ODMG C++

C. Delobel

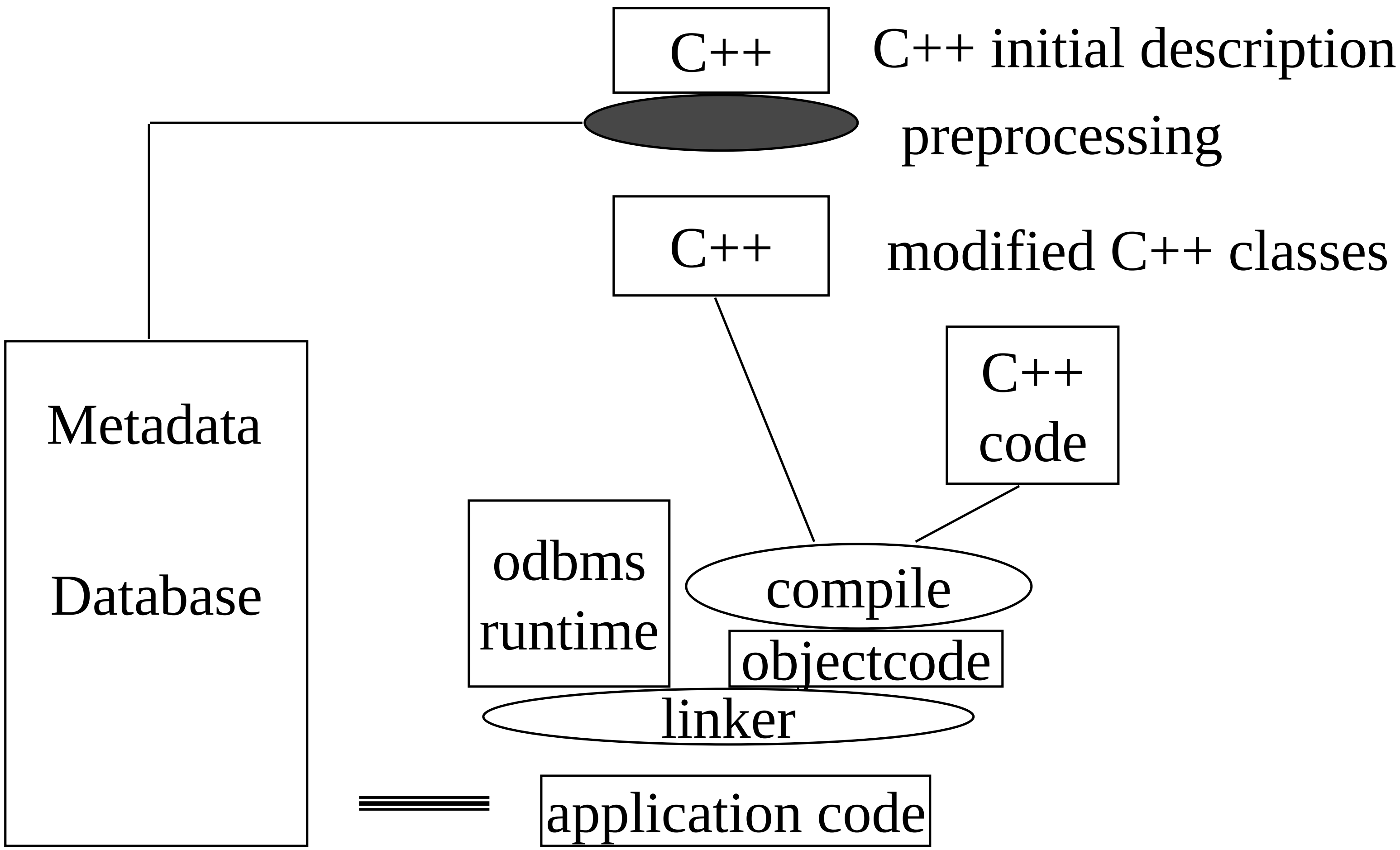
Design Principles

- Stick to the ODMG data model.
- Provide a C++ library where there is a unified type system across the programming language and the database.
- The ODL/C++ language contains extensions to describe C++ classes: persistent capable class, collections, relationships, naming capability, ...
- The OML/C++ is C++ compliant. May be compiled by a standard C++ compiler.

