Languages, Their Features and Limitations

5

Abstract

The implementation of persistence and class libraries depends on features provided by the programming language. Even though C, C++, C#, Objective-C and Java are quite similar in many aspects, they are significantly different in what they allow us to do about persistence and class libraries.

Keywords

Laguages • Persistence • Limitation • C • C++ • C# • Objective-C • Java

The first four chapters explored various approaches to implementing persistent objects and bi-directional associations, including tricks that make the persistence more efficient and make it easy to use. This chapter looks at individual languages and their features that may help to develop such systems.

Figure 5.1 sums up the features required for the basic styles of persistent objects:

A: When saving object-by-object, we can use normal allocation, but when saving entire blocks of memory, we have to replace the allocator.

B: Regardless of how we save the objects, we need to know the locations of all pointers.

C: When saving object-by-object, we need to know the type (the class) of the target object. Without this information, the algorithm could not collect all objects by following the pointers.

D: Members other than pointers are needed only when saving in the ASCII mode.

J. Soukup and P. Macháček, *Serialization and Persistent Objects*,
DOI 10.1007/978-3-642-39323-5_5,

special allocator	pointer location	target class	other members	virtual funct.	insert members	type of pesistence
		C	D	E		ASCII serialization
	B				F	binary images serialization
A						block of memory

Fig. 5.1 Features required for the basic three types of persistency

E: When collecting all objects by traversing the pointer links, we need virtual functions that detect the true type of each object. When saving blocks of memory, inheritance is transparent (irrelevant).

F: The library of intrusive associations requires coordinated insertions into participating classes, regardless of how the data is stored to disk.

5.1 Plain Old C Language

This section describes the internal design of Data Object Library (DOL). This library is a proof that we can implement a full fledged persistence in plain C, but we have to be smart about it and use a code generator with a lot of macros.

The first version of DOL was released in 1989, and it worked only in C. It included a library of intrusive associations such as described in Chap. 3 Today, DOL supports persistent C++ objects, but most of its internal design is still the original C code. Code Farms does not support the C version any more; if you wanted to use it, you would have to back several years to some earlier version. The new InCode library uses the same data structures with only improved parameterization and avoiding macros. The main difference from C++ is that we work with structures, not classes, and the generic functions which control the data structures are macros parametrized by the association ID, not class methods.

Figure 5.2 shows the information (chequered boxes) that cannot be obtained automatically, and must somehow be entered by the user. In C we do not have to worry about inheritance and virtual functions, there is no problem with writing our own allocator and we can insert members with macros. If we plan to implement persistence based on a block of memory, all we need are the pointer locations. Binary serialization also needs the type of target objects, and for ASCII serialization we need information about all the members, not just for the pointers. DOL provides all three types of persistence, and you can switch between them within one program run.

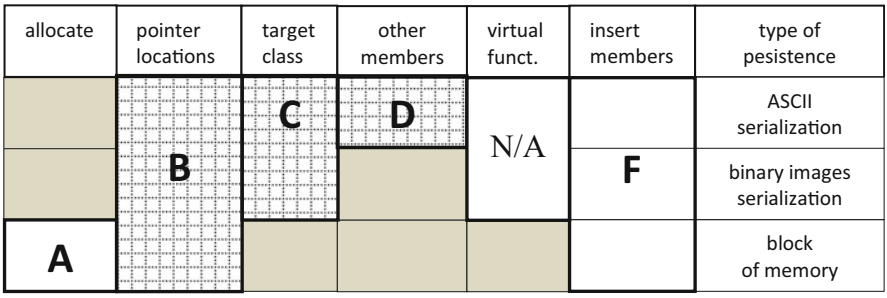

Fig. 5.2 In C, we do not need virtual functions, but pointer locations and target class are more difficult to obtain. Checkered information must be provided by the user; it cannot be obtained automatically. However, areas B and C are needed only when you are constructing the library. When you program with DOL, only D must be supplied by the user. Information from checkered boxes cannot be obtained automatically

OVERALL CONCEPT

DOL comes with an extensive library of generic intrusive data structures, and its persistence assumes the strategy which we still recommend and which was described in Chap. 3, namely that application classes have no pointer members; if there are any pointers that are a part of a data structure, they are transparently inserted by the library.

The generic library classes are implemented with macros, and, besides the types of participating classes, they are also parameterized by the data structure ID. For example, if you declare[1]

```
ZZ_ORG_SINGLE_AGGREGATE(books,Library,Book);
```

and then use it like this

```
struct Library *lib;
struct Book *bk;
...
ZZ_ADD(books,lib,bk);\
```

Note that the parameters are not only *Library* and *Book*, but also *books*.

the parameters are not only `Faculty` and `Student`, but also `students`.

The code generator called *zzprep* creates short segments of code that pull it all together. This preprocessor creates additional code. It does not modify the application code, so you use a debugger as usual.

When you program with DOL, you do not have to identify pointers because there are none in the application classes, at least not explicitly. Pointers are controlled by the library, and they were identified and described when the class was added to the library.

[1] This corresponds to the syntax we have been using in this book: Association SingleAggregate <Library,Book> books; Note that DOL includes the directory "test" with many programs that test all the features of this library. Tests that use ZZ_ORG_¼ are the C tests; C++ tests use ZZ_HYPER_¼ instead. For example, test0a.c is a C test.

