecciones

Es posible llevar a cabo diversas operaciones sobre una colección, algunas de la cuales tiene que ver sobre su cardinalidad (el número de elementos que constituyen la colección).

- Una colección sin elementos se conoce como *conjunto vacio* o *conjunto nulo*.
- Las colecciones que contienen un sólo elemento se conocen con el nombre de *singuletón*.
- En la lista de operaciones sobre colecciones mostrada en la tabla, `coll` representa una expresión de algún tipo colección que contiene elementos `expr` de tipo `T`.
- La operación `element` provoca una excepción si el operando no es un *singuletón*.

Expresión	Resultado	Descripción
`exist(coll)`	booleano	*True* si `coll` contiene al menos un elemento.
`unique(coll)`	booleano	*True* si `coll` contiene un sólo elemento (singuletón).
`element(coll)`	instancia `T`	El único elemento `expr` en `coll` si `coll` es un *singuletón*
`expr in coll`	booleano	*True* si `expr` es un elemento de `coll`.
`distinct(coll)`		Una colección con los elementos duplicados de `coll` eliminados.
`count(coll)`	entero	El número de elementos de `coll`

Operaciones comunes sobre colecciones ...

Algunos ejemplos son:

- Determinar si el empleado más listo de la empresa trabaja en el departamento de arquitectura de redes:

```
masListo in Arquitectura_Redes.empleados
```

  (recordemos que ambos eran objetos con nombre).

- Las siguientes dos consultas utilizan la operación `count`:

```
count(Deptos)
count(Arquitectura_Redes.empleados)
```

- Cuando se aplica la operación `distinct` sobre un `set<T>` o un `bag<T>`, el resultado es de tipo `set<T>`. Cuando se aplica el operador sobre un conjunto ordenado, como `list<T>` o `array<T>`, el resultado es del mismo tipo que el operando. El orden se preserva pero los elementos duplicados se eliminan:

```
distinct( list(1,2,3,4,5,4,3,2,1) )
```

  da como resultado:

```
list(1,2,3,4,5)
```


Operadores agregados

Los operadores agregados se puede aplicar sobre aquellas colecciones que contienen elementos de tipo numérico (*integer* y *float*). Las operaciones soportadas se indican en la tabla:

Expresión	Resultado	Descripción
`min(coll)`	instancia `T`	El valor mínimo en `coll`.
`max(coll)`	instancia `T`	El valor máximo en `coll`.
`sum(coll)`	instancia `T`	La suma de los elementos de `coll`.
`avg(coll)`	*float*	La media de los valores de `coll`.

Operaciones específicas de `set` y `bag`

- **Operaciones binarias**: OQL soporta la operaciones sobre conjuntos unión, intersección y diferencia. Dadas las expresiones $\texttt{coll}_1$ y $\texttt{coll}_2$ de tipo `set<T>` o `bag<T>`, las operaciones que se pueden realizar se muestran en la tabla:

Operación	Descripción
$\texttt{coll}_1$ `union` $\texttt{coll}_2$	Unión de los conjuntos $\texttt{coll}_1$ y $\texttt{coll}_2$.
$\texttt{coll}_1$ `intersect` $\texttt{coll}_2$	Intersección de $\texttt{coll}_1$ y $\texttt{coll}_2$.
$\texttt{coll}_1$ `except` $\texttt{coll}_2$	La diferencia $\texttt{coll}_1$ - $\texttt{coll}_2$.

  - Si una de las colecciones es un `bag` y la otra un `set`, el `set` se convierte previamente a `bag` y el resultado es un `bag`.
  - Si ambos son de tipo `set` el resultado es también un `set` y del mismo modo, si ambos son `bag` el resultado es `bag`.

Como ejemplo, consideremos las siguientes expresiones:

```
set(1,2,3) union set(3,4,5)          → set(1,2,3,4,5)
set(1,2,3) union bag(3,4,5)          → bag(1,2,3,4,5)
bag(1,2,2,3,4) except bag(1,2,3,4) → bag(2)
bag(1,2,2,3,4,4) intersect
bag(1,1,2,4,4,4)                     → bag(1,2,4,4)
```


Operaciones específicas de `set` y `bag` ...

