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

```
class Author {
    ...
    ...
    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:

Book **mask** with types	0	-1	0	18	17

Book **mask** without types	0	-1	0	1	1

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 {…};
```

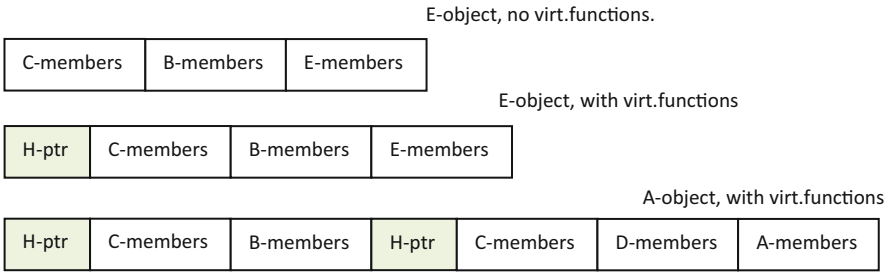

Fig. 2.1 Examples of hidden pointers in C++ objects (Visual Studio 2010). Note that an A-object includes two different instances of the C-class

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++.

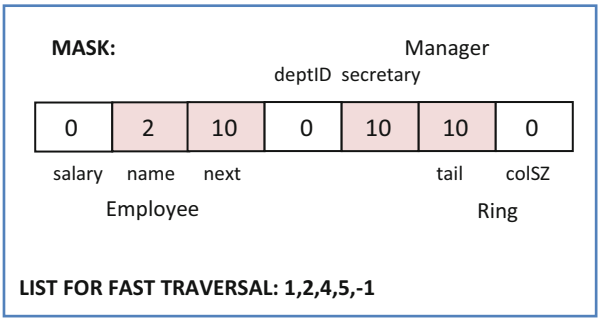

Fig. 2.2 Mask for the Manager class from Listing 2.2. Each box corresponds to a potential pointer location (4B or 8B depending on the system architecture). Pointer locations are marked by the index of the target class, here 2 = text string, 10 = Employee

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++.

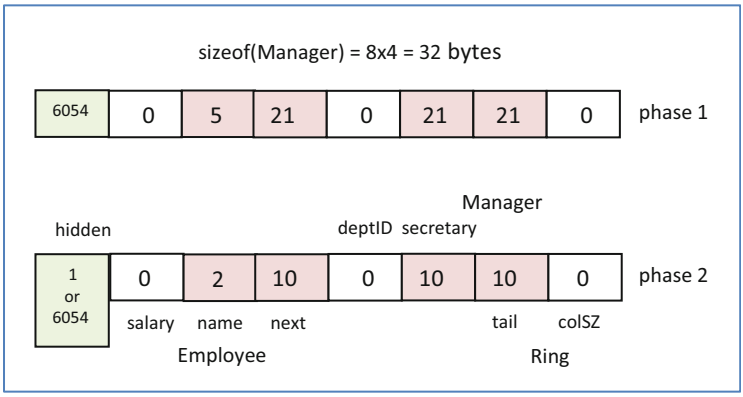

Fig. 2.3 Generating mask for the Manager class. Listing 2.5 produces directly the phase2 mask with true value of the hidden pointer (6054). The online version at bk/chap2/list2_5.cpp generates first the phase1 mask and then converts it to phase2 with 1 marking positions of hidden pointers. Mask codes: 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 it that simple.

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.

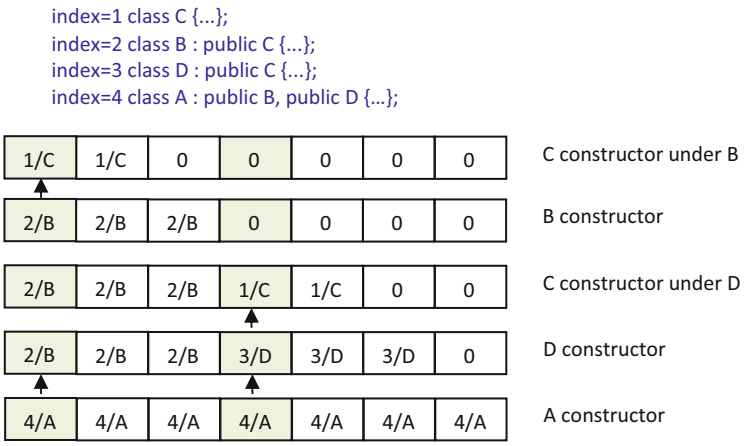

Fig. 2.4 Evolution of tMask when allocating a new A-object. This is a dynamic process which takes the advantage of default constructors for all the classes being called bottom up. Any time a non-zero location or a hidden pointer is overwritten, it is an indication of inheritance—see the *arrows*

```
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:

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

[18] Public, private or protected.

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

[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.

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 {
      ...
};
```

2.1.4 Arrays

When you come across a pointer while reading C++ code, you cannot tell whether it leads to a single object or to an array of objects. And even if you know that it leads to an array, you have no clue about its size. That can lead to nasty surprises. The program in Listing 2.8 writes outside of its memory space, and that results in strange behaviour. It compiles on our laptops,[20] but then it we attempt crashes when we attempt to run. However, when we uncomment the printf() statements, it compiles and runs without crash. Yet there is nothing wrong with the printf() statement.

[20] Using Visual Studio 2010.

Listing 2.8 A pointer can lead to a single object or to an array, which is a potential source of errors

```
class A {
public:
     int weight;
};
class B {
     A *ap;
     A arr[8];
public:
     void foo(){
          ap=new A;
          ap->weight=123;
          ap[0].weight=234; // OK even though ap is not an array
// printf(" before the first potential problem\n");
          ap[2].weight=567; // wrong, possible crash
          ap=new A[60];
// printf(" before the second potential problem\n");
          ap[60].weight=789; // possible crash, index overflow
          ap->weight=999; // OK, really gets ap[0].weight
// printf(" before the third potential problem\n");
          arr[62].weight=789; // possible crash, index overflow
     }
};
```

In order to save an array properly to disk, we need to know when the pointer represents an array, and the size of that array. This is the reason why all persistent systems and languages with built-in persistence assume that pointer members always point to a single object, and that arrays are implemented through a special Array class, which stores the pointer, the size of the array, and the number of used entries. This class is usually one of the special types, and has a pre-assigned internal index just like `char`, `int`, or `float`.

2.1.5 Extracting Inheritance

An interesting feature of what we have done so far is that we have achieved persistency without extracting any information about inheritance among application classes. Virtual function `PersistObj::trueClass()` does everything we need.

However, there are situations when the information about inheritance may be useful or even essential, for example when generating UML class diagram.[21] In languages with reflection this information is readily available. In C++, there are two ways to extract this information, and both are simple and straightforward.

METHOD 1: Partial syntax analysis (using a code generator).

[21] We will discuss this in more detail in Sect. 4.4.

– Concatenate all the source with definitions of all application classes into one file. This usually means all the *.h files; for small programs it may be just one *.cpp file with the entire program.
– Make pass eliminating comments and lines starting with #. At the same time, break the source into tokens separated by one space. Monitor the depths of { }, (), [] and <> brackets (each separately) and throw away any token for which at least one of these depths is not 0. After you do this, Listing 2.6 is reduced to

```
class Employee { } ; INIT_STAT ( ) ; class Ring { } ; INIT_STAT ( ) ;
class Manager : public Employee { } ; INIT_STAT ( ) ; int main ( ) {
}
```

– Make another pass searching for token `class`. When you find it, look for two possible patterns:

`class A {` A does not inherit from another class
`class A :` A inherits from one or more classes In the second case, the continuation must be

```
X B , … , X D {
```

where X is anything or missing, and B . . . D are names of the classes from which A inherits.

METHOD 2: Evolving tMask (no code generator).

Let's write a program which watches how default constructors visit their parts of the object. The algorithm is similar to how we created the pointer mask in Sect. 2.1.3.2, but instead of recording pointers we will let all default constructors to write in it the index of the class to which they belong. The constructors are invoked bottom up, and when the areas overlap it implies inheritance—see Fig. 2.4.

Listing 2.9 shows the implementation of the algorithm which is a bit tricky to debug. It produces the following output:

```
A=4      Inheritance of A:
B=2      ----- 2 inherits from 1
C=1      ----- 3 inherits from 1
D=3      ----- 4 inherits from 2
E=5      ----- 4 inherits from 3
```

How does it work:

`Utility::tMask` stores the mask as it is built, and because it is static and public, it is essentially a global variable.

Operator `new()` which is under `Persist(T)` catches the initial mask before the constructors start to add to it, but only when `inhFlg=1`. For `inhFlg=0`, operator new() allocates normal objects as expected.

Under `INH_REC(T)`, a call to `Utility::iRep()` is inserted to every default constructor. This function fills the appropriate section of the mask with the class signature.