Most users may never create new library classes. Because that is done only once, such registration of data structure and its pointers does not have to be particularly efficient or elegant. DOL The library keeps a *master*[2] file, where all the data structures (associations) and their pointers must be manually registered.

Listing 5.1 shows a section of this file, and what we have to do to register new data structure SINGLE_AGGREGATE, which works with pointers from 4 to 6: they are ZZp (from child to parent), ZZt (tail of the children ring) and ZZs (from child to its next sibling). In the C version of DOL, associations or data structures are called "organizations".

The last section[3] of the Listing 5.1 records methods of each association and the file where their source is stored.

Listing 5.1 Adding SINGLE_AGGREGATE to the master file where associations DOUBLE_LINK and DOUBLE_RING are already recorded

```
ZZorganization {
      0 DOUBLE_LINK                0  1
      1 DOUBLE_RING                2  3
      2 SINGLE_AGGREGATE           4  6
}

/* ind usedOn pointTo type ptrName, type=a means a pointer */
ZZpointer {
     0  0  2  a  ZZf     /* forward link */
     1  0  1  a  ZZr     /* reverse link */
     2  1  2  a  ZZf     /* forward ring */
     3  1  1  a  ZZb     /* reverse ring */
     4  2  1  a  ZZp     /* parent aggregate */
     5  1  2  a  ZZt     /* tail aggregate */
     6  2  2  a  ZZs     /* sibling aggregate */
}

/* function organization fileName */
ZZfunction {
    add       0  adddlink
    del       0  deldlink
    fwd       0  fwddlink
    rev       0  revdlink
    addTail   1  addtdrin
    next      1  nextdrin
    prev      1  prevdrin
    ...
    addTail   2  addtsagg
    next      2  nextsagg
    parent    2  parsagg
    ...
}
```

[2] In DOL, it is file macro/zzmaster.

[3] This section shows the concept, not the exact format.

For example, when the code generator reads these definition of these associations

```
ZZ_ORG_SINGLE_AGGREGATE(books,Library,Book);
ZZ_ORG_SINGLE_AGGREGATE(published,Author,Book);
```

it retrieves the names of all the structures and it gives them an index: 0 = Library, 1 = Book and 2 = Author. It also combines this information with the *master* file, makes a list of pointers that have to be inserted into these structures and creates file zzincl.h with ZZ_EXT_.. macros that insert the required pointers:

```
#define ZZ_EXT_Library \
    Book *_books_ZZt
#define ZZ_EXT_Book         \
    Library *_books_ZZp;   \
    Book *_books_ZZs;      \
    Author *_published_ZZp;\
    Book *_published_ZZs
#define ZZ_EXT_Author \
    Book *_published_ZZt
```

Note that this arrangement always positions all the pointers at the beginning of the object. When swizzling them or writing them to disk, we do not need a bitmap to identify their locations—we only need to know how many pointers we have at the beginning of the object.

The following code is an example of using DOL in a C application:

```
#include "zzincl.h" /* generated by the code generator */
struct Library {
      ZZ_EXT_Library;
      ... other members as usual
};
struct Author {
      ZZ_EXT_Author;
      ... other members as usual
};
struct Book {
      ZZ_EXT_Book;
      ... other members as usual
};

ZZ_ORG_SINGLE_AGGREGATE(books,Library,Book);
ZZ_ORG_SINGLE_AGGREGATE(published,Author,Book);

int main(){
      Library *lib; Author *auth; Book *bk1,*bk2;
      ZZ_PLAIN_ALLOC(Library,1,lib);
      ZZ_PLAIN_ALLOC(Book,1,bk1);
      ZZ_PLAIN_ALLOC(Book,1,bk2);
      ZZ_PLAIN_ALLOC(Author,1,auth);

      ZZ_ADD(books,lib,bk1);
      ZZ_ADD(books,lib,bk2);
      ZZ_ADD(published,auth,bk1);
      ...
}
#include "zzfunc.c" /* generated by the code generator */
```

where ZZ_PLAIN_ALLOC (T, 1, p) is the C equivalent of p=new T() in C++,
ZZ_PLAIN_ALLOC(T, n, p) is the C equivalent of p=new T[n] in C++,
ZZ_ADD(org, p1, p2) is the C equivalent of org.add (p1, p2) in C++.

The code generator prepared the definitions of macros ZZ_PLAIN_ALLOC and ZZ_ADD so that they are readily available through the include file, zzincl.h. The internal implementation is rather complex, but the following code samples show the general idea how it all works. In order to understand the concatenations (##), look above for the example of ZZ_EXT_.. statements:

```
/* macro from the library */
#define ZZ_PLAIN_ALLOC(TYPE,N,PTR) \
PTR=(TYPE*)calloc(sizeof(TYPE),N);

/* line generated specifically for this project */
#define ZZ_ADD_books ZZ_ADD_SINGLE_AGGREGATE

/* prepared by the code generator, ## concatenates */
#define ZZ_ADD(ID,PAR,CHI) \
ZZ_ADD##ID(ID,PAR,CHI,_##ID##_ZZp,_##ID##_ZZt, _##ID$$_ZZs)

/* macro from the library, parent and child */
#define ZZ_ADD_SINGLE_TRIANGLE(id,par,chi, \

                                parent,tail,sibling) \
     if((chi)->parent!=NULL || (chi)->sibling!=NULL){\
          ... error exit or do nothing
     }\
     else {\
          if((par)->tail==NULL){\
               (par)->tail=(chi);\
               (chi)->parent=(par);\
               (chi)->sibling=(chi);\
          }\
          else  {\
               (chi)->sibling=((par)->tail)->sibling;\
               ((par)->tail)->sibling=(chi);\
               (chi)->parent=(par);\
          }\
     }\
}
```

We do not need to know any details about *other members* (see box E in Fig. 5.2) if we want to run an ASCII serialization. We only need to know how to write and read back these members from the disk.

