Fundamentals of Persistence

2

Abstract

This chapter is the heart of the book. It explains algorithms, technical details and programming tricks of various approaches to implementation of persistent data—binary and ASCII serialization, memory paging, disk paging and smart pointers. The last section presents QSP (Quasi-Single-Page), a new design of persistent data which, besides other languages, also works in Objective-C and with iPhone applications.

Keywords

Algorithm • Hidden pointer • Object graph • Pointer mask • Regular pointer • Reference • Smart pointer • Swizzling pointers • Traversing objects

This chapter describes several different approaches to the implementation of persistent objects, including algorithms and implementation techniques some of which may not have been published. We start with the concept of pointer mask which, for each class, stores the information about the location of its pointers.

Some algorithms and implementation techniques presented in this chapter have never been published. All the examples in this Chapter are coded in C++, yet many of these ideas are also applicable to other languages. We'll start with the concept of the pointer mask which, for each class, stores the information about the location of its pointers.

Pointer Mask is an object that is used to capture the structure of a class, focusing specifically on where its pointer members are located. You can think of it as a singleton instance of the class which is first filled out with zeros and then all its pointers are set to small positive integers, either 1 (just to identify the pointer location) or to a number specifying the pointer type.

J. Soukup and P. Macháček, *Serialization and Persistent Objects*,
DOI 10.1007/978-3-642-39323-5_2,

Pointer masks have many uses and advantages:

- They tell us instantly (both in code and visually) where we have all the pointers.
- They make it easier to code and debug algorithms.
- They are easy to generate automatically.
- Other representation such as the list of pointers and their offsets within the object can be easily derived from the mask.
- By comparing the masks, we can see whether the old/new classes are different.

Another way of looking at the pointer mask is to start with the fact that, within any object, pointers always start on a 4-byte boundary.[1] Imagine any object broken down into 4-byte sections of potential pointer locations. Instead of some valid pointer, the mask stores an integer in each of these four bytes, so naturally it has the same size as any instance of this class. These integers are 0 for those object members that represent just numbers or text, and are set to non-zero value for pointers.

When constructing a pointer mask, it is important to know that, at the setup time, just before the program starts to run, the persistent system assigns to each class an integer index. It is the same code as if you wanted to find out how many application classes are involved:

```
class Utill {
     static int classesCount;
};
class Library {
     static int classIndex;
};
class Book {
     static classIndex;
);
class Author {
     static classIndex;
};

int Util::classCount=0;
int Library:classIndex=classCount++;
int Book:classIndex=classCount++;
int Author:classIndex=classCount++;
```

[1] On a 64-bit architecture, it is 8-bytes.

POINTER MASK (Example)

```
class Book {                        class Author {
    int numPages;                       ...
    char *title;                        ...
    char category                       static int classIndex;
    Author *authors;                };
    Book *next;
    static int classIndex;
};
```

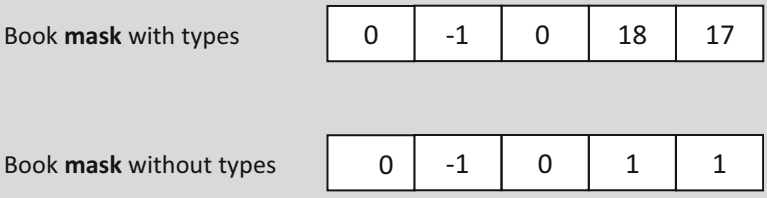
The compiler may keep internal table that looks like this

```
Book   [  int    char*   char     Author*    Book*  ]
```

where each of these members takes 4 bytes of the object, 8 bytes on a 64-bit architecture. Pointers, integers and floats all start on a 4-byte boundary, and even the single character takes 4 bytes including the 3 bytes of padding the compiler inserts. Note that the static members (here classIndex) are not stored inside these objects.

In the persistence systems which store pages of objects as blocks of bytes, we are interested only in the locations of pointers, but if we want to traverse the object graph - as in a typical serialization, we need to know the pointer types.

For this purpose, we create a mask, specific for each class, which has exactly the same number of bytes as one instance of that class. Each 4-byte location which is a potential location of a pointer is treated as an integer, which is 0 for locations that do not store pointers. For pointer locations, it stores the pointer type as the classIndex of its target object. Pointers to built-in types have fixed numbers, for example char* may be recorded as -1. If we assign Book::classIndex=17 and to Author::classIndex=18, then the masks are:



Pointer masks will get more interesting when we will discuss composite objects involving structure-members, inheritance (especially multiple inheritance) and hidden pointers inserted by the compiler.