Listing 2.9 Extracting inheritance without using code generator (Fig. 2.5)[22]

```
typedef unsigned long UL; // unsigned integer as long as a pointer
#define HP_LIMIT 1000; // lower limit on hidden pointers

class Utility {
public:
     static int totIndex;
     static int inhFlg; // 0=normal, 1=detecting inheritance
     static UL *tMask;
     static void reportInheritance(int a,int b){
          printf("----- %d inherits from %d\n",a,b);
     }
     static void iRep(int sz,UL *localMask,int cIndex){
       int i,report;
       sz=sz/sizeof(char*);
       for(i=0; i<sz; i++){
       if(localMask[i]>0 && localMask[i]<1000){
             if(i+1>=sz)report=1;
             else if(localMask[i+1]!=localMask[i])report=1;
             else report=0;
             if(report)reportInheritance(cIndex,(int)localMask[i]);
          }
          localMask[i]=cIndex;
       }
     }
};
int Utility::totIndex=1; // we want it to strat from 1, not from 0
UL* Utility::tMask=NULL; // allocated by new()
int Utility::inhFlg=0;

#define PERSIST(T)                                          \
public:                                                     \
static int classIndex;                                      \
static int inhFlg; /* 1 when searching for inheritance */   \
void* operator new(size_t size){                            \
     void *r=malloc(size); /* normal operation */           \
     if(Utility::inhFlg)Utility::tMask=(UL*)r;              \
     return r;                                              \
}                                                           \
static void reportInheritance(){                           \
     Utility::inhFlg=1;                                     \
     Utility::tMask=(UL*)(new T);                           \
     delete Utility::tMask;                                 \
     Utility::tMask=NULL;                                   \
     Utility::inhFlg=0;                                     \
}

#define INH_REC(T) if(Utility::inhFlg) \
  Utility::iRep(sizeof(T),(UL*)((T*)this),classIndex)

#define INIT_STAT(T) \
int T::classIndex=Utility::totIndex=Utility::totIndex++

class C {
PERSIST(C);
     int c;
public:
     C(){INH_REC(C);}
};
INIT_STAT(C);

... other classes coded in the same style

class A : public B, public D{
PERSIST(A);
 int a;
public:
     A(){INH_REC(A);}
};
INIT_STAT(A);

int main(){
    A::reportInheritance();
```

[22] Running source is in bk\chap2\list2_9.cpp

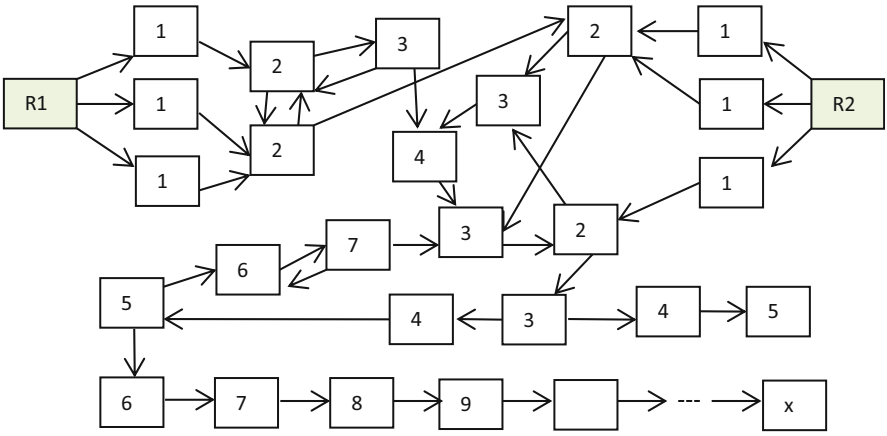

Fig. 2.5 Visiting objects by traversing pointers from roots R1 and R2. Nodes marked 1 are direct neighbours of the roots, nodes marked 2 are their direct neighbours, and so on until there are no unvisited neighbours. The numbers express the depth from the roots

2.1.6 Collecting All Active Objects

In serialization, and only in serialization,[23] we have to find all objects in our data space and save them to disk. There are three possible approaches:

(A) If the data structures are simple, the application can include a function which traverses all the objects and writes them to disk. That is efficient, but difficult to manage for complex projects, and certainly not automatic.

(B) For each class we can maintain a list of its active objects. Operator `new()` adds an entry to this list, and destructor moves it to the free list. Both lists use the same two[24] references per object. This is fast, efficient, the list of active objects is ready when we need it and, as a bonus, we get the free lists for the reuse of discarded objects. The price is the space of two references per object, and the potential problem with objects that were not properly destroyed remaining on the active list.

(C) Objects and references form a directed graph — see Fig. 2.5. Usually, in this graph, there is one or a few root objects from which we can reach all the other objects by traversing the references. If there isn't such a root or roots, you can always add a class that will serve this purpose. This is the method used by most existing serializations, and it deserves more discussion. There are two basic approaches to its implementation, and if you are not careful there may be unpleasant side effects.

[23] When the persistence is built on memory pages we do not need to do this.

[24] When using a doubly-linked list, removing an object is instant; removing it from a singly-linked list requires a search.

When traversing a general graph like this, we can proceed in two ways—depth first or breadth first. The depth-first search is usually coded as a recursive function which we call with obj=root. This is a pseudo-code:

```
void depthSearch(void *obj){
     for(all references ref of obj){
          if(ref not stored yet){
            depthSearch(ref);
          }
          store(obj);
     }
}
```

The breadth-first search is best implemented with a FIFO[25] queue, and the implementation is not recursive:

```
void breadthSearch(void *obj){
     queue.in(obj); // add obj to the queue;
     while(queue not empty){
          obj=queue.out(); // get next object from the queue
          save(obj);
          for(all references ref of obj){
               if(ref has not been in queue yet){
                    queue.in(ref);
               }
          }
     }
}
```

Both implementations need one bit on every object, in order to mark whether the object has been stored (depth-first) or whether it has entered the queue (breadth-first).

Both implementations need additional storage, stack or queue, which may grow to the size proportional to the number of objects. There is an important difference though: the system stack used by the recursive function usually has a fixed limit, but you can code the FIFO queue so that it increases its size when needed. If you have many objects the implementation with the fixed-sized queue may crash. That is the reason why Java serializations and some C# serializations crash for long chains of references.[26]

In order to traverse the graph of references, regardless which algorithm we use, we need three things:

(a) Location of all the pointers embedded in each object; this information is in the mask we described in Sect. 2.1.3.
(b) Type (and size) of the target object as allocated. When inheritance is used, this type may not agree with the type of the pointer.
(c) Additional bit or some other way to mark objects that have been recorded; otherwise the traversal may end up in an infinite loop.

[25] First In First Out.

[26] DOL and QSP persistence for Objective-C in Chap. 7 use the breadth-first implementation.

The standard way of solving item 2 is to have a virtual function which returns the type of the allocated object, either directly or through class Reflect—see online listing `list2_9x.cpp`.

There are three ways to provide the additional bit needed under item 3:

1. We can add a member *from below* or *from above* (Sects. 2.1.1.1 and 2.1.1.3).
2. We can keep a dictionary, for example a hash table, of references to all objects that have been stored (depth-first) or were admitted to the queue (breadth-first). This requires both additional storage and processor cycles at the part of code which repeats many times.
3. If you aim for ultimate performance, you can use the following, a rather dirty trick to store the required bit.

Useful Trick No. 4

All references and object sizes are multiples of four, with two lower bits never used.

One of these bits can record whether an object was already visited. If you have any control over the allocation, the obvious candidate is the field storing the size of the object. This field usually precedes the memory image of the object.[27]

Example of how this could be applied to a member[28]:

```
#define flagMask (size_T)3;
#define sizeMask ~(size_T)3;

class Book {
     size_t size; // always multiple of 4
     void setFlag(int flg){
          size=size&sizeMask;
          if(flg)size=size | 1;
     }
     int getFlag(){ return size&flagMask; }
     void setsize(int flg){
          size=size&sizeMask;
          if(flg)size=size | 1;
     }
     size_t getFlag(){ return size&flagMask; }
```

Now we have everything ready for the algorithm which collects all objects—see Listing 2.10. Function `getAllObj()` lives under the `Utility` class, and it builds a chain of `UtilLink` objects that point to recorded active objects.

Functions `setBit()`, `clearBit()`, and `getBit()` provide access to the special bit. The mask of regular pointers has been converted into a more convenient format—a table where each entry gives the offset of the next regular pointer.

[27] For more details see Chap. 6 where this trick is used in the QSP persistence for Objective-C.

[28] `size_t` is the same as `unsigned int` for 32-bit compiler, but 64-bit compiler it is `unsigned long long`, which is 8 byte long.

For example, if the mask is [H, int, R, R, float, R, int], where H is the hidden pointer, and R is the regular pointers, then tab[]={8,12,20,−1}.