- **Operaciones `subset` y `superset`**: Dadas dos expresiones $\texttt{coll}_1$ y $\texttt{coll}_2$ de tipo `set<T>` o `bag<T>`, es posible determinar si uno de ellos es un subconjunto o superconjunto del otro. Las operaciones de la tabla producen un resultado booleano indicando si la expresión es *true* o *false*.

Expresión	Descripción
$\texttt{coll}_1$ `<` $\texttt{coll}_2$	*True* si $\texttt{coll}_1$ es un subconjunto propio de $\texttt{coll}_2$.
$\texttt{coll}_1$ `<=` $\texttt{coll}_2$	*True* si $\texttt{coll}_1$ es un subconjunto de $\texttt{coll}_2$.
$\texttt{coll}_1$ `>` $\texttt{coll}_2$	*True* si $\texttt{coll}_1$ es un superconjunto propio de $\texttt{coll}_2$.
$\texttt{coll}_1$ `>=` $\texttt{coll}_2$	*True* si $\texttt{coll}_1$ es un superconjunto de $\texttt{coll}_2$.

Las siguientes expresiones producen el resultado booleano mostrado:

```
set(1,2) < set(1,2)    → false
set(1,2) < set(1,2,3)  → true
set(1,2) <= set(1,2)   → true
bag(1,2,3) > bag(1,2)  → true
bag(1,2,3) > bag(5,6)  → false
```

Operaciones específicas de colecciones indexadas (`list` y `array`)

En la tabla se muestran las operaciones aplicables. Se asume que $\texttt{coll}_1$, $\texttt{coll}_2$ y $\texttt{coll}_3$ son expresiones que denotan colecciones indexadas (`list<T>` y `array<T>`) que contienen elementos de tipo `T` y que `i` y `j` son números enteros.

Expresión	Resultado	Descripción
`first(`$\texttt{coll}_1$`)`	instancia `T`	El primer elemento de $\texttt{coll}_1$.
`last(`$\texttt{coll}_1$`)`	instancia `T`	El último elemento de $\texttt{coll}_1$.
$\texttt{coll}_1$`[i]`	instancia `T`	Elemento de $\texttt{coll}_1$ en posición i.
$\texttt{coll}_1$`[i:j]`	$\texttt{coll}_3$	La subcolección de $\texttt{coll}_1$ que empieza en i y acaba en j.
$\texttt{coll}_1$ `+` $\texttt{coll}_2$	$\texttt{coll}_3$	La concetenación de $\texttt{coll}_1$ y $\texttt{coll}_2$.

Operaciones sobre diccionarios

Sea una colección `dict` de tipo diccionario con clave de tipo $\texttt{T}_1$ y valor de tipo $\texttt{T}_2$ y una expresión `expr` de tipo $\texttt{T}_1$, entonces la expresión:

```
dict[expr]
```

es una expresión de tipo $\texttt{T}_2$ con el valor que está asociado con `expr` en `dict`.

Funciones

Sea una función `func` que retorna un tipo `T` y que requiere cero o más argumentos de tipo $\texttt{T}_i$ y una serie de expresiones $\texttt{expr}_i$ de tipos $\texttt{T}_i$ de tipos compatibles, entonces:

`func()` o `func(`$\texttt{expr}_1$`, `$\texttt{expr}_2$`, ..., `$\texttt{expr}_n$`)`

es una expresión de tipo `T` que invoca a la función `func`.

- La habilidad del OQL para invocar funciones u operaciones sobre objetos lo convierte en un lenguaje muy extensible, ya que el entorno específico del modelo de objetos de la aplicación está disponible en el entorno de consulta.
- Una aplicación puede definir nuevas operaciones agregadas como `min`, `max` y `avg`. Por ejemplo, si una aplicación necesita calcular la mediana de un conjunto de enteros, la función puede escribirse en C++ y después enlazarse con el entorno OQL.