For that DOL has an elegant solution. For each class the user has to supply a ZZ_FORMAT statement. For example:

```
class Book {
     ZZ_EXT_Book
     int ISBN;
     float cost;
};
ZZ_FORMAT(Book,"%d %6.2f,ISBN,cost");
```

This statement contains enough information for the code generator to create the write and read functions that will always match, yet it has the flexibility to handle any interpretation of numbers and text.

5.2 C++ Language

C++ is excellent for implementing persistent data except for one thing: It does not support reflection. Since Chaps. 2 and 3 were built on C++ examples, we do not have to discuss the capabilities of the language itself, but we can look at the different approached used in the available C++ products.

Over the past 2 decades, most of the work on persistency was done in C++, and the history details of all these projects are interesting. Under the name of Organized C (orgc), DOL has been commercially distributed since 1989. Pointer swizzling at the page fault was first proposed by Wilson (1990). Singhal et al. (1992) reported on a university project called Texas, to which we could not find any references after 2000. Soukup (1994)[4] introduced memory blasting. Free[5] software (Knizhnik, POST++, 1999) is available for download. The Boost persistence was designed during 2002–2004 without its author being aware of the Code Farms libraries (DOL, PPF, InCode). Figure 5.4 shows the time progress of these projects.[6]

File mapping has been used in the **ObjectStore** line of products since the inception of the company as described in Lamb et al. (1991). Recently, Zikari (2010) confirmed the company still uses the same methodology. When using ObjectStore (c) PSE Pro for C++, the user must replace all calls to new() throughout the application.[7] The user does not have to identify pointer members, and neither published papers nor the documentation explain how PSE does it. Our guess is that the PSE code generator performs a partial syntax analysis of the application classes—essentially what you get through reflection in languages like Objective-C or Java. To verify this hypothesis is difficult because the PSE code generator produces a binary file, not a source you could examine visually.

Figure 5.3 shows the information (chequered boxes) that cannot be obtained automatically in C++ and must somehow be entered by the user.

Data Object Library (DOL) supports three styles of persistence (*binary serialization, ASCII serialization* and *memory blasting*). The serialization is automatic and supports schema migration. DOL combines persistence with an extensive library of data structures (associations) which include bi-directional associations not supported by STL—see Fig. 5.5. It provides a more extensive protection against pointer errors than Java, and all classes have iterators which allow to delete objects while iterating the containers. The total space for its executables (code generator and compiled library) is under 400 kB.

[4] pp. 386–392.

[5] POST++ comes in source from which all comments have been removed, and it is rather difficult and time consuming to figure out its inner workings.

[6] Code Farms were incorporated in 1988 and ObjectStore in 1989.

[7] Compare this to PPF which re-defines operator new() and makes this transparent.

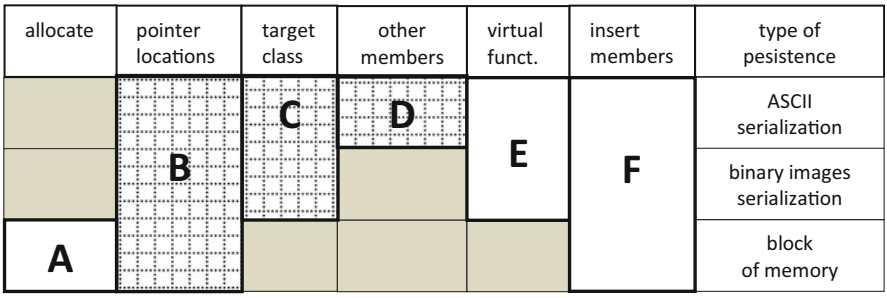

Fig. 5.3 In C++, virtual functions that we may need when inheritance is involved can be inserted with a macro. Otherwise the situation is similar to C, where the main problems are boxes B, C and D. Information from checkered boxes cannot be obtained automatically

year	**DOL**	**ObjSt.**	Texas	**PPF**	POST	Boost
1989	S					
1990	S	M				
1991	S	M	M			
1993	SB	M	M			
1997	SB	M	M	P		
1998	SB	M	M	P	M	
2000	SB	M		P	M	
2004	SB	M		P	M	X

Fig. 5.4 History of persistent systems, commercial projects in bold.Legend: S = automatic serialization, both binary and ASCII, M = memory mapping, B = memory blasting, P = persistent pointers, X = binary, ASCII and XML serialization. Information from checkered boxes cannot be obtained automatically

DOL assumes that application classes do not store any raw pointers and participate only in data structures (associations) from DOL. DOL includes some unusual classes, for example class **Pager** which stores nonstructural information such as text, pictures or tables of numbers in a separate file, and pages it to memory as needed. Class **Property** is useful in two situations. (1) When you may need, sometimes in the future, to add members to applications classes without adjusting the schema. (2) When some members are only sparsely used. For more details, see Sect. 4.2.