2.1 Algorithms and Techniques

This chapter describes how to add, automatically and transparently, members and methods to a class. It discusses regular pointers, hidden pointers inserted by the compiler, smart pointers, references, and pointer swizzling. discusses two algorithms (recursive and stack based) which traverse the pointer network and collect all active objects – the critical step in every serialization.

2.1.1 Adding Members and Methods to a Class

Both when making objects persistent and when building intrusive data structures (see Chap. 3), we need to add capabilities to the existing classes. That implies additional methods and members to support these capabilities. There are four ways to do it: from below, from above, inserting them inside, and using a linked storage. Examples in this book mostly *inside* the required methods and members, but keep in mind that this is not the only way. In some situations one of the other options may be a better solution.

2.1.1.1 Adding from Below

If we want to add certain methods and members to every allocated object, we can derive all application classes (and all library classes) from the same base class. For example

```
class PersistBase {
    int counter;
    int mySize();    // ??? see Note1
    static int mode; // ??? see Note2
};
class Employee : public PersistBase {
    int ID;
    Employee *next;
};
```

Note1: Unless mySize() could reach into the allocation record, which may depend on the compiler and OS, or unless counter keeps the size from the time the object was allocated, this would not work.

Note2: This value would be the same for all classes and all objects, an interesting implementation of "global" variable—see bk\chap2\fromBelow.cpp.

2.1.1.2 Inserting Inside

If we want to add more than one member or method to a class,[2] we can insert them with a macro. In the following example each class has an index, and even from the base class we can determine the size of the allocated object. The program prints `size=16` which is the size of Manager.[3]

```
#define Persist(T)                              \
public:                                         \
    virtual int mySize(){ return sizeof(T); } \
    static int classIndex

class Employee {
    Persist(Employee);
    int ID;
    Employee *next;
};
class Manager : public Employee {
    Persist(Manager);
    Employee *secretary;
};
int main(){
    Manager *m=new Manager;
    Employee *e=m;
    printf("size=%d\n",e->mySize());
```

Useful Trick No. 2

Macros, especially long ones, complicate debugging, because compilers and debuggers treat each macro as a single line, but sometimes there is no other choice. The way to minimize the negative impact of a long macro is to insert, with a macro, a short function which calls another function outside of the class.[4] For example, in Listing 2.9 - far below, p.60, macro `INH_REC(T)` inserts a line with a call to `Util::iRep()`. This does two things: (1) it allows us to insert the function yet code it, or most of it, as normal code, not as a macro and (2) it allows the outside function to use class parameters which are private and normally not available outside.

[2] The difference from adding to an object *from below* becomes apparent when inheritance is involved.

[3] Two 4-byte members in Employee, one in Manager plus one hidden pointer as will be explained in Sect. 2.1.2.

[4] This coding style was recommended by Sean Yixiang when coding the Objective-C persistence in Chap. 7.

Here is a simpler example, where we are adding a long function foo() to class Book. The function needs the value of member ISDN, which is private. We can do it with a long macro, which is not nice and is difficult to debug:

```
#define FOO \
void foo(){ \
   .. long code using value of ISDN \
}

class Book {
private:
    int ISBN;
public:
    FOO
};
```

Instead of using a long macro, we can code the main part of foo() outside of Book, either as a plain C function, or as a static function of some utility class:

```
class Utility {
friend class Book;
    static void foox(int isbn){
        ... bulk of the function, using the private Book::ISDN
    }
}

#define FOO \
    void foo(){Book::foox(ISBN);}

class Book {
private:
    int ISBN;
public:
    FOO
};
```

2.1.1.3 Adding from Above

As *from below*, this method allows one to expand object, not class. We derive a special class from the class we want to expand and add the members and methods there. The disadvantage is that in calls to new() and possibly other methods you have to cast to the expanded class (starting with Exp_. . .). For example:

```
class Employee {
    int ID;
    Employee *next;
};

class Exp_Employee : public Employee {
public:
    Exp_Employee *nextFreeList;
    static Exp_Employee *freeListStart;
    static void addFreeList(Employee *e){
        Exp_Employee *ee=(Exp_Employee*)e;
        ee->nextFreeList=freeListStart;
        freeListStart=ee;
    }
    static void delFreeList(Exp_Employee *e){...}
};
Exp_Employee* Exp_Employee::freeListStart=NULL;

int main(){
    Employee* e=new Exp_Employee;
    Exp_Employee::addFreeList(e);
```