Listing 2.10 The heart of the algorithm that expands, breadth-first, from the root to all other objects via their pointers. This sample program works only with pointers to single objects. It handles neither strings nor arrays

```
class PersistObj {
public:
        virtual Reflect *trueClass(){return NULL;};
        virtual void createMask(){};
};
// when we remember head and tail, this list works as a queue
class UtilLink {
public:
     PersistObj *obj;
     UtilLink *next;
     UtilLink(PersistObj *rt,UtilLink *last){
                              obj=rt; if(last)last->next=this;
                              next=NULL;}
};

UtilLink* Utility::getAllObj(PersistObj *root){
     UtilLink *u,*unew,*tail; PersistObj *p,*regPtr;
     int i,*tab,*code; Reflect *ref; void **locRegPtr; char *msk;
     createAllMasks(); // until this time they may not been needed
          ref=root->trueClass(); // get reflection for the target object
          root=ref->trueObj; // replace root by the true object
          allObj=new UtilLink(root,NULL); // root is first on the list
          for(u=tail=allObj; u; u=u->next){
               p=u->obj; // object to expand, it is a true object
               already
              Utility::clearBit(p); // before reflection, clear the
               bit
               ref=p->trueClass(); // reflection on the target object
               tab=ref->ptrOff; // offsets for pointers on p
               for(i=0; tab[i]>=0; i++){ // walk through regular pointers
                   locRegPtr=(void**)((char*)p+tab[i]); //location of
                   regPtr
                   regPtr=(PersistObj *)(*locRegPtr);
                   if(regPtr==NULL)continue; // do not follow, NULL
                   pointer
                   // skip when target on the list or when p==target
                   if(Utility::getBit(regPtr) || p==regPtr)continue;
                   ref=regPtr->trueClass(); // reflection on the target
                   regPtr=ref->trueObj; // replace regPtr by true object
                   unew=new UtilLink(regPtr,tail); // add to the chain
                   tail=unew; // remember the new tail of the chain
                   Utility::setBit(regPtr); // mark new object as expanded
          }
          Utility::setBit(p); // give p the "used" status again
     }
     // make all objects valid again by removing the bit
     for(u=allObj; u; u=u->next){
          p=u->obj;
          Utility::clearBit(p);
     }
     return allObj; // beginning of the chain
}
```

Note that in C and C++ (but not in C#. Objective-C or in Java) pointers can lead into the middle of an object. This can be result of multiple inheritance or of an

improper use of an embedded object as shown in Listing 2.11, where d spans over 56 bytes, between addresses 6044696 and 6044751, and pointers `a, b, c,` and `x` point to various locations inside this span.

The objects that we want to save should include only full, allocated objects, not their parts possibly overlapping or incomplete. In Listing 2.11 the virtual function `trueObj()` takes care of pointers such as `a` and `b` (it replaces them by `d`), but unfortunately it cannot correct pointers such as `c` or `x`. However, if such a pointer exists anywhere in your design,[29] there must also be a pointer to the entire D object which contains the small part.

Such duplications are easy to eliminate. Before writing the object to disk, do this:

ALGORITHM 2.1: Eliminate Embedded Objects from the List

1. Sort the objects by two keys:
   Priority 1: Increasing starting address
   Priority 2: Decreasing address of the last byte
2. Traverse the list while dropping embedded objects.
   Assuming we have an array of pointers to the objects, arr[], we do it like this:

```
for(i=0,k=1; k<numObj; k++){
if(arr[k]->start[k] <= arr[k]->end[i])continue;
//remove k
else {i++; arr[i]=arr[k];}
}
```

Listing 2.11 In C++, there are three situations when a pointer can lead inside an object—in case of multiple inheritance, when pointing to an embedded object or pointing inside an array of objects

```
class A {int a; };
class B { int b; };
class C {int c; };
class D : public A, public B {
public:
    int d;
    C cObj;
    C arr[10];
};
int main(){
    D* d=new D;
    A* a=(A*)d;
    B* b=(B*)d;
    C* c= &(d->cObj);      // bad practice, but it can happen
    C* x=(&(d->arr[7])); // bad practice, but it can happen
printf("sizeD=%d d=%d a=%d b=%d c=%d x=%d\n",sizeof(D),d,a,b,c,x);
// PRINTS sizeD=56 d=6044696 a=6044696 b=6044700 c=6044708 x=6044740
```

[29] It should not – it would be a poor design.

2.1.7 Java-Style Collecting Objects

As we can deduce from the output of Java serialization in Fig. 1.6, Java uses the depth-first algorithm which calls recursive function `serializeObj(root)`—this is a pseudo code[30]:

```
void serializeObject(Object obj){
     if(class.myClass is not serialized)writeClass(obj.myClass);
     mark obj as serialized;
     for all members m of obj do {
          if(m is a reference){
               if(object m already serialized){write reference;}
               else {serializeObject(m)};
          }
          else write m;
     }
writeObject(obj);
}
```

where `serializeObject()` recursively traverses inheritance hierarchy.

As we explained before, besides the performance penalty for calling a recursive function, this approach is vulnerable to stack overflow. For example, if you have a linked list of 100,000 objects, you may need 100,000 stack frames, and your program will crash with *StackOverflowError*.

2.1.8 Binary Serialization

We use the term *binary* for the serialization in which the byte images of the objects are written to the disk as they are. This is quite different from the binary Java serialization or the binary serialization in C# which creates and expands the description of each object and stores this description in a binary format.

Of all the approaches to persistence described in this book, only the binary option of the DOL library has used[31] this method. Yet it is simple, and as the benchmark in Chap. 8 shows, it is highly efficient.

DOL is based on the idea of integrating a library of data structures with persistence.[32] The application classes are not allowed to use members which are raw, plain pointers. All pointers are pre-registered in the library, so there is no need to detect them in the application classes.

When collecting active objects, the breadth-first approach is used, and when writing objects to disk, each object or array or objects is written in two records:

[30] The underlined functions are pseudo code. There are no functions with these names in Java.

[31] Since 1989.

[32] The idea was introduced in Chap. 1 and is discussed more in Sect. 2.1.3.4.

1. Header, as a block of bytes:

```
struct ObjectHeader {
     unsigned long objAddress; // starting address
     int objSize; // size of single object
     int numObjects; // 1 if single object
     int typeCode;
};
```

2. The object as the block of bytes.

Note that there are no generated object IDs. The original object address is used as its ID.

The pointers are swizzled when reading the data from the disk. Pairs (oldAddress, newAddress) are stored in a hash table, with the oldAddress used as the key. The table is used both as a container of all objects that were read from the disk and also for the conversion of the pointers to these objects.

With buffered IO, the disk access is reasonably fast. Figure 2.6 shows the typical format of the output file.

2.1.9 ASCII Serialization

The serialization which stores objects as blocks of bytes is highly efficient in both speed and data footprint. However, when transferring data from one operating system to another, for example between MS Windows and UNIX or Apple, binary data is meaningless unless you provide an automatic format translation. However, an ASCII text file usually works without a special conversion. This is one of the reasons why C# and Java provide XML serialization. ASCII format also allows visual reading of the file, which helps debugging.

The problem with generating ASCII representation of objects is that, especially in C languages, some types may need a different representation depending on the context. A byte can be a true ASCII character or a small integer; representing characters as numbers is inefficient and misleading when debugging, and if the character represents a number, some values will be unprintable characters. With float numbers there may be a question of accuracy. My experience is that pointers can be safely stored as (unsigned int) or hex, but that for other fields it is better to let the application programmer decide about their storage format.

That brings us to the problem we encountered with a large business system where serialization and deserialization functions represented one-third of the code—see footnote 3 in Chap. 1. Maintaining separate serialization functions for writing/reading is dangerous. If the two functions do not match, everything breaks down.