Persistent Pointer Factory (PPF) uses a completely different style of persistence, see Sect. 2.4.2. It is based on persistent pointers which page disk to memory on demand. Because these pointers store the disk address and not the memory address, they are persistent and do not require swizzling when moving the data to or from the disk. This library being written is proof that C++ persistence can be implemented in pure C++ without any code generation or using system specific

ring (singly and double linked; sort, merge, and split functions)
collection (singly and double linked; sort, merge, and split functions)
aggregate (singly and double linked; **OneToMany** association)
trees (singly and double linked)
name (variagle length string)
single and double **link** (pointer link, or **OneToOne** association)
LIFO and **FIFO** queues
reference (similar to Java reference)
array (array of object or array of pointers, also **binary heap**)
hash table (use default or your own hashing)
graphs (directed, not directed, singly or doubly linked)
ManyToMany (including two iterators)
type (essentially a form of reflection)
pager (persistent storage of large non-structural information, texts)
property (run-time expansion of objects by any number of named members)

Fig. 5.5 Persistent data structures supported by DOL

functions such as file mapping. It has been offered on the web for over a decade, but in spite of being elegant and compact,[8] it did not become popular, probably because it originally did not include any data structures, and programmers looking for persistence chose DOL with its extensive library.

While writing this book, we paired PPF with InCode library,[9] which is a modern library of bi-directional associations, but does not have persistence. The result is the **PPFIC library**. For the performance comparison with other libraries, see Chap. 7 (Benchmarks).

The main difference between PPF and POST or PSE is that PPF works with soft pointers which require a short arithmetic calculation on each pointers access, while PSE and POST use hard pointers which are swizzled when the page is loaded to memory.

Boost is an open source library of data structures which also includes serialization. The serialization requires too much user input to qualify, in our terms, to be considered an automatic persistence, yet it was recently proposed as a C++ standard. Because of that, and because of its massive use worldwide, we treat it as one of the serializations that are part of the language, such as Java serialization or C# serialization. For more on Boost, see Sect. 1.5.5.

[8] Based on C++ templates, total source 2700 lines including comments, executable 164 kB.
[9] See Sect. 3.1.3.

Just for your curiosity, let's look at how much the syntax of working with DOL improved when moving from C (Sect. 5.1) to C++ here:

```
#include "zzincl.h" /* generated by the code generator */
class Library {
    ZZ_EXT_Library;
    ... other members as usual
};
class Author {
    ZZ_EXT_Author;
    ... other members as usual
};
class Book {
    ZZ_EXT_Book;
    ... other members as usual
};

ZZ_HYPER_SINGLE_AGGREGATE(books,Library,Book);
ZZ_HYPER_SINGLE_AGGREGATE(published,Author,Book);

int main(){
   Library *lib; Author *auth; Book *bk1,*bk2;
   lib=new Library;
   bk1=new Book;
   bk2=new Book;
   auth=new Author;

   books.add(lib,bk1);
   books.add(lib,bk2);
   published.add(auth,bk1);
   ...
}
#include "zzfunc.c" /* generated by the code generator */
```

In file zzincl.h, code generator prepared ZZ_EXT_... statements the same way as it did in C, but it added the friend statements which allow the relevant interface classes to reach inside class Book. It also replaces operator new() depending on whether the persistence uses memory blasting or serialization. MB_ALLOC(Block) is a macro which not only allocates the object from special pages, but it also updates the corresponding bitmap with the location of pointers embedded in each Book.

```
#define ZZ_EXT_Book \
friend class ZZHbooks;\
friend class ZZHpublished;\
    Book * _published_ZZs;\
    Book * _book_ZZs;\
    Author * _published_ZZp;\
    Library * _books_ZZp;\
public:\
    void * operator new(size_t size){\
        if(memoryBlasting) return MB_ALLOC(Book);\
        else return calloc(size,1); /* serialization */\
    }\
    ...
```

Internally, the interface class is renamed, using the association name. For example
`ZZ_HYPER_SINGLE_AGGREGATE(books,Library,Book)` becomes `ZZHbooks`
`ZZ_HYPER_SINGLE_AGGREGATE(published,Author,Book)` becomes `ZZHpublished`
Internally, individual methods can either call the original C macro (STYLE 1), or be fully coded in C++ (STYLE 2):

```
// STYLE 1: new interface hiding the macro design
#define ZZ_HYPER_SINGLE_AGGREGATE(id,pType,cType) \
class ZZH##id { \
public: \
     void add(pType *p,cType *c){ ZZ_ADD(id,p,c); } \
     ... all the other methods \
} id;

// STYLE 2: true C++ code
#define ZZ_HYPER_SINGLE_AGGREGATE(id,pType,cType) \
class ZZH##id { \
typedef tail _##id##_ZZt; \
typedef sibling _##id##_ZZs; \
typedef parent _##id##_ZZp;\
public: \
     void add(pType *par,cType *chi){\
          if(chi->parent!=NULL || ch)->sibling!=NULL){\
               ... error exit or do nothing
          }\
          else {\
               if(par->tail==NULL){\
                    par->tail=chi;\
                    chi->parent=pa);\
                    chi->sibling=chi;\
               }\
               else {\
                    chi->sibling=(pa)->tail)->sibling;\
                    (par->tail)->sibling=ch);\
                    chi->parent=par;\
               }\
          }\
     }\
     ... other the other methods \
} id;
```

Note that the syntax in which application uses this interface can be set up in two ways. When it is as we just described, the application calls are

```
books.add(lib,bk);
published.add(auth,bk);
```

which is the style used in DOL.