2.1.2 Hidden Pointers

The first step to implementing any style of persistence is to understand the internal representation of objects. In the early years of C++ there was a multitude of compilers, each with its own quirks and representation of objects. Writing portable C++ persistence used to be a pain.[5]

The C++ standard does not specify the internal implementation of objects, but most compilers today use the model shown in Fig. 2.1.[6] If neither the class itself nor the classes from which it inherits have virtual functions, the memory image consists of all the members (fields) in the same order as they are hierarchically listed in the class definition.[7] If there are virtual functions, then there is a *hidden pointer* at the beginning of the object.[8] In the case of multiple inheritances, there are additional hidden pointers inside the object. Hidden pointers point into the internal table of virtual functions, and are identical for all instances[9] of the same class. Application programmers have no access to these hidden pointers and tables, and often are not even aware of their existence.

[5] The code of DOL library (Data Object Library 2013) still has `ifdef` statements for Borland, Watcom, Microsoft, Mac, Linux, Zortec, DEC, VMS, Sun, Lucid, GNU, IBM, Solaris, Liant, Amdahl, Coherent, Apollo, Saber and HP compilers.

[6] For the program which generates this information, go online to `bk/chap2/dispPtrs`

[7] As in plain C.

[8] In most OO languages including Java and C# the internal object representation is probably similar.

[9] Terms *object of classA, A-object,* or *instance of A* mean the same thing.

```
class C {...};
class B : public C {...};
class D : public C {...};
class A : public B, public D {…};
class E : public B {…};
```

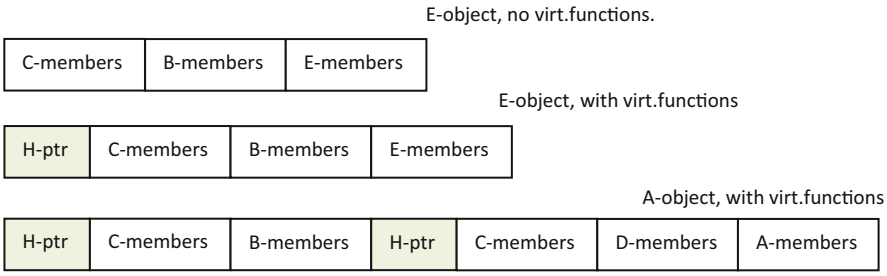


Fig. 2.1 Examples of hidden pointers in C++ objects (Visual Studio 2010). Note that an A-object includes two different instances of the C-class

On a 32-bit architecture, pointers and 4-byte numbers always start on a 4-byte boundary. On 64-bit architecture, pointers and 8-byte numbers usually start on an 8-byte boundary.[10] The `sizeof()` function returns the true size of the object, including the hidden pointers.

A convenient tool for detecting and manipulating these pointers is operator `new()` which can be controlled by an outside variable, static pointer `objBuf`, to do three things[11]:

(1) When `objBuf=NULL, new()` allocates a new object as usual.
(2) When `objBuf` points to a block of memory, `new()` adds hidden pointers to it, thus turning it into a valid object.
(3) When `objBuf=(char *)(1), new()` allocates a 0-filled object, then sets the hidden pointers to

Case (1) is used for allocation of objects during the program run.
Case (2) is useful when retrieving persistent objects from the disk.
Case (3) creates a mask similar to Fig. 2.1.
The algorithm recognizes a valid pointer by having a value which is a multiple of 4.

[10] The lowest two bits of any pointer are always 0 and, temporarily, they may store flags or other information during some algorithms.
[11] See Listing 2.1.

Listing 2.1 Overloaded operator new() which works in three different modes: normal, updating hidden pointers, and generating a mask. [For the explanation of how this relates to so called "placement new", see the Note after the listing.]

```
class A {
    ... private members, no pointers
public:
    static void *objBuf; // controls what new() does
    static void *mask;   // for
    void* operator new(size_t size){
        unsigned long u=(unsigned long)objBuf;
        if(u==0) return malloc(size); // normal operation
        else if(u&3) return mask=calloc(1,size); // create mask
        else return(objBuf); // insert hidden pointers
    }
};
void* A::objBuf=NULL;
```

Note:

Placement new gets a section of memory and turns it into a valid object by filling in the hidden pointers. For example for class Book,

```
void *v=calloc(sizeof(Book),1);
Book *bp=new Book(v);
```

or on one line

```
Book *bp=new Book( calloc(sizeof(Book),1) );
```

If we wanted to control the allocation of objects by calloc or some custom allocation function the application would have to change all the calls to new() to this ugly and potentially error-prone syntax.[12]

Overloading new() as we did in Listing 2.1 hides all this, and the application can create objects as usual. No change of calls to new() is required:

```
Book *bp=new Book();
```

However, the last line of operator new in Listing 2.1

```
else return(objBuf); // insert hidden pointers
```

is really nothing else than placement new, which we use in a special case when we just want to set or update hidden pointers. The difference from the normal placement new is that the memory is not supplied as the function parameter, but as the static class member `objBuf`.