ASCII serialization in DOL[33] (Data Object Library 2013) has complete control of both hidden and regular pointers, and it stores/restores them transparently and

[33] In DOL, some macros have different names, but for the sake of clarity we use macros that we have been using so far.

```
+++++++++++++++ GENERAL INFO  (block of 28 bytes )
style = style of persistence
time = time stamp  (8 bytes)
numClasses = number of classes
++++++++++++++++ NEXT CLASS +++++++++++++++++++++
ClassHeader (block of 12 bytes)
mask1 = mask of pointers (objSize bytes)
mask2 = mask of inheritance (objSize bytes)
nameString  (size defined in the header)
++++++++++++++++ NEXT CLASS +++++++++++++++++++++
ClassHeader (block of 12 bytes)
mask1 = mask of pointers (objSize bytes)
mask2 = mask of inheritance (objSize bytes)
nameString  (size defined in the header)
++++++++++++++++ NEXT CLASS  +++++++++++++++++++++
        .....  repeats numClasses -times
++++++++++++++++ NEXT OBJECT ++++++++++++++++++++
ObjectHeader (block of 12 bytes);
Object ... raw block of bytes
++++++++++++++++ NEXT OBJECT ++++++++++++++++++++
ObjectHeader (block of 12 byte s);
Object ... raw block of bytes
++++++++++++++++ NEXT OBJECT ++++++++++++++++++++
        .....  repeats until the end of file
```

Fig. 2.6 Format of the DOL binary serialization file. All records are binary

automatically. The user supplies only the format for the remaining fields such as characters, floats, and signed/unsigned integers, and a simple code generator creates pairs of serialization functions that are guaranteed to match. Here is how Employee and Manager objects are managed in the application code, and what is the resulting disk image:

```
// PERSIST(T) manage pointers such as next,tail,secretary
class Employee {
     PERSIST(Employee);
     float salary;
     int phone;
};
FORMAT(Employee,"%6.2f %d",salary,phone);


class Manager : public Employee {
     PERSIST(Manager);
     char deptID[4]; // string of up to 3 characters
};
FORMAT(Manager,"%3s",deptID);


// Invocations of the data structures from a library
RELATION_RING(Manager,Employee) myEmployees;
RELATION_ONE_TO_ONE(Manager,Employee) toSecretary;
```

Image of a Manager object on the disk file:
Line 1 (address, class, how many): `6044696 13 1`
Line 2 (automatic pointers—next, tail, secretary): `6045012 6044540 6045084`
Line 3 (user controlled—salary, phone, deptID): `10450.50 6133885211 A23`

2.1.10 Deallocation and Garbage Collection

The great advantage of serialization is that it does not require any garbage collection or special deallocation techniques. During the program run, objects are dynamically allocated and deallocated through calls to `malloc()` and `free()` which are hidden under the operator `new()` and `delete()`. And because only active objects are written to the disk, the serialization itself works as a space-cleaning mechanism.

2.2 Memory Paging

Persistence based on memory paging is a good alternative to serialization. We allocate objects from pages of memory, and when storing the data we move entire pages between the memory and the disk, without looking at individual objects. This method is fast and space efficient, but it must take over both allocation and reuse of the free space including arrays. Since we are not saving individual objects, we need a different mechanism to identify pointers and, for this purpose, a special bitmap can be handy.

2.2.1 Bitmap

The mask which we used in serialization clearly identified pointers inside any object, without paying attention to inheritance and embedded objects. It gave us a flat view with positions of the pointers clearly visible.

Perhaps we can apply a similar idea to the entire data space, and instead of saving individual objects, we could save the entire data space, in one shot, as a large block of bytes. The only thing we would need would be a mask that would show us where are the pointers that we have to swizzle. If that mask uses one bit for each potential location of a pointer with addresses divisible by 4 (or 8), the mask would add the overhead of only 1/32 (or 1/64) of the data space—a quite reasonable price to pay for the service we'll get—see Fig. 2.7.

For example, assume that we have an address space of 65536 bytes, from 52004 to 117539 and at address 70104 we allocate a 20-byte object with 3 pointers offsets

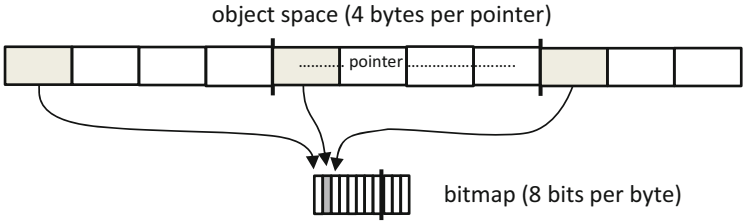

Fig. 2.7 Mapping potential pointer locations from the object space into the bitmap

{0,8,16}. The address space has 65536/4 = 16384 potential pointer locations, so the bitmap needs 16384 bits = 16384/8 = 2048 bytes.

When allocating the new object at address 70104, the pointer locations are 70104, 70112 and 70120, and the following bits must be set: (70104 − 52004)/4 = 4525, (70112 − 52004)/4 = 4527, and (70120 − 52004)/4 = 4529.

We have to mark both the hidden and the regular pointers because both must be swizzled, each using a different algorithm.

Note that when using smart pointer `PersistPtr<T>` explained in Sect. 2.1.2, updating of the bitmap is especially efficient. The default constructor of this pointer can automatically set the appropriate bit in the bitmap.

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

When swizzling pointers during serialization, we had to swizzle only regular pointers. We knew the type of each object, so we could just copy[34] hidden pointers from the mask of its class.

When we work with a block of memory, swizzling is more difficult. We have to distinguish between hidden and regular pointers, and we cannot copy hidden pointers from the mask because we have no clue which mask would apply.

A hidden pointer can be recognized by its value—it must be in a narrow address range of the virtual function table for the old data.

Since the introduction of C++ in the early 1990s, all C++compilers used the same convention. If two programs shared the same *.h files and listed them in the same order, the virtual function tables were identical, except for usually being in a different memory location. The conversion of hidden pointers was easy: after we

[34] This was done by a special operator new().

detected a hidden pointer, we added an offset which was the same for all the hidden pointers.

This year some applications using DOL memory blasting[35] occasionally crashed with a mysterious error, which sometimes did not repeat. After a week of detective work we found that Microsoft Visual C++ 2010 usually maintains the same v.f. table but, for unknown reasons,[36] it may change the order in which the classes are listed in that table. Usually, the table entries are uniformly spaced, but we encountered one case when they were not—by mere 4 bytes, but enough to confuse our original, simple algorithm. Replacing it was not trivial, because swizzling of hidden pointers is typically performed for every active object, so the performance matters.

The new DOL algorithm first checks whether all the old/new pairs fit the uniform-offset pattern. If they do, it uses the offset. If they do not, it uses an algorithm which is easiest to explain by an example:

Let's assume that we have four classes and that we know the values of their hidden pointers—both the old ones (before saving to disk) and the new ones (when reading the data from disk).[37] Assume that the old values are sorted[38]:

```
i   class           olddif   new
0   Publication     3359488  8799040
1   Journal         3359504  16 8799008
2   Book            3359536  32 8799056
3   Report          3359572  36 8799024
```

In this case, the range of the original pointers is $335572 - 3359488 = 84$, which is different from the new pointers $8799008 - 8799060 = 48$. Also, the old pointers are not uniformly spaced.

We make a sparse table `sTab[]` with $(3359572 - 3359488)/16 + 1 = 6$ entries, where 16 is the smallest value in the dif column. Then for each `i` we set `sTab[old[i]-old[0])/16]=new[i]`, which gives us the following table:

```
k sTab
0 8799040
1 8799008
2 0
3 8799056
4 0
5 8799024
```

The conversion is instant. For example, when converting old hidden pointer 3359536, we calculate $k = (3359536 - 3359488)/16 = 3$, and the new value is

[35] DOL binary and DOL ascii do not use bitmap.

[36] This could be because of the incremental compilation in VS2010.

[37] These values can be found from masks derived in Sects. 2.1.1 and 2.1.2.

[38] In order to demonstrate the algorithm, we disturbed the numbers more than when we encountered them in real situations.

`sTab[3]=8799056`. In real applications, we have not encountered a case where `sTab[i]=0` for more than one `i`.

2.2.2 Pages of Memory

The bitmap allows us to build simple yet highly efficient persistent data. We can allocate a large block of memory, and we set aside an additional, 32-times smaller block, for the bitmap. We modify the new() operator so that it allocates new objects from this block, and we make sure that all default constructors mark the bitmap for all the pointer members. When saving objects to disk, we simply dump the entire block to disk, together with the old address and type of the root object and the table of old hidden pointers.

When reading the data from the disk we allocate the same amount of memory, copy the disk content in it and swizzle all the pointers recorded in the bitmap. If there is only one block, all pointers are swizzled by the same increment. The bitmap is persistent—no swizzling is required.

This is so simple and efficient that you must be wondering why anybody would bother to use serialization. The weakness of this approach is the fixed size of the block. In real life applications, you rarely know how much space your data might require, and allocating a bigger block of memory, copying the old image in it and swizzling all the pointers may pause your program for long enough to make this approach prohibitive. After all, this is essentially the early-Smalltalk model from Chap. 1, only improved by the bitmap.

What we need is an arrangement which would use not one block of data but pages of virtual memory, with system pages still working behind the scene as usual.

The following scheme was proposed by Mark Kraemer from Zycad Corp. in 1993, was implemented as *memory blasting* in DOL, and was first published on pp. 379–386 in Soukup (1994).

The DOL implementation assumes that the size of these pages is a power of 2. This is only a minor performance improvement which allows frequently used division to be replaced by logical shift, and modulus operation by logical AND. The following description assumes that the page size, `pgSz`, can have any size which is A multiple of 4 bytes (or 8 bytes on 64-bit hardware).