If we set up the interface class like this

```
#define ZZ_HYPER_SINGLE_AGGREGATE(id,pType,cType) \
class id { \
    ... \
};
```

the application calls would be

```
books::add(lib,bk);
published::add(auth,bk);
```

5.3 Java Language

From the viewpoint of data structures and persistence, Java is significantly simpler than C++, and very much like C#:

- It does not have pointers, only references.
- A member can be a reference, but not an object (instance of some class).
- Multiple inheritance is not allowed.
- It has *reflection* which solves the problem with finding reference members.
- There is no equivalent of C macros.

Internally, references store object addresses, but their values are not available to the application programmer, and operator new() cannot be overloaded. From Fig. 5.6 we can conclude that implementing serialization should be easy, but persistence based on the block of memory would be difficult or impossible to design.

It's not surprising that Java has built-in serialization which saves the data in a special byte-encoded format; see Sect. 1.5.2 What is surprising is that the ObjectStore company now has PSE Pro for Java.

Blog (Weinreb 2007) describes how the original PSE Pro for Java was implemented, but the past tense is being used—perhaps meaning that the existing implementation is different: *The PSE Pro for Java had its own storage engine which is used just for object-level faulting with a specialized lightweight, small footprint, storage engine. However, it did not support concurrent access between separate Java processes*. The idea of injecting JVM instructions into Java class files is also mentioned.[10]

Besides Java built-in serialization—see Sect. 1.5.2, there are several systems that store Java objects in a database which is not the type of persistence we cover in this book. An example of such a system is Hibernate.

UMPLE (2012) is a model-programming technology which resembles the InCode library in that it represents associations as first class objects, includes a library of data structures (associations) and alternates between controlling them with a textual schema which is in the code or controlling them with the UML class diagram—see Sect. 3.5. It runs with Java, PHP and Ruby, but it does not support persistence.

[10] JVM—Java Virtual Machine; this is an equivalent of inserting machine code instructions into an object file.

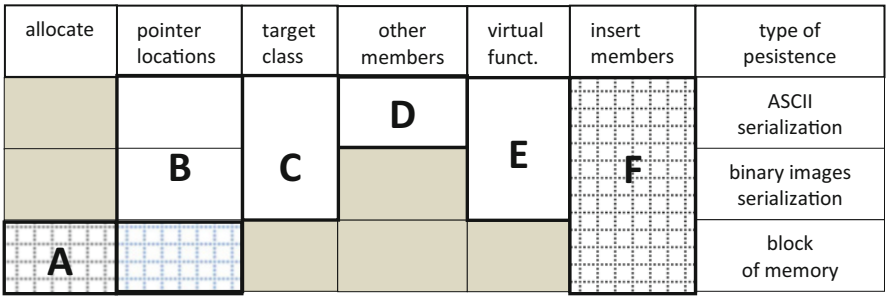

Fig. 5.6 C# and Java are similar. Their reflections allow one to identify references, their target types and other members, but you cannot overload operator new() or get the address stored inside the reference. Inserting members is difficult, there are no macros and instances cannot be inserted as members. Information from checkered boxes cannot be obtained automatically

5.4 C# Language

From the viewpoint of data structures and persistence, C# is significantly simpler than C++, and very much like Java—see Fig. 5.6:

- It does not have pointers, only references.
- A member can be a reference but not an object (instance of some class).[11]
- Multiple inheritance is not allowed.
- It has *reflection* which solves the problem with finding reference members.
- There is no equivalent of C macros.

Internally, references store object addresses, but their values are not available to the application programmer, and operator new() cannot be overloaded. From Fig. 5.6 we can conclude that implementing serialization should be easy except for the insertion of pointers required for intrusive data structures. However, persistence based on the block of memory would be difficult if not impossible to design.

C# has a **built-in serialization** which supports both binary and XML formats. The advantage of the XML format is that it can read the stored data even if the

[11] This is easiest to explain on a C++ example:

```
class A {...};
class B {
   A *ap; // corresponds to reference in Java
   A aa; // is not allowed in Java
};
```

Note that ap leads to an object allocated separately, but aa is allocated as a part of any B object.

structure of the serialized objects are changed, for example if we add or remove members from some classes. The disadvantage of the XML format is a larger size of the data file. For more details see Sect. 1.5.3.

The **C# library of associations (Osterby** 2000) is not persistent, but it is interesting because it inserts the references that form the association without using a code generator or Aspects. It uses runtime type instantiation, which is not available in Java. This was a university research project which is not active any more. The C# version of Java Hibernate is called NHibernate. Commercial product DevXPress also provides persistency which is using a database.

5.5 Objective-C Language

Of all the languages discussed in this book, Objective-C is the most suitable for building persistence.[12] It has all the advantages of C and C++ including access to addresses of objects and pointers, but it also has reflection which can identify pointers and their locations without any user input.