Predicados y consulta de colecciones

En la sección anterior se han introducido muchas de las expresiones de consulta y operadores del OQL. En esta sección veremos los predicados y su utilización en el filtrado de los elementos de una colección.

- La construcción `select ... from ... where` es familiar a los usuarios de SQL.
- El OQL proporciona una claúsula `select` similar pero con extensiones significativas que extienden en gran medida la capacidad del SQL.

Variables de iteración

- Las expresiones de consulta utilizan una *variable de iteración* cuando iteran sobre una colección para hacer referencia al elemento actual de la iteración.
- Una variable de iteración se declara en la claúsula *from* de una consulta *select*.
- Una variable de iteración tiene el mismo tipo que los elementos de la colección a la que se asocian.

Predicados

- Un *predicado* es una expresión booleana que limita los elementos resultado de otra expresión de consulta.
- Se utiliza para filtrar los elementos para los que el predicado no se cumple. La claúsula *where* de SQL y OQL contiene un predicado booleano.
- La variable iteradora se utiliza dentro del predicado para hacer referencia al elemento actual.

Cuantificación universal

La cuantificación universal es una expresión booleana que es cierta sólo si el predicado es cierto para todos los elementos de una colección.

Dadas una variable `var`, una colección `coll` y una expresión predicado `predicate`, la expresión booleana OQL:

```
for all var in coll : predicate
```

es cierta sólo si `predicate` es cierta en todos los elementos de `coll`.

Por ejemplo, la siguiente consulta determina si todos los empleados del departamento de arquitectura de redes tienen un salario mayor de 50.000 ptas:

```
for all e in Arquitectura_Redes->empleados :
    e->sueldo > 50000
```

En esta expresión de consulta:

- `e` es la variable iteradora utilizada en el predicado para hacer referencia al elemento actual.
- La variable `e` es del tipo `d_Ref<Empleado>` puesto que el tipo de `empleados` es `d_Set< d_Ref<Empleado> >`.
- Esta expresión se evaluaría como *false* tan pronto como se encontrara a un empleado con un salario inferior o igual a 50.000 Ptas.

Cuantificación existencial

La cuantificación existencial es una expresión booleana que es cierta si al menos un elemento de la colección cumple la condición expresada por el predicado.

Dadas una variable `var`, una colección `coll` y una expresión predicado `predicate`, la expresión booleana OQL:

```
exists var in coll : predicate
```

es cierta sólo si `predicate` es cierta para al menos un elemento de `coll`.

Por ejemplo, la siguiente consulta determina si algún departamento de la compañía tiene más de 250 empleados:

```
exists d in Deptos : count(d.empleados) > 250
```

- La implementación de esta expresión no necesita acceder a todos los elementos de la colección.
- La evaluación se detendrá y devolverá *true* tan pronto como encuentre el primer elemento para el cual el predicado es *true*.

Predicados compuestos

- El OQL puede utilizar variaciones de los cuantificadores universal y existencial comentados que utilizan los operadores de comparación (`=, !=, <, <=, >, >=`).
- Dadas una colección `coll` de elementos de tipo `T`, una expresión `expr` de tipo `T` y un operador de comparación $\otimes$ del conjunto (`=, !=, <, <=, >, >=`), son posibles los predicados compuestos mostrados en la tabla.

Predicado compuesto	Expresión equivalente
`expr` $\otimes$ `some coll`	`exists var in coll : expr` $\otimes$ `var`
`expr` $\otimes$ `any coll`	`exists var in coll : expr` $\otimes$ `var`
`expr` $\otimes$ `all coll`	`for all var in coll : expr` $\otimes$ `var`

Los predicados `some` y `any` son sinónimos y tienen un comportamiento equivalente. Los siguientes predicados compuestos evalúan como *true*:

```
20 = some list(10, 20, 30, 40)
100 < some list(2, 300)
"blue" = any array("red", "blue", "green")
100 > all set(25, 50, 70)
```

La claúsula `select ... from ... where`

Dados:

- Un conjunto de expresiones $\texttt{coll}_i$ de colecciones de elementos de tipos $\texttt{T}_i$
- Un conjunto de variables de iteración $\texttt{v}_i$
- Un predicado booleano `predicate`
- Una expresión de ordenación `ord` y un proyector `projection`

la siguiente expresión es una expresión de consulta:

```
select [ distinct ] projection
from coll1 as v1, coll2 as v2, ..., colln as vn
[ where predicate ]
[ order by  ord ]
```

(los elementos entre corchetes indican opcionalidad)

- El resultado de una expresión de consulta `select` es una colección de elementos cuyo tipo viene determinado por el proyector `projection`.
- El tipo mismo de la colección viene determinado por la forma de la consulta y por la presencia de las palabras claves `distinct` y `order by`.

Por ejemplo, la palabra clave `distinct` hace que se eliminen los elementos duplicados.

`order by`	`distinct`	Colección	Duplicados
Si	Si	`list`	Si
Si	No	`list`	No
No	Si	`set`	Si
No	No	`bag`	No

La claúsula `select ... from ... where ...`

La evaluación de una expresión de consulta `select` requiere los siguientes pasos conceptuales:

1. Se construye el producto cartesiano de las colecciones $\texttt{coll}_i$. Cada colección es tratada como un `bag` de valores $\texttt{T}_i$. El producto cartesiano resultante es un `bag` del siguiente tipo:

   $$\text{bag} < \textbf{struct}< \text{v}_1 : \text{T}_1 , \ \text{v}_2 : \text{T}_2 , \ \ldots , \ \text{v}_n : \text{T}_n) >$$

2. El producto cartesiano es filtrado eliminando aquellos elementos que evalúan *false* el predicado booleano `predicate` expresado en la claúsula `where` (si no existe claúsula `where` no se elimina ningún elemento).
3. Si está presente la claúsula `order by`, el resultado se transforma en una `list` que se ordena de acuerdo con la expresión de ordenación `ord`.
4. El proyector `projection` se aplica al resultado filtrado del apartado anterior.
5. Si la palabra clave `distinct` está presente, se eliminan los valores proyectados duplicados.

Algunas notas:

- La cardinalidad del producto cartesiano es el producto de las cardinalidades de cada colección. Este valor puede llegar a ser un número extremadamente grande.
- La implementación suele llevar a cabo algún tipo de filtrado previo antes de formar el producto cartesiano para reducir el número de elementos.
- En muchas implementaciones no se forma nunca el producto cartesiano aunque el resultado final es matemáticamente equivalente.

La claúsula `from`

La claúsula `from` especifica la colección que se debe utilizar para formar el producto cartesiano.

Existen algunas diferencias entre la claúsula `from` de SQL y la implementada por OQL. En este último:

- No se trata de símples nombres de *extent* similares al nombre de la tabla utilizado en SQL.
- En OQL se puede utilizar cualquier resultado de una consulta cuyo tipo sea una colección.

Entre los argumentos de la claúsula `from` de OQL se encuentran:

- Un *extent*.
- Una colección con nombre.
- Una colección literal.
- Atributos de tipo colección embebidos como miembros de una clase.
- Una colección independiente.
- Una colección devuelta como resultado de un método o función de la clase.
- El resultado de una operación de conjuntos binaria: `union`, `intersection` o `difference`.
- El resultado de una operación de subcolección o concatenación.
- Otra expresión de consulta `select` (subconsulta).

La claúsula `from` ...