Benefits

- Extend programming language with persistent capability.
  - migrate programming applications in memory to database applications,
  - manipulating object databases with standard programming language.
- Provide to programming language database functionalities: query, transaction, security, indexing, ...

Preprocessing Phases



Persistence-capable Class

- To become persistent an object must be of a class derived from d_Object.
- For each persistence-capable class K, a twin class d_Ref<K> is defined.
- Instances of this class behave like C++ pointer.

Example

```
class Person:
  public d_Object {
  private:
    char* name;
    d_Ref<Person> spouse;
    d_Set<d_Ref<Person>>
       family;
   public:
    Person (char* name);
     ~Person();
};
```

```
class Person: virtual public o2_root {
  private:
    char* name;
    d_Ref<Person> spouse
    d_Set<d_Ref<Person>> family;
   public:
    Person (char* name);
     ~Person();
 protected:
virtual void o2_new();
virtual void o2_read();
virtual void o2_write();
.........
};
```

Example : database root

```
d_Session session;
d_Database base;
d_Transaction trans;
session.set_default_env;
session.begin(argc,argv);
base.open("origin");
trans.begin( );
d_Ref<Person> adam = new (&base) Person ("adam");
adam.create_persistent_root("adam", "d_Ref<Person>");
trans.commit;
base.close( );
session.end( );
```


C. Delobel

Example (cont.)

```
.....
session.set_default_env;
session.begin(argc,argv);
base.open("origin");
trans.begin( );
d_Ref<Person> adam("adam");
d_Ref<Person>  eve = new (base) Person ("eve");
adam -> spouse = eve;
d_Ref <Person> cain = new (&base) Person ("cain");
adam ->family.insert_element (cain);
trans.commit;
base.close( );
session.end( );
```

C++ ODL Types

- basic types
- predefined types
- structured types
- d_Ref
- collection
- relationships

Basic Types

Type name	Range	Description
d_Short	16 bits	signed integer
d_Long	32 bits	signed integer
d_UShort	16 bits	unisgned integer
d_ULong	16 bits	unsingned integer
d_Float	32 bits	single precision
d_Double	64 bits	double precision
d_Char	8 bits	ASCII
d_Octet	8 bits	no interpretation
d_Boolean	d_True,D_False	

Predefined Types

- d_String
- d_Interval
- d_Date
- d_Time
- ......

d_Ref

- Objects may refer to other objetcs through a smart pointer called a d_Ref. A d_Ref is parametrized by the type of the referenced object.
- d_Ref is defined as a class template :
  - template <class T> class d_Ref { .....}
- d_Ref <T> can be converted into d_Ref_Any, this is to support a reference to any type.

Collections

- There is a list of parametrized collection classes: d_Set, d_Bag, d_List, ...
- All these classes are subclass of d_Collection.
- The collections classes are delivered as templates:
  - template <class T> d_Collection : public d_Object{..}
  - template <class T> d_Set : piblic d_Collection{...}
- Examples:
  - d_List < d_String > >
  - d_Set < d_Ref < Person > >