There are no chequered boxes in Fig. 5.7 because all the steps are easy. You cannot replace operator new() but you can code your own method *alloc*. The reflection gives you all the members, and tells you what the types are. You do not need information about the pointer target type, because each object, in this case the target object, can tell you its type. All methods in Objective-C are virtual, so there is no special problem with virtual methods. You cannot insert an object instance as a member, but a member can be a struct—which is all we need when we built intrusive data structures.

We know only one implementation of persistent objects for Objective-C, the built-in serialization called *Archiving,* which generates XML, ASCII or binary output as we explained in Sect. 1.5.4. To support archiving, a class has to implement the NSCoding protocol which has methods to encode (to archive) and decode (to unarchive) instances of that class which, as we said at the beginning of this book, is something we want to eliminate. Also quoting[13] the Apple documentation:

> *Cocoa archives can hold Objective-C objects, scalars, arrays, structures and strings. They do not hold types whose implementation varies across platforms, such as union, void*, function pointers, and long chains of pointers.*

If it cannot handle long chains of pointers it cannot handle intrusive data structures.

[12] It is also a young language only about 10 years old which has little in common with Objective-C from the 1990s.

[13] http://developer.apple.com/library/mac/#documentation/cocoa/conceptual/Archiving/Articles/archives.html

special allocator	pointer location	target class	other members	virtual funct.	insert members	type of pesistence
		C	D	E		ASCII serialization
	B				F	binary images serialization
A						block of memory

Fig. 5.7 Of all the languages discussed in this book, Objective-C is the most suitable for building persistence—in addition to all the useful features of C++, it also supports reflection. Its weak point is the NextStep (NS) library. Note that all the information can be obtained automatically (there are no checkered boxes)

The following chapter (Chap. 6) will take us through the implementation of fully automatic persistence for Objective-C, with code examples in Objective-C only. Objective-C syntax is quite different from the C++ or Java, and not all readers will be familiar with it. The remaining part of this chapter will introduce Objective-C syntax, just enough that you should be able to read the code samples in Chap. 6. It will also show some algorithms that are conceptually different from C++.

You can compile Objective-C programs with gcc under Windows or Linux, or with iOS on any Apple hardware, e.g. Mac. Look at Listing 5.2 for the comparison of the C++ and Objective-C syntax.

Normally all application classes are derived, directly or indirectly, from class NSObject. For example, in Listing 5.2, class Publication might have been derived from it:

```
@interface Publication : NSObject
```

Reserved keyword *id*, usually written as (id) means "pointer to any Objective-C object", not just "pointer to any object derived from NSObject". The advantage of deriving object from NSObject is that NSObject implements support methods required by runtime. In Listing 5.2, method *init* returns (id).

Listing 5.2 Comparing syntax of C++ and Objective-C

C++ file: Book.h

```
class Author;

class Book : public Publication {
  int pages;
public:
  char *title;
  Book *next;
  static int ID; // class ID
  Book();
  int getPages();
  void setBook(int pg,char *tit);
  static int getID();
};
```

C++ file: Book.cpp

```
int Book::ID=13;
Book::Book(){pages=0;title=NULL;}
int Book::getPages(){
  return pages;
}
void Book::setBook(
              int pg,char *tit){
  pages=pg;
  title=tit;
}
int Book::getID(){return ID; }
```

C++ file: main.cpp

```
int main(){
  Book *bk1,*bk; int sz;
  char *t="C++ Manual";
  bk1=new Book(;
  bk1->setBook(120,t);
 sz=Book:getID();
  // ...more books form a chain
  for(bk=bk1; bk!=NULL;
                bk=bk->next){
    printf("%s\n",bk->title);
  }
  return 0;
}
```

Objective-C file: Book.h

```
@class Author;

@interface Book : Publication
{
@private
  int pages;
@public
  char *title;
  Book *next;
}
- (id) init;
- (int) getPages;
- (void) setBook: (int) pg
           title: (char*) tit;
+ (int) getID; //size of object
@end
```

Objective_C file: Book.m

```
@implementation Book
static int ID=13; // class ID
-(id) init {
  self=[super init];
  pages=0; title=NULL;
  return self;
}
-(int) getPages {
  return pages;
}
-(void) setBook: (int) pg
           title: (char*) tit {
  pages=pg;
  title=tit;
}
+ (int) getID {return ID;}
@end
```

Objective-C file: main.m

```
int main(){
  Book *bk1,*bk; int sz;
  char *t="C++Manual";
  bk1=[[Book alloc] init];
  [bk1 setBook: 120 title: t];
  sz=[Book getID];
  // ...more books form a chain
  for(bk=bk1; bk!=nil;
                bk=bk->next){
    printf("%s\n",bk->title);
  }
  return 0;
}
```

The main difference from C++ is that classes are not abstract, untouchable entities: they are objects. You can examine them during the program run or pass them as function or method parameters. Keyword Class (without *) means “pointer

to a class". This concept opens magical possibilities about which you could not dream in C++, for example:

```
Book *bk=[[Book alloc] init];
(id) v=bk;
Class cls=[v class];
const char* className = class_getName(cls);
NSLog(@"yourObject is a: %s", className);
```

An instance of a class cannot be used as a member of another class. This is similar to Java; however, unlike Java, Objective-C allows members which are instances of a structure. Compare the rules with this C++ code:

```
class A {...};
typedef struct myStruct {
    void *mask;
    int maskSize;
} myStruct;

class B {
   A a;            // not allowed in Objective-C
   myStruct ms;    // works in Objective-C
   A *ap;          // works in Objective-C
};
```

For any Objective-C object, its hidden pointer (first 4 or 8 bytes) points to the class object. Keyword *self*, when used in an object method, has the same meaning as *this* in C++. For example, see the return of method *init* in Fig. 5.1. However, inside a class method[14] it represents theclass.