Son muchas las diferencias en los conceptos y en la implementación entre SQL y OQL:

- En el SQL de una base de datos relacional, los componentes de una claúsula `from` son *tablas* compuestas por *tuplas*. Desde la prespectiva del C++ y del OQL, una *tupla* es un `struct` cuyos miembros son los correspondientes a cada columna de la tabla.
- En OQL, los tipos de elemento de una colección pueden ser cualquier tipo `T` soportado por el esquema. Esto proporciona mucha mayor flexibilidad: el resultado de un `select` puede servir de operando para cualquier expresión que espere una colección.
- El SQL se basa en el *cálculo relacional de tuplas* que requiere que el producto cartesiano sea de tipo tupla. El OQL se basa en el *cálculo relacional de dominios* y las componentes del producto cartesiano son dominios. Esta aproximación es más general y permite un mayor grado de flexibilidad.
- El argumento de la claúsula `from` de OQL es un conjunto de colecciones separadas por comas.
- Es posible asociar una variable de iteración con cada colección: el valor del elemento actual de iteración se puede utilizar en otras partes de la consulta.

Por ejemplo, la siguiente consulta usa la colección con nombre `Deptos` para acceder a los nombres de todos los departamentos de la compañía:

```
select d.nombre() from Deptos as d
```

que puede escribirse opcionalmente como:

```
select d.nombre() from Deptos d
```

La claúsula `from` ...

Un ejemplo adicional: La siguiente consulta obtiene el nombre y apellido de los hijos de todos los trabajadores del departamento de recursos humanos:

```
select c._fname, c._lname
from   Deptos d, d.empleado e, d._hijo c
where  d._nombre = "Recursos_Humanos"
```

- Esta consulta accede a todos los departamento cuyo nombre es `"Recurso Humanos"` empleando la variable de iteración `d`.
- Los empleados de dichos departamentos son referenciados mediante `e` y los hijos de cada uno de estos empleados son accedidos vía el atributo `_hijo`.
- La consulta navega directamente a través de los objetos relacionaddos estáticamente en la base de datos.
- No son necesarios, por tanto, operaciones de búsqueda y combinación entre los *extents* `Departamento` y `Empleado`.

En una base de datos relacional sería necesario especificar una condición de combinación (*join*) en la cláusula `where` para establecer la relación entre las tuplas de las tablas.

La siguiente expresión SQL es equivalente a la del ejemplo anterior:

```
select c._fname, c._lname
from   Departamento d, Empleado e, Hijo c
where  d.nombre="Recursos_Humanos" and
       d.deptID = e.deptID and
       c.parentID = e.empID
```


La claúsula `where`

- La claúsula `where` contiene un predicado booleano `predicate` que se evalúa lógicamente sobre cada elemento del producto cartesiano formado en la claúsula `from`.
- Las variables de iteración especificadas en la claúsula `from` se pueden utilizar en la expresión del predicado como referencia a los componentes de producto cartesiano.

La claúsula `select`

- Una consulta `select` devuelve una colección. El tipo de los elementos de esta colección viene determinado por el proyector `project` de la claúsula `select`.
- Conceptualmente, el paso de proyección en la evaluación de la consulta recibe como entrada la colección de elementos que resulta de aplicar las otras claúsulas de la consulta.
- Un elemento de la colección devuelta por una consulta `select` es una variable simple de un tipo determinado.

  El tipo puede ser cualquiera de los soportados por OQL:
  - Una referencia a objeto.
  - El valor de un objeto (incluyendo sus atributos).
  - Una `struct`.
  - Un tipo primitivo.
  - Una colección.

La claúsula `where` ...

Las siguientes consultas ilustran algunas de las posibilidades.

- Acceder al presupuesto de cada departamento:

```
select d._presupuesto
from   Deptos d
```

  Puesto que esta consulta implica a una sóla clase, la variable de iteración puede eliminarse sin ambigüedad:

```
select _presupuesto
from   Deptos
```

  La funcionalidad del OQL permite que una consulta como la anterior sea utilizada directamente como operando de otra expresión. Por ejemplo, la siguiente consulta obtiene el presupuesto total de todos los departamentos:

```
sum( select _presupuesto from Deptos )
```

- La siguiente consulta:

```
select d
from   Deptos d
where  d._presupuesto > 1000000
```

  es un `bag` de referencias a `Departamento`. Para acceder al nombre y presupuesto de cada departamento es necesario formar un resultado de tipo `struct`:

```
select struct(nombre: d->_nombre,
              presupuesto: d->_presupuesto)
from   Deptos d
where  d._presupuesto > 1000000
```