The problem with this entire approach is that, for a given pointer, we need fast access to the page in which its target object is located. For this purpose, we keep array `pageStart[]`, where `pageStart[i]` stores the starting address of the page which starts anywhere between `i*pgSz` and `(i+1)*pgSz-1>`. In other words, page starting on address `p` is recorded in `pageStart[p/pgSz]`, using integer division. For any allocated page there is only one corresponding entry in pageStart[], and some pageStart[] entries may be 0; see Fig. 2.8. The best way to learn how this works is to go step by step through a simple example.

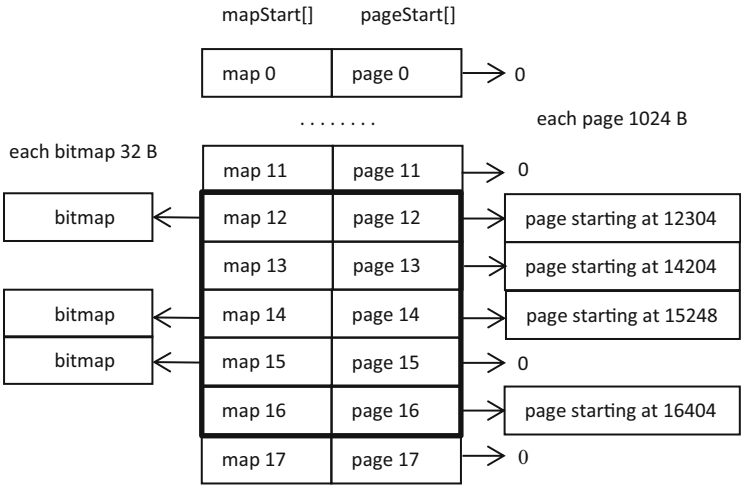

Fig. 2.8 Persistent memory consisting of pages pgSz=1024

Example[39]:

As shown in Fig. 2.8, assume we allocated four pages that start at 12304, 13584, 15248, and 16404, and we record them by dividing their starting address by the page size, for example 13584/1024 = 13.

Question: Where is pointer 15372?

Answer: 15372/1024 = 15, pageStart[15]=0, it is on page 14 (one step down).
Address inside page = 15372 − 15248 = 124, bit number 124/4 = 31

Question: Where is pointer 15260?

Answer: 15260/1024 = 14, pageStart[14]<=15260, it is on page 14.
Address inside page = 15260 − 15248 = 12, bit number 12/4 = 3

Question: Where is pointer 15208?

Answer: 15208/1024 = 14, pageStart[14]>1508, it is on page 13 (step down).
Address inside page = 15208 − 14204 = 1004, bit number 1004/4 = 251

Example[40]:

Assume that data from Fig. 2.8 was stored on disk, and we are restoring the data. Typically, the pages are allocated to completely different locations—they may not be in the same order. Let's see how we swizzle regular pointers if the new pages are as shown in Fig. 2.9.

Question: Convert old pointer value 14228.

Answer: 14338/1024 = 13, 14228 > 14202, old page is 13.
Address within the page is 14228 − 14204 = 24
From Fig. 2.9, old page 13 corresponds to new page 52
New address is 53748 + 24 = 53772

[39] This is the same example as used in Soukup (1994) on pp. 381–382.

[40] This example is *not* in Soukup (1994).

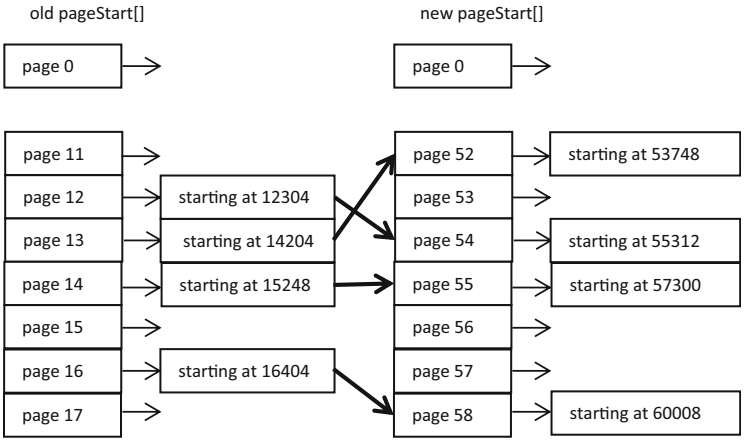

Fig. 2.9 Assignment of pages when reading data from disk can be random (example)—see *heavy arrows*. Page size must remain the same, here 1024 bytes

Potential improvements and interesting details

1. The beginning of arrays pageStart[] and mapStart[] is usually unused, for example entries 0–11 in Fig. 2.9, and entries 0–51 for the new pageStart[] in Fig. 2.9. The idea is to start the array from the index corresponding to the first page, and be ready to shift the assignment if some page has a lower address.
2. There is no need to store on disk the unused entries –those with pageStart[i]=0. You can either store the used section (heavy frame in Fig. 2.8) as it is, or store the array as a sparse array.
3. Bitmaps are persistent; they do not need any swizzling.
4. The size of arrays mapStart[] and pageStart[] depends on the original estimate of the total expected data space. If this estimate is exceeded, we do not have to reallocate the existing pages, only these arrays. In order to make this fast, we recommend to select the page size close to the realistic estimate of the required space—without any safety. That way you end up with one or a few pages, and even if the arrays have to be re-allocated, it is fast.

Useful Trick No. 5

If your objects have many non-structural members, detecting pointers by traversing the bitmap is not efficient, because the algorithm walks through many 0s before it hits a pointer. We can speed up this search significantly by treating the bitmap as an array of integers. Only if an integer is not 0 do we examine its bytes, and only if a byte is 0 do we examine its bits.

2.2.3 Dynamic Allocation and Garbage Collection

Any of these techniques work quite well until you start to destroy objects. Unless you manage free space, the memory and disk footprints may grow out of control.

Real life story

One of our consulting appointments was to examine the core of a telephone switch and try to improve its speed. Telephone switch is a computer which processes human voices converted to a stream of numbers, multiplexed and transferred in packages. The system was written in a special object-oriented language (not C++), and each telephone call created hundreds of objects which, often within seconds, were again deleted.

We suspected that the creation and especially destruction of all these objects took a long time, so we made the following experiment. We requested a block of memory at the beginning of each phone call, and allocated all the objects from it.[41] *Instead of the destruction of individual objects at the end of call, we simply freed the entire block of memory. It made the switch 30 % faster!*

Then we tried a different strategy. For each class, we kept a linked list of free objects. Instead of allocating new objects, we picked the ready-made objects from this list and, instead of destroying them, we hooked them to the list by resetting two pointers. It was even faster!

We learned two lessons: (a) When looking for high performance, do not underestimate the time needed for creation and destruction of objects. (b) Keeping free lists of objects by class is very efficient.

The problem with building persistence on pages of memory is that you must take over memory management including the disposal and reuse of objects. And this memory management must be persistent. For example, the disk file must record the beginning of each list, and the pointers connecting the free list must be marked in the bitmap.

A highly effective and simple to implement method of reusing objects is to keep, for each class, a list of discarded objects. If the list is empty, operator new() allocates the next object from the last, not completely used page. When it is not empty, it just picks up an object from the beginning of the list. Operator delete() always attaches the unwanted object to the list.

If you want to be able to mix non-persistent objects with persistent ones, you need new() and delete() for persistent objects, and a set of different functions, say npNew() and npDelete(), for non-persistent objects which are managed outside your memory pages, directly from the heap.[42]

[41] This is sometimes called *arena allocation.*

[42] You can override delete() but you cannot overload it with different parameters.

When an object is discarded, we believe it is not a dirty technique to use its first 4 bytes for the pointer that forms the free list. When the object is being reused, we only have to correct the first 4 bytes if there is a hidden pointer. The free list works like this:

```
class Book {
    Book *freeList;
    void addFree(Book *b){
        *((Book**)b)=freeList;
        freeList=b;
    }
    Book *getFree(){
        Book* b=freeList;
        if(b)freeList=(*((Book**)b);
        return b;
    }
```

There are several ways to make free lists persistent, but only the following method is both conceptually correct and has the ultimate runtime performance. When moving an object to the free list, we reflect the change of its status by changing its fingerprint in the bitmap to 100...00, essentially registering only one pointer at the beginning of the object. When reusing an object (and removing it from the free list), its bitmap record will go back to the fingerprint of the particular class. This way, the free list pointers will be automatically swizzled with other pointers.

Note that if every object starts with a hidden pointer as happens in DOL, the first four bites of any object are already marked as a pointer in the bitmap. Thus without any special action, the free list is automatically persistent. However, some of the old pointers that may still be in the object image will go through the swizzling uselessly and will make it slower.

Listing 2.12 shows the implementation.