Unlike in C++, Objective-C programmers often use C-style, free floating functions. The following example demonstrates this style of design where *self* represents the class. This is a situation where we want to add the same simple method

createMask to every application class and let all these methods call one C-style function which actually does the job. The example is shown for application class Book:

[14] Method that would be static in C++ and in Objective-C starts with '+'.

Listing 5.3 C-style function implementing multiple methods, with self representing a class, not an object

```
(id) createMaskGeneral(Class cls);

@interface Book : NSObject
  //...
  + (void) createMask; // one line added to every application class[15]
  + (void) checkMask;
@end

// C-style function which actually creates the mask
void *createMaskGeneral(Class cls){
    (id) obj=[[cls alloc] init];
     //...more code
    return obj;
}

@implementation Book
  static Book *mask=nil;
  + (void) createMask {
      mask=createMaskGeneral(self); // <<<<<<<<<
  }
  + (void) checkMask {
      if(mask==nil) printf("error: Mask remains unset\n");
      else {
          Class cls=[mask class];
          printf("mask is an instance of class=%s\n",
                                  class_getName([cls class]));
       }
  }
@end

int main(){
    [Book createMask];
    [Book checkMask];
    return 0;
}
```

The penalty for the dynamic typing and all the magic we can do with the classes is that programs coded in Objective-C are more error prone and more difficult to debug. We will discuss that at the end of this chapter.

An important part of Objective-C is the NextStep (NS) library, which provides NSObject that all the application classes should inherit, basic arrays and collections, and also the Objective-C version of the String class NSString, which can store either C-style text or Unicode:

```
char *s="abcd"; // C-style string
NSString *ns= @"abcd"; // Objective-C type string
char *c=[ns cString]; // conversion from NSString to C-string
printf("%s %s\n", s, [ns cString]);
```

[15] This method can be added at runtime through a utility which uses reflection.

Objective-C also has reference counting which is similar to Java. Each object keeps the count of pointers that lead to it. If the count drops to 0, it means nobody refers to it, and it can be discarded or reused. Compared to Java where this counting is completely transparent and is an integral part of a complex memory management scheme, Objective-C permits custom allocation and a manual control of this count. If you wonder where objects keep this count, it is part of the memory used by the allocator, outside of the object, just before the address where the object starts—see Fig. 6.2.

For more on the garbage collection in Java which was introduced to simplify programming, and now "*its tuning is a long exercise which requires lot of profiling and patience to get it right*"; see Javin (2011).

The Automatic Reference Counting (ARC) was introduced to Objective-C in 2001. It does static code analysis, and automatically inserts retains and releases for objects created by the code.

> Note that Objective-C ARC does not provide a cycle collector; users must explicitly manage the lifetime of their objects, breaking cycles manually or with weak or unsafe references.

ARC may be explicitly enabled with the compiler flag `-fobjc-arc`, or disabled with the compiler flag `-fno-objc-arc`.

The key issue when designing persistence for Objective-C is how to use introspection (reflection) to detect pointers, and then convert this information into the pointer mask. Listing 5.4 shows the trick.

The mask is one of the pieces of information about the class that we keep in structure *persist_params*. *CreateMask* creates one instance of the class and calls *assignIvarValue* with 1. This 1 is the value we want to use to mark the pointers in the mask. Call to *class_copyIvarList* returns an array with one entry for each variable[16] of the class. With it, we get *encoded Type*, and its first character tells us whether it is or isn't a pointer. If it is a pointer we set its value to 1, if not then under default we set it to 0.

This loop does not affect the value of the hidden pointer which is already there. It is not considered to be a variable.

[16] *variable* is Objective-C lingo for C++ *member*

Listing 5.4 Automatic generation of the pointer mask through reflection

```
// Copyright (c) 2013 Raj Lokanath. All rights reserved.
// Modified by Jiri Soukup, 2013

@implementation Util
+ (void) createMask: (Class) klass params: (persist_params*) data {
    data->mask = [[klass alloc] init];
    [self assignIvarValue:1 inObject:data->mask];
}

+ (void)assignIvarValue:(int) value inObject:(id)object {
    unsigned int ivarCount = 0; int ivarIndex;
    Class cls = [object class];
    Ivar *alliVars = class_copyIvarList(cls,&ivarCount);
    for (ivarIndex = 0; ivarIndex < ivarCount; ivarIndex++) {
        Ivar ivar = alliVars[ivarIndex];
        const char *encodedType = ivar_getTypeEncoding(ivar);
        // NSLog(@"%s type %s",ivar_getName(ivar),encodedType);
        switch (encodedType[0]) {
             case '@':// reference of object
            case '*':// pointer to native type
            case '^'://pointer to type
                 object_setIvar(object, ivar, (void*)value);
                 break;

            default:
                 object_setIvar(object, ivar, 0);
                 break;
        }
    }
    free(alliVars);
}
@end
```

Note that the code in Listing 5.4 does not detect pointers which are inside struct instances. In the following sample, pointers *name* and *item* will not show in the mask for class Produce. This is something that is feasible to do; it just needs more code.