[12] Note that this is similar to what you have to do when using ObjectStore (c) PSE Pro for C++.

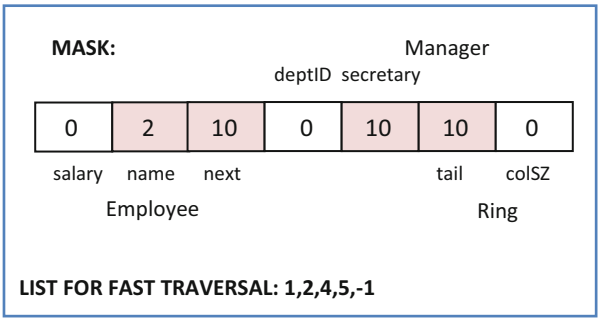


Fig. 2.2 Mask for the Manager class from Listing 2.2. Each box corresponds to a potential pointer location (4B or 8B depending on the system architecture). Pointer locations are marked by the index of the target class, here 2 = text string, 10 = Employee

2.1.3 Regular Pointers

Regular pointers are the pointers the application inserts into classes. After you write objects to disk and then read them back to memory, the new objects are in different locations, and all the regular pointers must be replaced (*swizzled*) to the new addresses of their target objects. If you read the object back within the same program run, hidden pointers are the same, but for a different run even hidden pointers usually change.

How to detect all these pointers is one of the key tasks every persistent system must tackle.

For example, if a company hierarchy is described by classes Manager and Employee, we can represent the internal structure of each class by a mask—see Listing 2.2 and Fig. 2.2. Such masks are useful when planning algorithms or debugging code, and we will use them extensively throughout this book.

Note that it is reasonably fast to traverse a mask when swizzling pointers. However, a small performance improvement can be achieved by keeping, in addition to the mask, a list of non-zero entries in the mask. Note that mask in Fig. 2.2 does not have any hidden pointers because the two classes have no virtual functions.

Listing 2.2 Another version of Manager/Employee classes (online listed only as list2_2.txt)

```
template< class T> class Ring {
    T *tail;
    int colSZ;
};
class Employee {
    float salary;
    char *name;
    Employee *next;
};
class Manager : public Employee {
    int deptID;
    Employee *secretary;
    Ring<Employee> myPeople;
};
```

2.1.3.1 Detecting Pointers with Reflection

When reflection is available, we don't need a mask. And even if we had one it would not help much. Languages with reflection usually work with references, and objects and their parts cannot be accessed by their memory addresses.

When we need to traverse references of an object, the reflection allows us to traverse members and, for each member, it tells us whether the member is a reference and what is the type of its target. Listing 2.3 shows how this is done in Java, and Listing 2.4 shows the C# implementation.

It may not be obvious from this code, but it traverses pointers all through the inheritance hierarchy, e.g. for the Manager object from Listing 2.2, the code visits

```
Employee::name,
Employee::next,
Ring::tail,
Manager::secretary.
```

Listing 2.3 Using Java reflection to traverse references[13]

```
import java.lang.*;
import java.lang.reflect.*;

Field[] fields = cls.getDeclaredFields();
Object val; Class targetClass;

for(Field field : fields ){
    if(field.getType().isPrimitive())continue;
    val=field.get(this);
    if(field.getType() == String.class){
        ... // create or find new val
        field.set(this,val);
    }
    else {
        targetClass=field.getType();
        ... // create or find new val
        field.set(this,val);
    }
}
```

[13] For full source, see bk/chap2/reflectJava.

Listing 2.4 Using C# reflection to traverse references[14]

```
//flags: which members we want to enumerate
System.Reflection.BindingFlags flags =
    System.Reflection.BindingFlags.Public |
    System.Reflection.BindingFlags.NonPublic |
    System.Reflection.BindingFlags.Instance;

Object val; Type targetClass;
foreach (System.Reflection.FieldInfo field in
                        this.GetType().GetFields(flags)){
    if(!field.FieldType.IsClass)continue; // not a reference
    val=field.GetValue(this);
    if(val==null)continue; // no conversion for null references
    if(field.FieldType == typeof(string)){  // string
        ... // create or find new val
     field.SetValue(this,val);
    }
    else {
        targetClass=field.FieldType;
        ... // create or find new val
        field.SetValue(this,val);
    }
}
```

2.1.3.2 References Registered for Each Class

All C++ and Objective-C persistent systems must get the information about pointers externally, and one possibility is to assume that the user registers all persistent classes by listing their pointers.