Listing 2.12 Keeping chains of free objects for each class[43]

```
// first 4 bytes of the object represent the 'next' pointer
#define NEXT(P) (*((void**)P))

class A {
    static void *objHead; // first 4 bytes of the prototype
    static void *freeTail;
    //...
public:
    void* operator new(size_t sz){
        void *p;
             ...
        if(!freeTail) p=allocateFromPage(sz);
        else {
            p=NEXT(freeTail); // from the chain of free objects
            if(p==freeTail)freeTail=NULL;
            else NEXT(freeTail)=NEXT(p);
            NEXT(p)=objHead;
        }
              restoreBitmap(p); // restore bitmap to valid obj.
              return (A*)p;
    }
    void operator delete(void *p){ // does not destroy the object
        setFirstBit(p); //set bitmap to 100..0 for this object
        if(!freeTail){NEXT(p)=p;} // puts it on the free chain
        else {NEXT(p)=NEXT(freeTail); NEXT(freeTail)=p;}
        freeTail=p;
    }
    int main(){
        A* ap=new A;
        delete(ap);
```

Note that this handles the reuse of single objects but not of arrays.

Note also that memory paging and serialization do not exclude each other. Serialization can traverse all objects regardless of how they were allocated; it is a handy tool which cleans the memory pages of any non-active objects that may be accidentally left there. When deserializing the data we only have to make sure that the new objects are allocated from our pages.

This can be arranged by overloading operator new() for all the application classes and controlled by a global switch, pgAlloc:

```
class Book {
    void* Book::operator new(size_t sz){
        if(pgAlloc) … // allocate from pages
        else … //allocate with malloc() or as char[sz]
    }
```

[43] The online code does not show the adjustments to the bitmap.

The purpose of pgAlloc is to allow serialization to operate in two modes: standalone or alternating with memory paging. Alternating memory paging with serialization is a good practice, because serialization automatically removes free lists, all garbage, and it defragments the data space.

So far, we have not discussed allocation and free storage of arrays. In C++ there are two types of operators new() and delete(): the static operators which are associated with some class and usually allocate/delete individual objects and the global operators which allocate/delete arrays.

Allocation of arrays brings the following challenges:

1. Can we allocate arrays that are larger than our page?
2. How to reuse arrays? Could we merge or split them?
3. A fast algorithm for finding a free array of the required size is instrumental.

When an array is larger than one page, we can allocate several abutting pages as one large block memory, and allocate the array across the page boundaries. All pages of this set must be marked so that when reading them from disk, they will be again adjacent to each other. Their bitmaps work as usual.

These problems have been well researched, and it depends on you and your application how fancy a management of the free space you chose. Because serialization provides defragmentation and cleanup, we are in favour of a simple free storage.

One possible way to manage free arrays is to use another bitmap,[44] which marks both ends of each free array—see Fig. 2.10.[45] When freeing an array, the bitmap tells us whether the new array butts to a free space on either end, and the record in the adjacent field tells us how large that free space is. Without any search or expensive calculation, we can combine adjacent free spaces.

The assumption for all this is that arrays are allocated from a different part of memory than single objects—always an array abutting on an array, which is a common practice today. Arrays must be at least 16 bytes long, because they have to be doubly linked—when reusing an array we may select an array from the middle of the chain. These arrays must be persistent, which affects the bitmap maintenance while moving arrays to/from the free list.

A more elaborate approach is shown in Fig. 2.11. Single free objects are stored by class, short arrays and strings by size, and large arrays are stored in a height-balanced binary tree, which is O(log n) for lookup, insertion, or removal. For specific differences see Wikipedia. (Knizhnik, POST++, 1999) uses AVL tree; DOL keeps chains of free objects but does not reuse arrays.

A frequently used improvement is an array of entries for short, frequently occurring arrays such as text strings, where entry for index `i=(sz/4-1)` leads to a chain of short objects of size `sz`; see Fig. 2.11.

Short arrays including short text strings are allocated and reused in the same way as single objects, without recombining them.

[44] This means an additional bitmap which is different from the one we used to mark pointer locations.

[45] Ending beyond the page border.

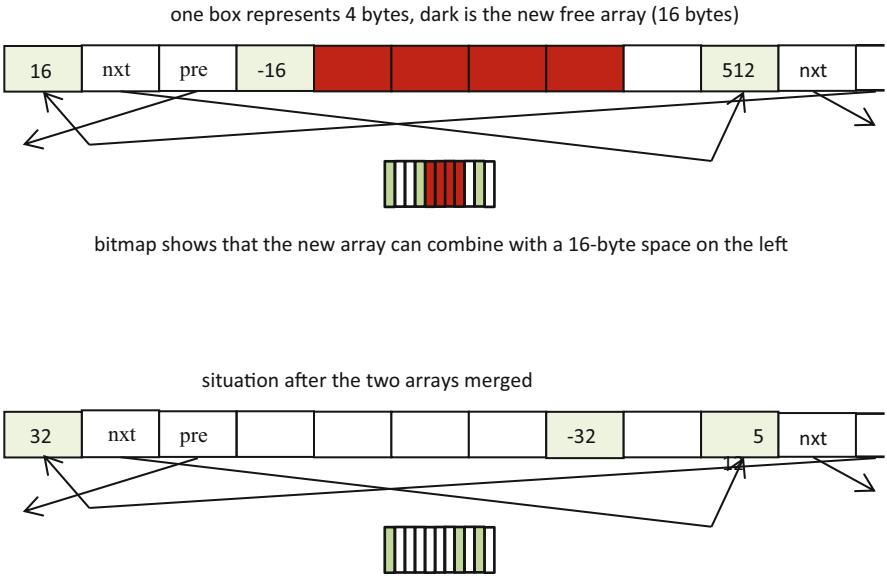

Fig. 2.10 Merging abutting free arrays: we start with two free arrays, one 16 bytes long, the other 512 bytes long, ends marked by *light colour* in the bitmap. *Boxes* shown in *dark colour* represent the free array which we want to add. Bits in the bitmap tell us that we have an abutting array on its left, 16 bytes long. This allows us to merge the two arrays. Pointers `nxt` and `pre` form a doubly linked list of free arrays

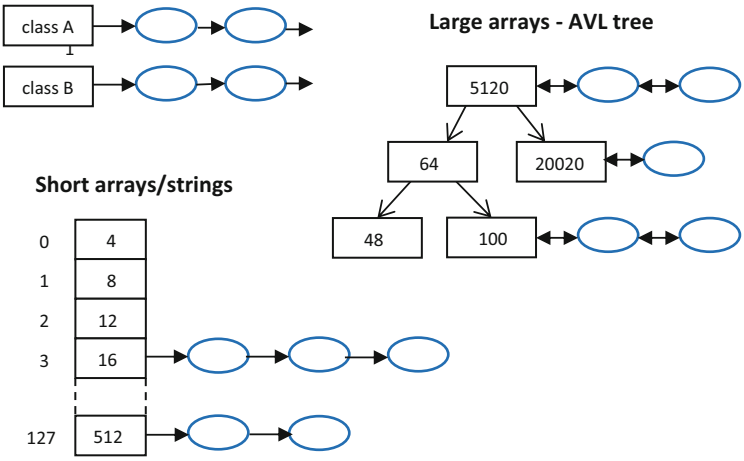

Fig. 2.11 Free storage for arrays including text strings. Short arrays of each size form a singly-linked chain and are allocated from the same part of memory as simple objects. Long arrays are allocated from a different part of memory, and arrays of the same size form a linked list. Their ends are marked by their size and are recorded in the bitmap, and they can recombine with abutting free arrays

2.3 File Mapping

Many operating systems including Windows and UNIX provide a function which maps a selected section of virtual memory to a disk file. Instead of implementing persistent objects with our own pages, as we did in Chap. 2.2, we can implement it with system pages of the file mapping function—essentially keeping a mirror image between the two entities. However, the data is not transferred as one block, it moves back and forth in system pages. The advantage is that these pages will naturally support transactions. The disadvantage is that you need some knowledge of system programming.

Before we dive into the programming details, let's look at the main concept which is simple. We establish a mapping between a section of virtual memory and a disk file, and then allocate all objects from this section of memory. When the application stops, all the objects are stored on the file. Then when we start the application with the same mapping, the objects magically move back to memory again!

One little detail, though. All this would work if, when reading the data from the disk, we could use the same section of virtual memory. Most of the time, this is not, and We really should swizzle all the pointers by the offset between starting addresses of the old and new memory section.

That again is not trivial. The data does not move between the disk and memory as one block, but as system pages. We could go through the disk data and swizzle the pointers there, but that would not be very efficient. Instead, we have to swizzle the pointers as they are loaded into memory.

It works like this. When the application traverses a pointer, the system checks whether the page with the address stored in the pointer (the address of the target object) is already loaded in memory. If it is not, it triggers a page fault which results in loading the page.