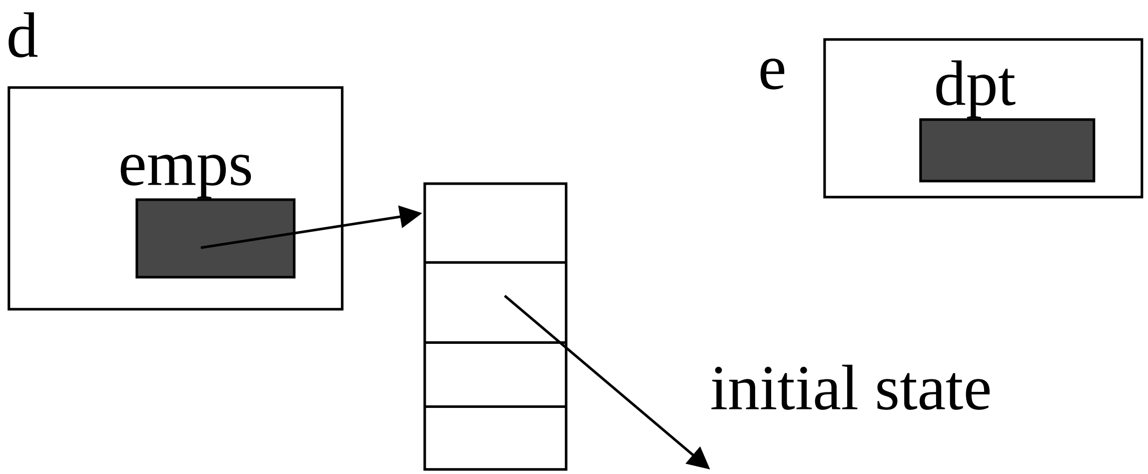
Relationships

- Provide a C++ interpretation of the relationships defined in the ODMG model.
- 1-1: d_Rel_Ref
- 1-n: d_Rel_Set or d_Rel_List and d_Rel_Ref
- m-n: d_Rel_Set or d_Rel_List and d_Rel_Set or d_Rel_List
- Relationships are implemented with class template.
- Creating, traversing and updating relationships are done with C++ OML.

Relationships: Example

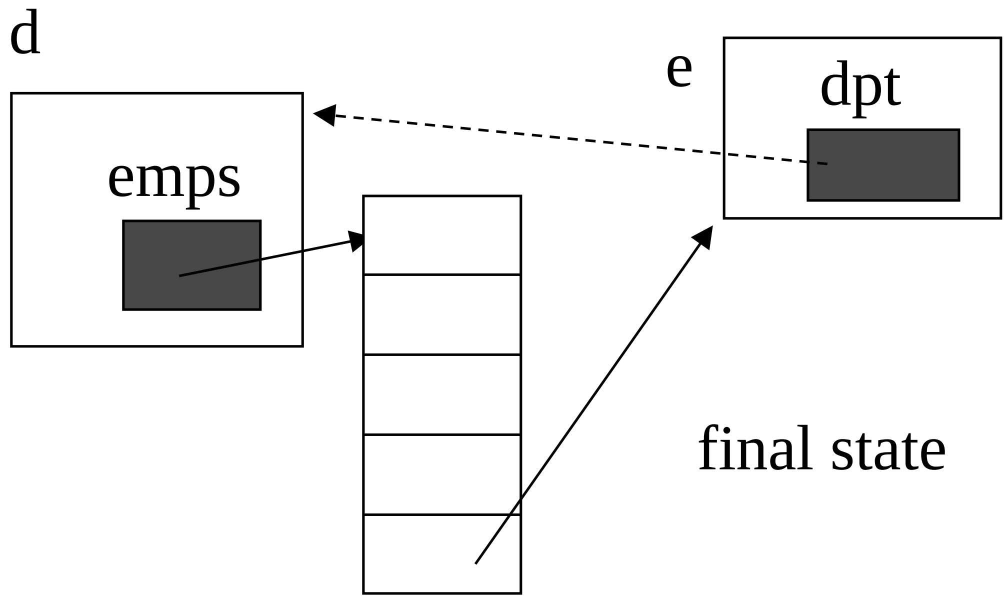
```
extern const char _dpt[], _emps[];

class Department {
    d_Rel_Set <Employee, _dpt> emps;
};

class Employee {
   d_Rel_Ref <Department, _emps> dpt;
};

const char _dpt[] = "dpt";
const char _emps[] = "emps";
```

Relationships (cont.)



Relationships (cont.)

d.emps.insert_element(&e); is equivalent to e.dpt = &d



Relationships Operations

- There exists a list of operations to insert, delete or update, the links: clear ( ), insert_element ( ), remove_element ( ), replace_element ( ).
- The principle is always the same: if you create a directional link by an instruction, in the database the inverse link is created.