La claúsula `order by`

Los resultados de una consulta puede ordenarse. Por ejemplo, la siguiente consulta devuelve todos los empleados del departamento de arquitectura de redes ordenados desde el mejor pagado hasta el sueldo más bajo y alfabéticamente en el caso de que dos o más empleados tengan el mismo sueldo:

```
select   e._lname, e._fname
from     Arquitectura_Redes.empleados e
order by e._sueldo desc, e._lname asc, e._fname
```

Joins

Una condición de *join* es una expresión booleana que define ciertas restricciones entre los valores del producto cartesiano.

- Aquellos elementos para los que no se cumple la condición son eliminados del producto.
- La condición de *join* se especifica en la claúsula `where`.
- Las condiciones de *join* son el único mecanismo para establecer relaciones en SQL. El OQL también implementa este mecanismo.

Dada una colección con nombre, `Deptos`, y otra colección con nombre `Proyectos` que contiene todos los proyectos de la compañía, la siguiente consulta devuelve el nombre de todos los departamentos cuyo nombre es igual al de un proyecto de otro departamento:

```
select   distinct p.nombre, d._nombre
from     Deptos d, Proyectos p
where    d != p->departamento and
         d._nombre = p.nombre
order by p.nombre
```

Definición de nombres de consulta

- El OQL soporta composición de funciones de modo que es posible elaborar consultas que contiene operandos con grados de anidamiento arbitrarios.
- Una consulta larga puede llegar a ser compleja y para simplificar su uso en operaciones compuestas es posible asignarle un nombre.
- De este modo, las expresiones de consulta puede definirse una vez y utilizarse en otras expresiones de consulta.

Dado un indentificador `name` y una expresión de consulta `query_expr`, entonces:

```
define [query] name(x1, x2, ..., xn) as query_expr
```

es una *definición de una expresión de consulta* que asocia el nombre `name` con la consulta `query_expr` que utiliza las variables x_i.

Por ejemplo, la siguiente definición de consulta devuelve todos los empleados cuyo nombre es `"Juan"`:

```
define todos_los_juanes as
  select e
  from   Empleados e
  where  e._fname = "Juan"
```

El siguiente ejemplo más elaborado devuelve todos los empleados que tienen el mismo apellido y además viven en una determinada ciudad:

```
define mismo_apellido_misma_ciudad(a,c) as
  select e
  from   Empleados e
  where  e._lname = a and e._direccion.ciduda = c
```


Definición de nombres de consulta ...

No es posible redefinir el nombre de una definición de consulta variando el número de argumentos. El nombre utilizado en la definición debe cumplir algunas restricciones:

- No puede coincidir con el nombre asignado a un objeto con nombre de la base de datos.
- No puede coincidir con el nombre de una función, método de clase o clase.
- El nombre no se puede reutilizar ya que esto conduce a ambigüedades.

La definición de una consulta se almacena en la base de datos y permanece activa hasta que se reemplaza por una nueva definición de consulta con el mismo nombre o hasta que se borra explícitamente mediante el comando:

```
delete definition name
```

Subqueries

- Una expresión de consulta consiste en un operador con un conjunto de operandos.
- Cada operando puede ser a su vez una expresión de consulta arbitraria.
- Consecuentemente, cualquier operando de una consulta puede ser a su vez una subconsulta (*subquery*).

Por ejemplo, la siguiente definición de consulta:

```
define todos_los_juanes as
  select e
  from   Empleados e
  where  e._fname = "Juan"
```

puede utilizarse en la especificación de la siguiente consulta:

```
define sueldo_de_los_juanes as
  select   struct( nombre: j._fname,
                   apellido: j._lname,
                   sueldo: j.sueldo )
  from     todos_los_juanes j
  order by b.sueldo desc
```

Aprovechando estas definiciones, la siguiente definición devolvería el apellido del "Juan" mejor pagado:

```
define juan_mejor_pagado as
  first(sueldo_de_los_juanes).apellido
```


Entorno de ejecución OQL

- Cada implementación que soporta OQL proporciona una interface a C++ y una interface interactiva que generalmente difiere entre productos.
- Las consultas pueden introducirse directamente en un entorno de tipo *shell* interactivo o en un fichero que es posteriormente procesado por el programa de consulta.