At this point we must swizzle the pointers before the control is returned to the application. For example, we can catch the page fault, copy the page from the disk without leaving this to the system, swizzle the pointers on the page, and return the control back to the application.

That assumes that we can find where the pointers are in the new page, and we have already discussed that at length. Pages may also move in and out of the memory, so it may happen that the page, which re-enters on the page fault already has the pointers swizzled. Swizzling them the second time would make them incorrect. One method to prevent that, is to allow each page to store its own starting address. If this address agrees with the address to which the page is being loaded, swizzling is not required.

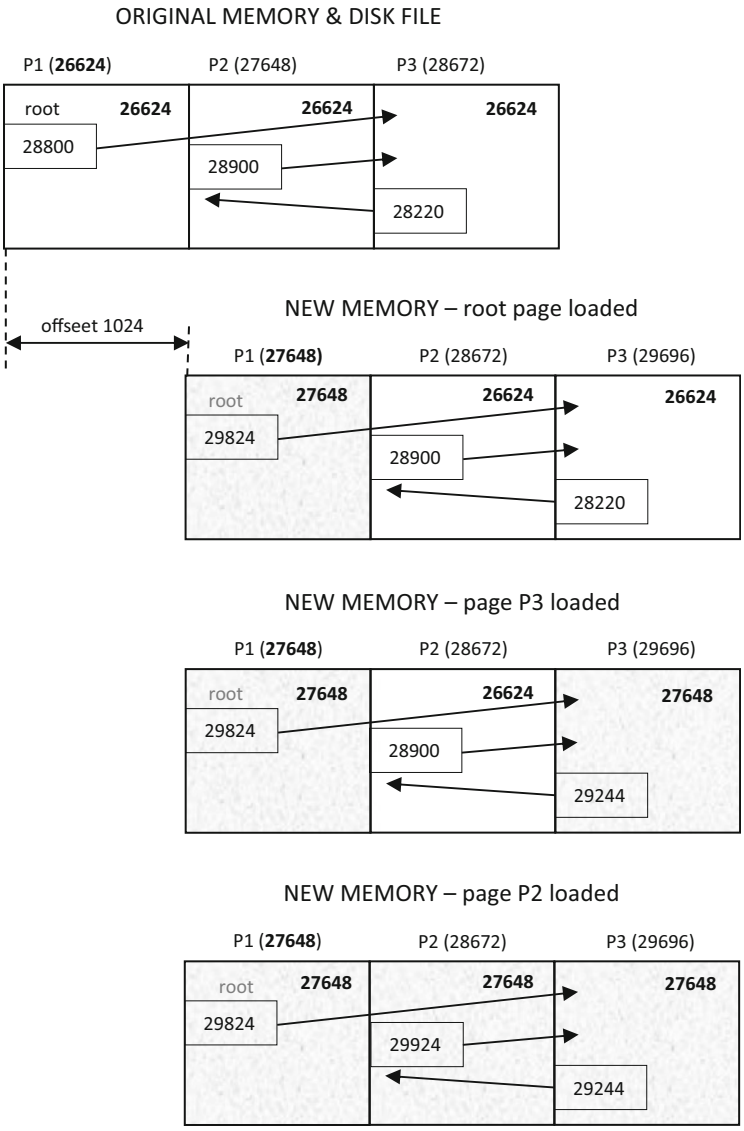

Fig. 2.12 Persistence based on file mapping (the concept). Every page remembers the P1 address (*bold*) at the time of its last pointer swizzling (upper-right corner of the page box)

Figure 2.12 demonstrates the idea. We start with the root page loaded and the 28800 pointer swizzled by the offset of 1024 to 29824. Traversing this pointer triggers page fault because page P3 is not yet loaded. After is it loaded, all its

pointers are swizzled by the offset, including the original 28220 which becomes 29244. When traversing this pointer, it triggers page fault for page P2, which is then loaded and its pointers swizzled. The original 28900 is now 29924. That pointer leads to page P3. If it is loaded, there is no page fault and everything runs smoothly and fast. If P3 is not loaded, we get a page fault, we load it, but we do not have to swizzle its pointers because we see that the beginning of the current memory section, 27648, agrees with the number recorded for this page—it is the number recorded at the right upper corner of the page.

The actual implementation is more complicated. File Mapping pages the disk file to file views which are in the virtual memory of individual processes.[46] Several processes may map to the same disk file simultaneously. A view can mirror the entire file or only its section.

This entire feature was clearly designed to facilitate a design of true databases, which may store large amounts of data and be accessed by multiple processes, often simultaneously. The purpose of View is to allow access to only a small part of an otherwise large collection of data, for example in bank or airline reservation transactions.

As explained in Sect. 1.1, this book is about persistent data that are accessed by only a single process at any given time, and for this reason all the following discussion will assume a single process accessing a disk file, and Listing 2.13 shows in code what we explained in Fig. 2.12.

Listing 2.13 is an excerpt from programs `List2_13a.cpp` and `List2_13b.cpp` which are online under bk/chap2. Read it without worrying about the numerous parameters that make the use of these functions a bit tricky. It is in the call to `MapViewOfFileEx()` that you can specify the address where you would prefer the data to start. As we explained, the function may not satisfy the request, but, if it does, there is no need for pointer swizzling.

Listing 2.13 Example of File Mapping under Windows. I order to time and test parts of the algorithm separately, the online version of this program consists of three source files: List2_13a.cpp, List2_13b.cpp, and List2_13c.cpp

[46] Each process has its own independent virtual memory.

```
HANDLE fh; // file handle as for normal disk IO
unsigned baseAddr; // requested address for the beginning of the data
void *newBase; // beginning of the data in the virt.memory
char *p;
unsigned vmSZ; // size limit of the data

// Instead of using create() or open() we have to use CreatFile(),
// otherwise CreateFileMapping() does not work.
// Use OPEN_ALWAYS or OPEN_EXISTING when creating or openning file.
fh=CreateFile(fName,GENERIC_READ|GENERIC_WRITE,0,NULL, OPEN_ALWAYS,
              FILE_FLAG_WRITE_THROUGH|FILE_FLAG_RANDOM_ACCESS, NULL);

// create a mapping object for the file, 0=offset on the disk
md=CreateFileMapping((HANDLE)fh,NULL,PAGE_READWRITE,0,vmSZ,NULL);

// Attempt to create a view starting at baseAddr
newBase=MapViewOfFileEx(md,FILE_MAP_ALL_ACCESS,0,0,0,baseAddr);

// If not successful, let the system to chose the new base address
if(newBase==NULL){
    newBase=MapViewOfFileEx(md,FILE_MAP_ALL_ACCESS,0,0,0,0);
}

// Examples of use:
p=(char*)newBase;
*((int*)(p+20032))=1937; // insert 1937 at address (newBase+20032)

float f=(*((float*)(p+11996))); // get float from (newBase+11996)

// flush all remaining pages to the disk
FlushViewOfFile(newBase,totSpace);

// without this, the file remains open
  UnmapViewOfFile(newBase);
```

UNIX (and Linux) has a similar set of functions.[47] Using parameter names as in Listing 2.13, they are

```
// combines CreateFileMapping() and MapViewOfFileEx()
newBase=mmap(baseAddr,vmSZ,protect,flags,fh,offset);

// equivalent of UnmapViewOfFile()
munmap(newBase,vmSZ);

// change size of mapping (Linux specific)
newBase=mremap(newBase,vmSz,newSize,flags);

// equivalent of UnmapViewOfFile()
msync(newBase,vmSZ,flags);
```

Figure 2.13 shows where UNIX maps the file. If you want to play with the Windows functions, beware: possible interaction between processes makes their use trickier and it is, in our opinion, poorly chartered territory. The online rating of

[47] For details, see UNIX man pages or http://my.safaribooksonline.com/book/operating-systems-and-server-administration/linux/0596009585/advanced-file-i-o/mapping_files_into_memory

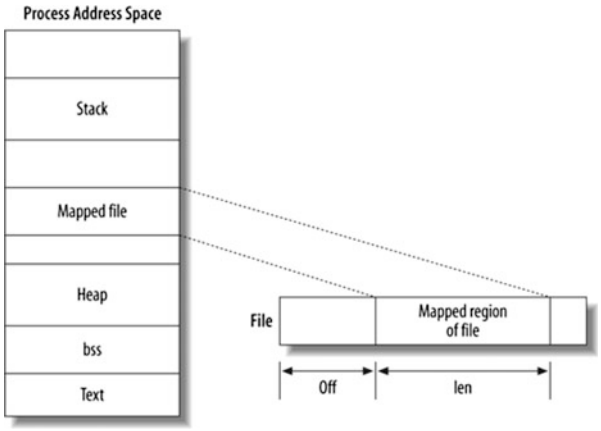

Fig. 2.13 File mapping under UNIX

Microsoft documentation ranges from 4/4 to 1/4. Mykhailo Oksenenko recently posted[48] several screens of errors and potential problems in their use, and even the Microsoft documentation for *Reading and Writing from a File View* warns that an attempt to use File Mapping of a sparse file of an NTFS partition may result in an I/O error.