Relationships: Templates

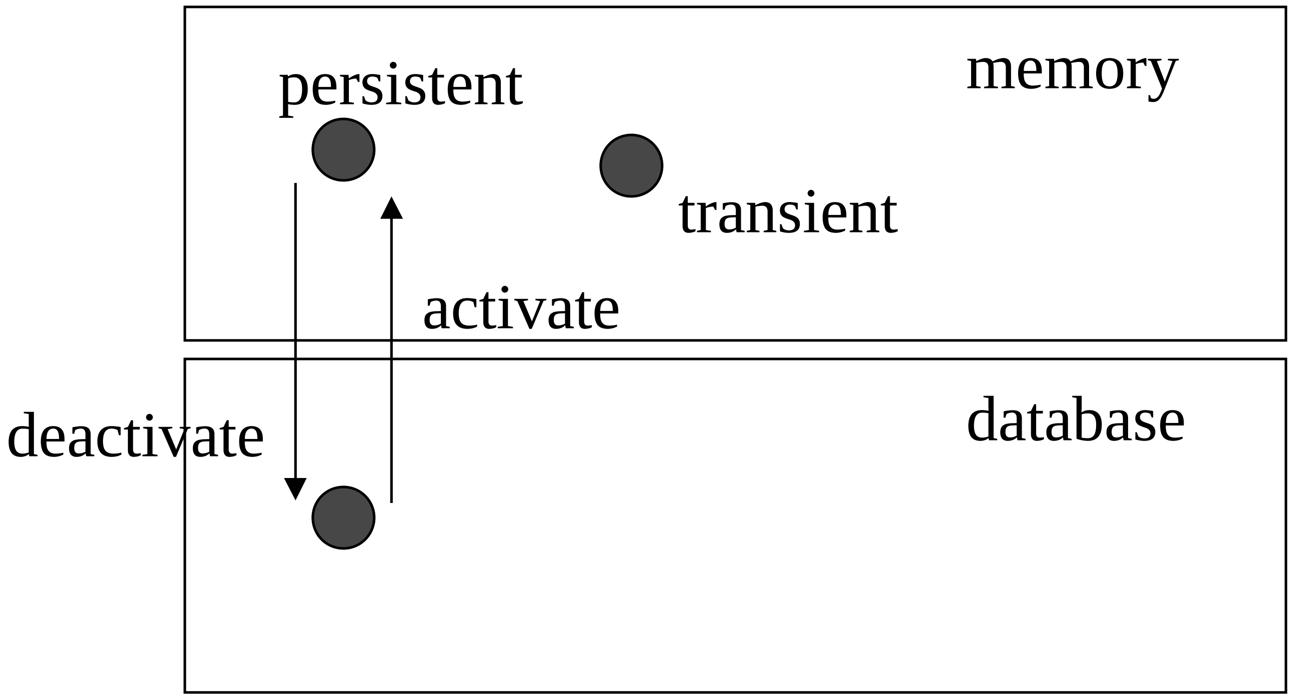
- Relationships are implemented with templates.

```
template < class T, const char* Member> class d_Rel_Ref :
                                        public d_Ref <T>
template < class T, const char* Member> class d_Rel_Set :
                                        public d_Set < d_Ref <T>>
template < class T, const char* Member> class d_Rel_List :
                                        public d_list < d_Ref <T>>
```

C++ OML

- Object creation
- Object deletion
- Object modification
- Object naming
- Manipulating collections
- C++/OQL

Database and Memory Cache



Object Creation

- Wether an object is transient or persistent is decided at creation time. The creation is done with the new operator.
- Persistent object creation
  - ➤ in the database new (mydatabase)
  - ➤ near another persistent object new (anobject)
- Example
  - ➤ d_Ref <Person> adam = new (mydatabase) Person;
  - ➤ d_Ref <Person> eve = new (adam) Person;

Object Deletion

- Deleting a persistent object is done with the operation delete_object ( ), which is a member function of the d_Ref class.
- The effect of the delete operation is to cut the link between the referenced object in memory and the corresponding object into the database. The definitive deletion is subject to transaction commit.

Object Modification

- When a persistent object is brought into memory, its initial state is identical to the database state.
- In memory the state is modidfied by updating properties or by invoking operations on it, as regular C++ operations.
- To inform the database that the state has been modified we need a special function mark_modified and the modifications are made at commit time.
- In some implementations, it is possible to detect that the state of an object has been modified.

Object Naming

- To retrieve objects in a database we need persistent names. These names are the root of the persistency.
- A name can identify a unique object or a collection of objects.
- The facility for naming is related with the class d_Database.