Consultas sobre colecciones en C++

La clase *template* `d_Collection<T>` proporciona cuatro funciones para recuperar elementos que cumplen una determinada condición expresada mediante un predicado OQL.

- El predicado puede ser cualquier expresión de consulta booleana.
- La consulta se ejecuta como si se tratara de una consulta OQL completa que llevara a cabo un `select` en una colección y con una claúsula `where` que consiste en el predicado booleano.
- La palabra clave `this` se utiliza en el predicado para referirse al elemento actual de iteración, exactamente igual que si se tratara de una variable de iteración asociada a la colección en una claúsula `from`.

Consultas sobre colecciones en C++ ...

- La función `exists_element` devuelve el valor booleano *true* si existe un elemento para el cual se cumple el predicado de la consulta. La función `exists_element` es equivalente a la siguiente consulta OQL:

```
exists this in coll : predicate
```

  Para determinar si alguno de los empleados del departamento de arquitectura de redes tiene un sueldo superior a los 12.000.000 Ptas. El siguiente código obtiene el resultado:

```
d_Database db;   // base de datos abierta

int result = 0;

d_Ref<Departamento> depto =
   db.lookup_object("Arquitectura_Redes");

if ( !depto.is_null() ) {
  result = depto->empleados.exists_element(
    "this->sueldo > 12000000");
}
```

  La variable `result` tendrá el valor *true* si existe el empleado. La variable `this` hace referencia al elemento actual de la iteración.

- La función `select_element` devuelve el valor del primer elemento encontrado que satisface el predicado booleano. El siguiente código accede a la primera instancia `Empleado` cuyo salario es mayor de 12.000.000 Ptas. Suponemos que `depto` referencia a un departamento:

```
d_Ref<Empleado> emp;

emp = depto->empleados.select_element(
  "this->sueldo > 12000000");
```


Consultas sobre colecciones en C++ ...

- La función `query` pone en una colección proporcionada por la aplicación que la invoca todos los elementos que satisfacen el predicado. La función `query` sería equivalente a la siguiente consulta OQL:

```
select this
from   coll this
where  predicate
```

  La siguiente consulta devuelve un `set` que contiene todos los empleados del departamento de arquitectura de redes cuyo apellido es "Pérez":

```
d_Set< d_Ref<Empleado> > perez;

depto->empleados.query(perez,
  "this->lname = \"Pérez\"");
```

  Una vez que la consulta ha sido evaluada, se puede emplear un iterador para recorrer todas las referencias a `Empleado` del resultado.

- La función `select` devuelve un iterador de modo que la aplicación puede iterar sobre los elementos que satisfacen el predicado. El código anterior se podría reemplazar por el siguiente:

```
d_Iterator< d_Ref<Empleados> > iter;

iter = depto->select(perez,
  "this->lname = \"Pérez\"");
```

  El iterador puede entonces utilizarse para acceder a cada elemento en el resultado. Esta aproximación elimina la necesidad de formar explícitamente la colección con los resultados.

Consultas sobre `d_Extent<T>`

Supongamos que se ha definido un *extent* para la clase `Empleado`. El siguiente código devolverá todos las instancias de `Empleado` de los empleados que viven en Valencia:

```
d_Database db;   // Una base de datos abierta
d_Iterator< d_Ref<Empleado> > iter;
d_Ref<Empleado> emp;
d_Extent<Empleado> Empleados(db);

emps = Empleados.select(
  "this->direccion.ciudad = \"Valencia\"");

while ( iter.next(emp) ) {
  // Operacion sobre el empleado referenciado
}
```

Consultas a la base de datos en C++

Una aplicación C++ puede submitir una consulta OQL arbitraria a la base de datos:

- Este mecanismo se debe utilizar necesariamente cuando la consulta no se lleva a cabo sobre una única colección.
- La consulta se puede ejecutar múltiples veces con parámetros diferentes y se construye utilizando la clase `d_OQL_Query`.