In C++, our favourite method is to use macros PTR and STR[15] in the default constructor. It has the advantage that it automatically traverses the inheritance hierarchy, and the result is a mask which is a flat view of even highly composite object. Here is an example of how to use these macros:

```
class Employee {
    static void **mask; // not persistent
    float salary;
    char *name;
    Employee *next;
public:
    Employee(){
        salary=0.0;
        STR(name); PTR(next,Employee);
    }
};
```

Listing 2.5 shows how this syntax can generate the mask. The listing may appear long, but note that there is a lot of repetition: the same functions and static variables are added to all three classes.

[14] For full source, see bk/chap2/reflectCs.

[15] A similar method to register pointers is also used by POST++.

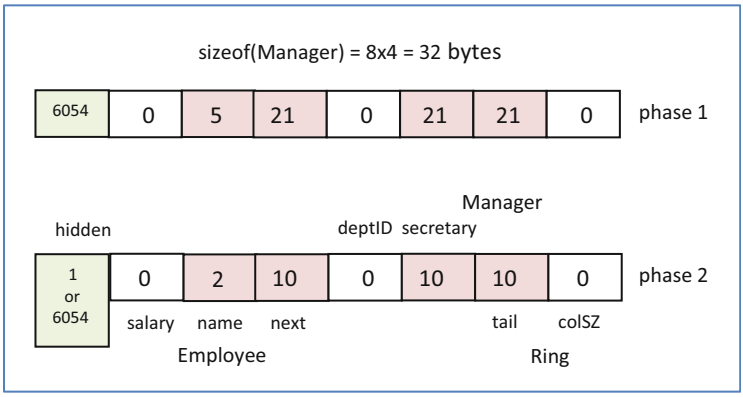


Fig. 2.3 Generating mask for the Manager class. Listing 2.5 produces directly the phase2 mask with true value of the hidden pointer (6054). The online version at bk/chap2/list2_5.cpp generates first the phase1 mask and then converts it to phase2 with 1 marking positions of hidden pointers. Mask codes: 0 = invariable members, 1 = hidden pointer, 2 = char*, 10 = Employee*

At the setup time, before the program starts to run, each class gets its unique index. Automatic assignment of class indexes happens at the setup time, before the application program even starts to run—look at the last line just before main().

Inside createMask(), the call to new() with objBuf=1 creates a 0-filled instance of Manager and inserts hidden pointers. Then, through PTR() and STR(), the default constructor Manager() marks the pointer locations in the mask.

Figure 2.3 has two numbers in the box for the hidden pointer: 1 or 6054. In most environments, hidden pointers are large numbers which are easy to distinguish from the class index stored for regular pointers. In environments, where the system stores index(!) into the virtual function table, we mark hidden pointers by using 1 in the mask, and storing the value of the hidden pointer in a separate, additional mask.

`objBuf` must be either a global variable or a static variable of a special Utility class.

Listing 2.5 Generating mask with both hidden and regular pointers (for full, slightly modified source, see bk/chap2/list2_5.cpp)

```
#define PTR_SZ sizeof(char*)
int totIndex=9; // index of application classes will start from 10
void *objBuf=NULL; // global allocation control

#define PTR(P,T)                          \
if(objBuff==NULL || objBuf==(void*)1)P=NULL; \
else P=(T*)(T::getIndex())

#define STR(P)                            \
if(objBuff==NULL || objBuf==(void*)1)P=NULL; \
else P=(char*)(2)

class Employee {
    float salary;
    char *name;
    Employee *next;
public:
    // ... static members and methods, new()as for Manager
    Employee(){STR(name); PTR(next,Employee);}
    int virtual trueClass(){return classIndex;}
};
// ... initialize static members as for Manager

class Ring {
    Employee *tail;
    int colSZ;
public:
    // ... static members and methods,new()as for Manager
    Ring(){PTR(tail,Employee);}
    int virtual trueClass(){return classIndex;}
};
// ... initialize static members as for Manager

class Manager : public Employee {
    static void *mask;
    static int classIndex; // app.classes start from 10
    static int mySize;
    int deptID;
    Employee *secretary;
public:
    Ring myGroup;
    static int getIndex(){return classIndex;}
    void* operator new(size_t size){
        unsigned long u=(unsigned long)objBuf;
        if(u==0) return malloc(size); // normal operation
        else if(u&3){ return mask=calloc(1,size); } // mask
        else return(objBuf); // insert hidden pointers
    }
    static void createMask(){
        int i; char *s; int *ip;
        objBuf=(void*)1;
        new Manager; // phase one of setting the mask
    }
    static void prtMask(){ ... }
    Manager(){PTR(secretary,Employee);}
    int virtual trueClass(){return classIndex;}
};
void* Manager::mask=NULL;
int Manager::mySize=sizeof(Manager);
int Manager::classIndex=totIndex=totIndex+1;

int main() {
    Manager::createMask();
    Manager::prtMask();
```