Class Database

```
class d_Databse {
        ........
        void open ( const char* database_name);
        void close ( );
        void set_object_name ( const d_Ref_Any& theObject
                                    const char* theName);
        d_Ref_Any lookup_object (const char* name);
        ........
}
```

Class Transaction

```
class d_Transaction {
        ......
        void begin ( );
        void commit ( );
        void abort ( );
        ........
}
```

Example of Object Naming

```
d_Database base;
d_Transaction trans;
base.open("origin");
trans.begin ( );
d_Ref < Person> adam;
        adam = new (&base) Person ("adam");
        base.set_object_name (adam, "adam");
trans.commit ( );
trans.begin ( );
        adam = base.lookup_object ("adam");
        cout<< adam-> name << endl;
trans.commit;
base.close ( );
```

Iterating on Collections

- To iterate on collections a class d_Iterator<T> is defined.
- The create_iterator function is defined in the class d_Collection.

```
template < class T> class d_Iterator {
        .......
        void reset ( );
        d_Boolean not_done ( );
        void advance ( );
        T get_element ( ) const;
        d_Boolean next ( T& objRef);
};
```

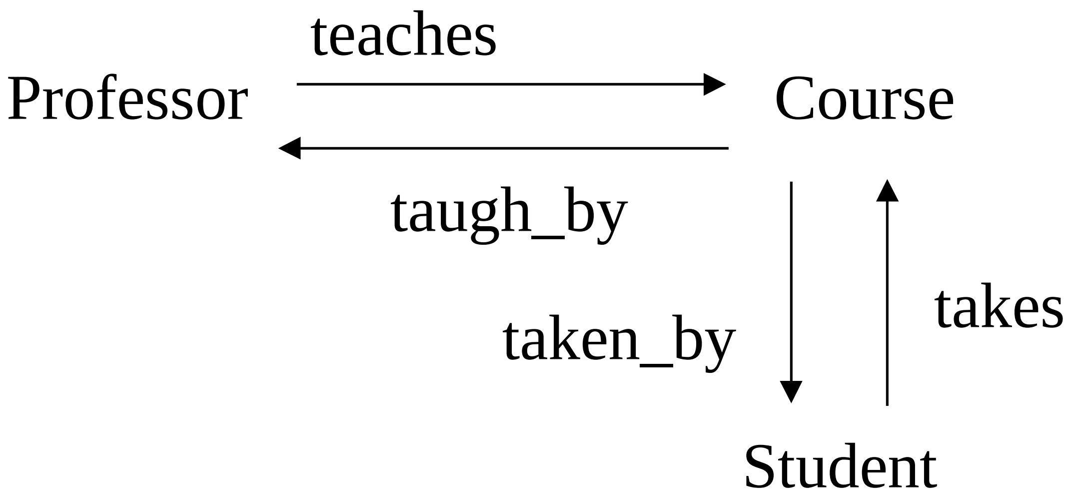
Example of Iteration

```
d_Iterator < d_Ref < Person> > iter =
                    adam->family.create_iterator ( );
d_Ref < Person >  s;
while ( iter. next (s) ) {
          //the next function checks the end of iteration,
          advancing the iterator and returning the current
          element  with the s variable//
.......
}
```

OQL Embedded in C++

- An OQL predicate may be applied to a collection through the query operation defined in the class d_Collection.
- Any OQL query may be issued which can take C++ arguments as inputs and returns a C++ object, or a C++ collections, or atomic data.

Example



Example (cont.)

```
d_Set < d_Ref < Student > > Students, mathematiciens, old;

Students.query ( mathematiciens,
                 "exists c in this.takes: c.subject = 'Math' ");

d_oql_execute ("select s
                from  s in $1
                where exists c in s.taken_by:
                          s.age > c.taugh_by.age"
                          << Students, old )
```

Example (cont.)

```
d_OQL_Query  query( "select s
                     from  s in $1
                     where exists c in s.taken_by:
                         s.age > c.taugh_by.age");
query = query << Students;
d_oql_execute ( query, old );
query = query << others;
d_oql_execute ( query, old);
```

Class d_OQL_Query

```
class d_OQL_Query {
        d_OQL_Query ( const char* s);
        ............
        template < class T >  friend
                        d_OQL_Query & operator <<
                        (d_OQL_Query &, const d_Collection & );
        ..........
        template < class T > friend int
                        d_oql_execute (d_OQL_Query &, T &);

}
```

Miscellaneous

- Class exception
- ODMG schema access class hierarchy :
  - d_Meta_Object
  - d_Type
  - d_Class, ...