Consultas a la base de datos en C++ ...

Las características de la consulta realizada son:

- Cuando la consulta se ejecuta inicialmente, la cadena de consulta es analizada, compilada y optimizada. En algunas implementaciones las llamadas subsecuentes para ejecutar la consulta con parámetros diferentes no incurren en el coste de estas etapas previas.
- Una instancia de `d_OQL_Query` contiene la especificación de la consulta. Una vez que se ha construido la consulta, se pasa como primer argumento de la función `d_oql_execute`. El segundo argumento de esta función *template* define el tipo de resultado esperado. Se trata de un argumento que referencia al parámetro en el cual se emplazará el resultado. Si el tipo del resultado no es correcto o si alguno de los parámetros usados en la construcción de la consulta tiene un tipo incorrecto, se provoca una excepción `d_Error_QueryParameterTypeInvalid`.
- La consulta debe construirse antes de poder ejecutarse. Una instancia de la clase `d_OQL_Query` se puede inicializar con una cadena de caracteres, un `d_String` u otro objeto `d_OQL_Query`. El operador `<<` se puede utilizar para añadir valores al final de la consulta.
- Una consulta puede tener parámetros cuyos valores es necesario proporcionar en el momento de realizar la consulta. Se denotan mediante una subcadena de la forma $\$i$.

Consultas a la base de datos en C++ ...

- La siguiente consulta devuelve el conjunto de empleados de la compañía que trabajan solos en un proyecto:

```
d_OQL_Query query("distinct(select e from Deptos d,
  d.proyectos p, p.miembros e
  where unique(p.miembros))");

d_Set< d_Ref<Empleado> > trabaja_solo;

d_oql_execute(query, trabaja_solo);
```

- La siguiente función imprime el nombre de todos los empleados con un apellido específico y que trabajan en un proyecto cuyo coste es menor de una determinada cantidad:

```
void print_nombre_empleado(long coste,
                           const char *apellido)
{
  d_Set< d_Ref<Empleado> > emps;
  d_Iterator < d_Ref<Empleado> > iter;
  d_Ref<Empleado> e;

  static d_OQL_Query q("select unique e from Deptos d,
     d.proyectos p, p.miembros e\
     where p.cost < $1 and e._apellido = $2");

  q << coste;
  q << apellido;
  d_oql_execute(q, emps);

  iter = emps.create_iterator();
  while ( iter.next(e) ) {
     cout << e->nombre() << endl;
  }
}
```


Consultas a la base de datos en C++ ...

- Cuando se invoca a `d_oql_execute`, la función no devuelve el control hasta que todos los elementos que satisfacen la condición son colocados en la instancia `emps` de `d_Set`.
- Si la consulta contiene un elevado número de elementos, puede producirse una sobrecarga innecesaria.

El ODMG permite utilizar un `d_Iterator` como segundo argumento de `d_oql_execute` para iterar sobre los elementos de la consulta sin necesidad de construir explícitamente el conjunto:

```
void print_nombre_empleado(long coste,
                           const char *apellido)
{
  d_Iterator < d_Ref<Empleado> > iter;
  d_Ref<Empleado> e;

  static d_OQL_Query q("select unique e from Deptos d,\
     d.proyectos p, p.miembros e\
     where p.cost < $1 and e._apellido = $2");

  q << coste;
  q << apellido;
  q_oql_execute(q, iter);

  while ( iter.next(e) ) {
     cout << e->nombre() << endl;
  }
}
```

- Utilizando esta función no es necesario construir la colección y el resultado de la consulta se transfiere a la aplicación por paquetes a medida que son necesarios.
- Si la aplicación decide interrumpir la iteración, no es necesario transferir hasta el resto de los elementos de la consulta, reduciendo el coste de procesado.