When we replace the statements that repeat for every class by macro `PERSIST(T)`, this complex code turns into nice and crisp Listing 2.6.

Macro `INIT_STAT(T)` initializes static variables for each class, and macros `PTR(P,T)` and `STR(P)` are as before. The parameters of all these macros are types; they are just like templates/generics except that they represent a block of code—not a class or a function.

Listing 2.6 Code from Listing 2.5, where generic-like macros replace code that repeats for every class

```
class Employee {
    PERSIST(Employee);
public:
    float salary;
    char *name;
    Employee *next;
    Employee(){ STR(name); PTR(next,Employee); }
};
INIT_STAT(Employee);

class Ring {
     PERSIST(Ring);
public:
    Employee *tail;
    int colSZ;
    Ring(){ PTR(tail,Employee); }
};
INIT_STAT(Ring);

class Manager : public Employee {
    PERSIST(Manager);
public:
    int deptID;
    Employee *secretary;
    Ring myGroup;
    Manager(){ PTR(secretary,Employee); }
};
INIT_STAT(Manager);

int main() {
    Manager::createMask();
    Manager::prtMask();
    printf(
      "classIndex: Employee=%d Ring=%d Manager=%d\n",
       Employee::getIndex(),Ring::getIndex(),
       Manager::getIndex());
    return 0;
}
```

Useful Trick No. 3
Macro PTR(P,T) can set member pointer to 1, or generate the pointer name and type as text strings.

```
#define PTR(P,T) \
     (P)=(T *)1;  \
     printf("pointer name=%s targetType-%s\n", #P, #T);
```

For the strings, the macro could be replaced by a method, possibly static method of the class; setting the pointer to a value must be through a macro if you want this simple interface.

2.1.3.3 Smart Pointer that Registers Itself

Another way to generate the mask is to replace pointer members that we want to be persistent by an instance of a special smart-pointer class, see Listing 2.7. Such a smart pointer does not take more space than a normal pointer and is used just as a normal pointer, but it can record itself in the mask.

Listing 2.7 Mask generation with smart pointer (code sketch only, no program online)

```
template<class T> PersistPtr {
    T *ptr;
public:
    PersistPtr(){
        ptr=NULL;
        ... // mark the mask at the position of 'this'
    }
    T* operator->() const{ return ptr; }
    ... // other operators
};

class Employee {
    PERSIST(Employee);
public:
    float salary;
    PersistPtr<char> name;        // <<<<<<
    PersistPtr<Employee> next;    // <<<<<<
    Employee(){}
};
INIT_STAT(Employee);

/* similar syntax for classes Ring and Manager */

int main() {   // remaing exactly as before
    Manager::createMask();
    Manager::prtMask();
    return 0;
}
```

So far we have been working with pointers leading to a single object or to a single text string. However, there can also be pointers to various types of arrays:

```
class B;
class C {
     B *bArr; // to array of B objects
     B **bpArr; // to array of (B*)
     int *iArr; // to array of int
     char *cArr; // to array of characters
     char **cpArr; // to array of (char*)
     int aSize; // assume all arrays have this size
};
```

To register all these situations, calls to `PTR()` and `STR()` are not sufficient. We also need to register the size of the array which in most cases is already a member of the class which stores the pointer. If it is not, we always can set up special macros for such situations: `ARR()` for an array of objects and `ARP()` for an array of pointers are handy to register such situations. For example, the pointers used by class C in the last example can be registered by the following default constructor:

```
class C(){ARR(bArr,A,aSize); ARP(bpArr,B,aSize); ARR(iArr,int,aSize);
          ARR(cArr,char,aSize); ARP(cpArr,char,aSize);
}
```

Note that `aSize` is the name of the member, not a numerical value!