On the other hand, since ARC does not allow object references (pointers) inside struct, it is questionable whether the persistence should support it.

```
typedef struct bunch {
     int cost;
     char *name;
     (id)item;
} Bunch;

@interface Produce
{
     float weight;
     Bunch myBunch;
}
@end
```

The other thing missing in Listing 5.4 is that it records only pointers at the level of the given class—and no pointers of its superior classes. If we need all the pointers, we have to go through the entire inheritance hierarchy—see the recursive implementation in Listings 5.5 and 5.6.

Listing 5.5 shows the C++ implementation, in which each application has method *createMask* which creates the mask but only at the level of this class. The code is taking advantage of the default constructor automatically invoking all superior default constructors. The Util class holds flag allocControl which is essentially a nicer form of a global variable. If this flag is 0, default constructors perform normal allocation. When it is 1, *createMask* is invoked.

Listing 5.6 shows the Objective-C implementation,[17] in which not only the mask generation but also the recursion is extracted to a common C-style function (you could use the word "generic") *createMaskGeneric()*. The main concept is the use of the superior class[18] from which this class inherits.

Note the different order in which the inheritance levels are traversed. The order makes no difference when constructing a mask.

Listing 5.5 output	Listing 5.6 output
Before creating the mask	Create mask for C
Create mask for A	Create mask for B
Create mask for B	Create mask for A
Create mask for C	
Create mask for D	

[17] In Objective-C, method init is used as a constructor, but init does not automatically traverse the inheritance hierarchy. Even if we wanted to use the C++ approach, we cannot use it. It would not work.

[18] Keyword or class method *super*.

Listing 5.5 Recursive mask generation, C++ style

```
class Util {
public:
     static int allocControl;
};
int Util::allocControl=0; // 0=normal allocation, 1=mask generation

class A {
     static void createMask(){ printf("create mask for A\n");}
public:
     A(){if(Util::allocControl) createMask(); /* …other code… */}
};

class B : public A {
     static void createMask(){ printf("create mask for B\n");}
public:
     B(){if(Util::allocControl) createMask(); /* …other code… */}
};

class C {
     static void createMask(){ printf("create mask for C\n");}
public:
     C(){if(Util::allocControl) createMask(); /* …other code… */}
};

class D : public B, public C {
     static void createMask(){ printf("create mask for D\n");}
public:
     D(){if(Util::allocControl) createMask(); /* …other code… */}
};

int main() {
     D *d=new D;
     printf("before creating the mask\n");
     Util::allocControl=1;
     d=new D;
     return 0;
}
```

Listing 5.6 Recursive mask generation, Objective-C style

```
void createMaskGeneric(Class cls){
    printf("create mask for %s\n", class_getName(cls)); // see [19]
    Class superClass = class_getSuperclass(cls);
    if([superClass isEqual: [NSObject class]]) return;
     createMaskGeneric(superClass);
}

@interface A : NSObject
   + (void) createMask;
@end
@implementation A
   + (void) createMask{ createMaskGeneric(self); }
@end

@interface B : A
   + (void) createMask;
@end
@implementation B
     + (void) createMask{ createMaskGeneric(self); }[20]
@end

@interface C : B
    + (void) createMask;
@end
@implementation C
     + (void) createMask{ createMaskGeneric(self); }
@end

int main() {
     [C createMask];
     return 0;
}
```

5.6 Errors and Debugging

In a more sophisticated language, the errors will also be more sophisticated, and some of them will show up at the run time instead of the compile time. That means more difficult debugging, and safety issues with software which simply must not fail (space shuttle, computer driven surgery, control of nuclear reactor).

A typical example is what happened to us when we began to test QSP persistence on the benchmark example from Chap. 7. The program crashed in the middle of the run with memory fault on a line which looked quite normal and was coded by all the rules. Only after several days (!) we realized that we did not include <objc/runtime.h> at the

[19] If this line is placed at the end of the function, the order in which the sub-masks are created would be the same as in the C++ version.

[20] Instead of doing this, Objective-C allows one to inject this method at runtime through reflection, without any need to modify the application class.

top of one source file.[21] Compiler in C++ or Java would catch the problem, and we could corrected it within seconds, but Objective-C did not complain about it because there, with its relaxed rules, there was some (remote) possibility of using the particular function even without objc/runtime.h.

Another thing to watch for is that Objective-C classes cannot have an equivalent of C++ static class member—a value associated with the class, not with any object. Variable *data->mask* which we use all through this Chap. 5, is a member of static struct *persist_params* with the scope of the given *.m file. Program using such variables must strictly maintain a separate *.m file for each class.

The Apple variety of Objective-C[22] supports associated object, which is another alternative to this type of object, but it must be created dynamically, unlike the static member in C++.

[21] When a method is not declared or imported, Objective-C assumes a return type id, and that was likely the cause of the crash in this case.

[22] The Windows/gcc variety of Objective-C does not support it yet.