2.1.3.4 Smart Library Registering Pointers

The problem with registering pointers is that if you miss even a single one, it will not be swizzled,[16] and your program will crash on loading the data from disk. Also, as will be explained in Sect. 2.1.6, if a pointer is missing in the mask, the object to which it leads and perhaps many other objects may be missing on the disk file. Registering pointers is not something application programmers should do in their everyday work.

The idea of registering pointers opens another Pandora's box. What is the true purpose of these dangerous pointers inhabiting our classes, and why are they allowed to live there with all the mischief they can cause? And could we hide and isolate them in some place where they would be under better control?

That goes far beyond persistence, but the problem with registration of pointers only adds to the many reasons why we should avoid raw-pointer members in application classes.

The purpose of pointers is to implement data structures and relations. For example, instead of using raw pointers `tail` and `next` in Listing 2.6, it is better to replace these pointers by a generic data structure consisting of classes Ring<T> and RingPart<T> that comes from a library which takes complete care of these pointers including their registrations, and these pointers are transparent to the application code.

[16] As introduced in Chap. 1, *swizzle* is a commonly used term for the process of updating pointers when the objects move to a different memory location.

When following this strategy, we end up with no pointer-members in our application classes. However, the necessary condition for all this is that the library must support bi-directional data structures, which also is the prime reason we always use DOL or InCode libraries and not the standard containers.

Compare the following three implementation of the same class:

```
class Project {     // Code with raw pointers
     char *name;    // bad choice, raw pointer
     Manager *mgr; // bad choice, raw pointer
};

class Project : public OneToOne<Manager>,
                public String { // better code, Style 1
};

class Project { // best code, Style 2
     String name; // better choice, pointer handled by library
     OneToOne<Manager> mgr; // library class, better choice
};
```

Styles 1 and 2 remove pointers from the application code but, in more complex situations, Style 2 ends up using multiple inheritance and, in our experience, it is more difficult to manage.

The format in which we record pointers in the library classes does not have to be particularly efficient or easy to use, because you register the class when you add it to the library, and, from that moment on, many people use it but nobody is even aware that there is any registration.

For example, Data Object Library (Data Object Library 2013) is a C++ library of bi-directional intrusive data structures which are also persistent. Each data structure is represented by a class which does not have any attributes, and its methods (operations of the association) have access to pointers and other attributes of the application classes that participate in the data structure. For an example, see Doubly Linked Aggregate in Fig. 2.4. When you want to set up an aggregate between classes Room and Students, you declare

```
Association Doubly_Linked_Aggregate <Room,Student> students;
```

The pointers are registered in a library files *registry* and *zzmaster* which essentially contain this record[17]:

```
Doubly_Linked_Aggregate 2
   1: child 2
   2: next 2, prev 2, parent 1
```

which means that in our Room/Student example we will have

[17] Line1: two participating classes, Line2: pointers in the first class with the index of their target class, Line3: pointers in the second class with the index of their target class.

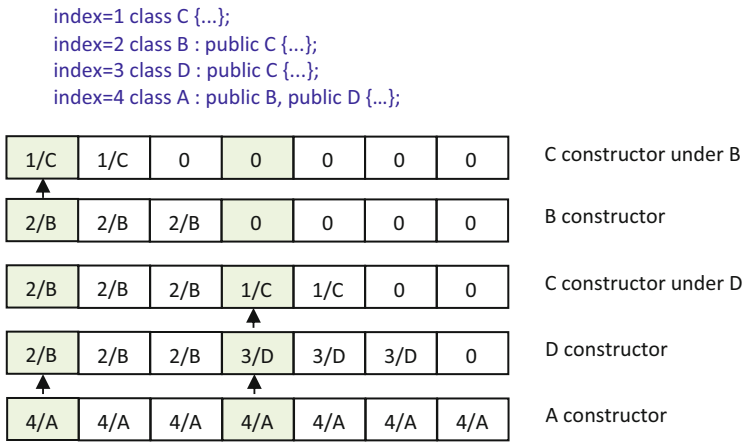


Fig. 2.4 Evolution of tMask when allocating a new A-object. This is a dynamic process which takes the advantage of default constructors for all the classes being called bottom up. Any time a non-zero location or a hidden pointer is overwritten, it is an indication of inheritance—see the *arrows*

```
class Room {
  Student *child;
  ...
};
class Student {
  Student *next;
  Student *prev;
  Room *parent;
  ...
};
```

Pointers can come only from the library, so the library can determine what the mask of the two classes will be. All this is transparent and the user does not have to worry about registration of pointers.

2.1.3.5 Detecting Pointers with a Code Generator

Until now we have assumed that the persistence would be added to the application program as additional source or library. However, applying a code generator to some of the tasks, such as detecting pointers, can significantly simplify the user interface. It is not considered a "pure" programming technique, because it may complicate debugging, use of debuggers and IDE, and using software designed in this way as a part of a larger system, but it leads to a more elegant interface.

We can think of many ways to detect pointers with a code generator. Let's explore one possible approach which we have never used on a real application, but which would be fairly simple to implement. Assume that for every class in the application source, e.g. class Employee, we create a twin, Twin_Employee, which

has the same members and thus the same mask. We discard all its methods, but add a default constructor with PTR() and STR() statements as in Listing 2.6. This allows us to generate simple code which, for each of the Twin-. . . classes, finds it mask. If we can link together the original class with its twin, it is as if we added the mask to the original class without providing any information about its pointers members.

```
class Employee {        // application class
    float salary;
    char *name;
    Employee *next;
public:
    float getSalary(){return salary;}
    void setSalary(float sal);
    Employee(){salary=10000;}
};

class Twin_Employee {       // twin class
    float salary;
    char *name;
    Employee *next;
public:
    Employee(){STR(name); PTR(next,Employee);}
};
```

What we proposed includes some logical leaps, and we have to explore the idea step by step in order to verify that it will really work. We do not have to make a complete syntax analysis.

Let's assume that, as the first pass, we convert the code to a stream of tokens while implementing all the name substitutions encoded by `typedef` or `#define` statements and removing comments and access indicators.[18] We get

```
class Employee { float salary ; char * name ; Employee * next ;
float getSalary ( ) { return salary ; } void setSalary ( float sal )
; Employee ( ) { salary = 10000 ; } } ;
```

In the second pass, we add the twin underscore (__) prefix to the class name and monitor the depths of { }, (), [] and <> brackets (each separately) as we traverse the tokens. We throw away any token for which the depth of { } is not 1 or the depth of any other bracket is more than 0. That gives us

```
float salary ; char * name ; Employee * next ; float getSalary ( ) {
 } void setSalary ( ) ; Employee ( ) { }
```

This allows us to identify statements which end with one of three ways:

[18] Public, private or protected.

`{ }`, or `( ) or;` or just `;`

Eliminate statements that do not end just with ";" and we have the list of members

```
float salary;
char * name ;
Employee * next ;
```

This allows the code generator to create the twin class

```
class Twin_Employee {  // added Twin_
    // next part is the list of members after pass 3
    float salary ;
    char * name ;
    Employee * next ;
    // remaining part is all generated, using members with *
public:
    Employee(){STR(name); PTR(next,Employee);}
};
```

This allows us to generate mask for class Twin_Employee as described in Sect. 2.1.3.2. The last missing piece of this puzzle is how, for an object of class Employee, we could quickly find the mask of Twin_Employee.

Let's assume that the code generator also creates class derived from class Employee, which adds methods and possibly members.[19] We will use prefix Exp_ for this class in order to show that it is an expansion of the original class. If we do not add any non-static members, the class will have the same original size.

```
class Exp_Employee : public Employee {
    void *getMask(){ return Twin_Employee::mask;}
};
```

The result is elegant. If you want to make any application code or library persistent you run the code generator on their classes and the only change you have to make in the code is to replace all calls to the new() operator:

```
int main() {
    Employee *e12, *e2;  void *mask;
    e1=new Exp_Employee;
    e2=new Exp_Employee;
    mask=(Exp_Employee*)e1->getMask();
    // otherwise use e1 and e2 as if there is no persistence
```

This is not necessarily better than using PTR() and STR() in your application classes. You may have many new() statements spread through your code, while PTR() and STR() statements are localized in the class definitions and may be much fewer. However, making an existing class library persistent with a code generator

[19] This is the method of adding *from above* as described in Sect. 2.1.1.3. It adds to each allocated object, not to the class.

may be easier, since a typical container library may not have many, if any, new() statements.

All this works even when some application classes inherit from other classes, assuming that the code generator converts all the classes to their twin classes. For example, if we have

```
class Employee {
      ...
};
class Manager : public Employee {
      ...
};
```

it converts it to

```
class Twin_Employee {
      ...
};
class Twin_Manager : public Twin_Employee {
      ...
};
```

http://www.springer.com/978-3-642-39322-8

Serialization and Persistent Objects
Turning Data Structures into Efficient Databases
Soukup, J.; Macháček, P.
2014, XVIII, 263 p. 133 illus., Hardcover
ISBN: 978-3-642-39322-8