There is also a catch: You can map the disk to the entire virtual memory, but when you open a View, you have to request a certain size. You cannot exceed or increase this size later,[49] so you typically request more memory than you need. However, only the used portion of your data is saved to the disk.

Programs available online (`List2_13a.cpp`, `List 2_13b.cpp`, and `List2_13c.cpp`) compare the speed of storing 300 MB of data for File Mapping and regular read () and write() with these results in ms[50]:

	Changes on every page	Write	Read[51]	Total
File mapping	*194*	*2682*	*3008*	*5884*
Read/write	*149*	*2763*	*2815*	*5727*

It is important to understand that you cannot compare numbers in the first three columns, because the two approaches do different things at different times. When making a change on every page, *file mapping* loads page by page, while *read/write* only changes a memory location. On the other hand, when writing to disk, *file mapping* writes only the remaining pages, while *read/write* writes the entire data space. When starting a program, *file mapping* only has to set up itself without moving any data, while *read/write* reads all the data to memory.

[48] http://mikelaud.blogspot.ca/2010/01/shm-hints-windows.html

[49] Under Linux you can, but it may stall the program execution for a while.

[50] Timed on the computer which was used for the benchmark in Chap. 7.

[51] After restarting the computer. If you read immediately after you wrote, the data is still in the system cache, and you get only 80 and 228 ms.

The speed of processing clearly is not the reason to use File Mapping for persistent data, but if we need to access only a few pages from an otherwise large amount of data, File Mapping will move only those pages and will definitely be more efficient than serialization.

Both ObjectStore (c) PSE Pro for C++ and POST++ are based on File Mapping.

As in the old Smalltalk model, when using POST++, the user must supply an estimate of the data size. If the task exceeds this size, the program will crash or it may pause for a long time.[52]

PSE does not require any initial size estimate, and we do not know what is under the hood. One can only guess that when PSE needs more the data space, it perhaps generates multiple Views—in the same style as when we worked with pages of virtual memory in Sect. 2.2.2. Listing 2.13 opens only one File Mapping and one View, but the online version of this program[53] tests opening several Views simultaneously.

This approach has been used for two decades. Singhal (1992) reported on a university project called Texas, to which we did not find any references after 2000. Soukup (1994)[54] showed four pages of code that demonstrated the idea in UNIX. QuickStore was described in White and DeWitt (1995). Free[55] software (Knizhnik, POST++, 1999) also uses this approach and is free to download from their website. For more information on these projects see Sect. 4.2. For more information on ObjectStore look at Lamb et al. (1991), Zikari (2010) and Haradhvala and Weinreb (1991).

2.4 Persistent Pointers

Until now, we assumed that the prime location of data was in memory and the pointers in the disk image stored the original memory addresses, and we had to swizzle pointers after we loaded the data into memory. This section assumes that the prime location of the data is on disk, and the pointers store the disk addresses. This implies a few arithmetic operations when dereferencing a pointer, but no pointer swizzling is required. The secret is a smart pointer class that makes all this transparent.

[52] The latter happens specifically under Linux.

[53] bk\chap2\list2_13a.cpp.

[54] pp. 386–392.

[55] 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.

2.4.1 The Main Idea

Imagine what would happen if, for every class, we would pre-allocate a large array of its instances, and when we would need an object or an array of that class, we would bypass allocation and simply pick it up from this array—see Listing 2.14.

Normally, pointer stores the starting address of the object to which it is pointing, but if we stored the index into this array, it would be persistent. It would not require swizzling, and we still could access the object very quickly.[56] For the implementation of the smart pointer which does this, see Listing 2.15.

Listing 2.14 Allocating an array of objects for each class

```
class Book {
    static Book *myArr; // preallocated array
    static size_t pool; // next available index
    void* operator new(size_t size){ // for a single object
                                if(!myArr)return (void*)calloc(1,size);
        void* v=(void*)(myArr+pool);
        pool=pool+size/sizeof(Book);
        return v;
    }
    void* operator new[](size_t size){ // for an array of objects
         //...identical with new()
    }
    static void start(size_t sz){
        myArr=new Book[sz];
        pool=1; // index=0 corresponds to pointer==NULL
    }
};
Book* Book::myArr=NULL;
size_t Book::pool=0;

int main(){
    Book::start(1000);
    Book* bp=new Book();
```

[56] The first object in the array could be unused, thus making index 0 equivalent with NULL pointer.

Listing 2.15 Persistent pointer storing the index

```
template<class T> class PersistPtr {
  unsigned index;
public:
  T* operator->() const{
    if(index)return (T::myArr+index); return NULL;
  }
  T* operator*() const{
    if(index)return (T::myArr+index); return NULL;
  }
  PersistPtr& operator=(T *rhs){
     if(rhs)index=((size_t)rhs-(size_t)(T::myArr))/sizeof(T);
     else index=0;
     return *this;
  }
  size_t getIndex(){return index;}
  void setInex(size_t i){index=i;}
};
class Book {
public:
     int id;
     PersistPtr<Book> next; // note no * in the syntax
       ... same as in Listing 2_12
};
Book* Book::myArr=NULL;
size_t Book::pool=0;

int main(){ // examples of different use
    PersistPtr<Book> bp1,bp2; // persistent pointers, no *
    Book *br;                 // regular pointer, use *
    Book::start(100); // start the allocator array for 100 Books
    bp1=new Book; bp1->id=1; // reg. to pers.conversion, use ->
    bp2=new Book; bp2->id=2;
    bp1->next=bp2; // use persistent pointers like reg.pointers
    br= *bp1;       // persistent to regular pointer conversion
    printf("%d %d\n",br->id,bp1->next->id);
```

Using a pre-allocated array of objects has many advantages:

(a) Elimination of unused objects requires reference swizzling but, in this case, such swizzling is simple and fast.
(b) If we want to visit all the objects (as in serialization), instead of traversing the network of pointers, we can traverse these arrays which is much simpler and faster.
(c) Serialization to disk for an array is more economical than for individual objects: it needs only one header.
(d) If we want to move entire pages of data between memory and disk, we don't need a bitmap to locate the pointers. For each class, we can visit the first pointer of all objects, then the second pointer of all objects, and so on.
(e) Page size can automatically adjust to a multiple of the object size, thus reducing problems with large arrays that cross page boundaries.

There are only two issues we have not discussed yet:

(1) Could unstructured, passive[57] objects—with the wide variety sizes such as text strings or arrays of integers—fit this scheme?
(2) When the preallocated array is all used up, can we enlarge it without copying the old array into the new one?[58]

Let's start with implementing each allocation array as a set of pages, which are treated as a "virtual" array. The smart pointer has to perform several arithmetic operations in order to dereference a pointer. That may not have a big impact on the overall performance, but you should be aware of it.

Even if we manage free space for potential reuse, it may be useful to be able to clean up all unused objects and to compress the memory—for example, when ending a session or when we save the data to disk. The following algorithm shows how to do that. After we shrink the arrays for all the classes, we have to swizzle[59] all their pointers, using array `conv[]` from this algorithm:

Alogrithm: Removal of Unused Objects by Shrinking the Array

Just before saving the data to disk, `Book::myArr` has free spaces xxxx

```
                0    1    2    3    4    5    6    7    8
myArr[] = obj1 obj2 xxxx xxxx obj3 xxxx obj4 obj5 xxxx
```

We shrink `Book:: myArr[]` and create a temporary conversion array,

```
size_t Book::conv[]
                0    1    2    3    4    5    6    7    8
myArr[] = obj1 obj2 obj3 obj4 obj5 xxxx xxxx xxxx xxxx
conv[] = 0    1    -1   -1   2    -1   3    4    -1
```

We do this for all classes, and then traverse all objects and convert their pointers. When pointer `ptr` points to a Book object, we convert it like this:

```
PersistPtr<Book> ptr;  // persistent pointer to Book
size_t k=ptr.getIndex(); // the old index
ptr.setIndex(conv[k]);  // index conversion
```

Each object is visited only once, and no search is required

The key to managing the free space is having all free pieces doubly-linked so we can quickly insert and remove both objects and arrays. The key to defragmentation is being able to combine butting free spaces into a single larger space.[60] The following example shows one of many possible implementations.

[57] Objects that do not harbour any pointers.

[58] Copying would trigger changes in all objects pointing into this array.

[59] Remember that, in this case, pointers store the integer index into the array, not the object address.

[60] The difference from the normal memory management is that here we deal with arrays of objects that have the same size.

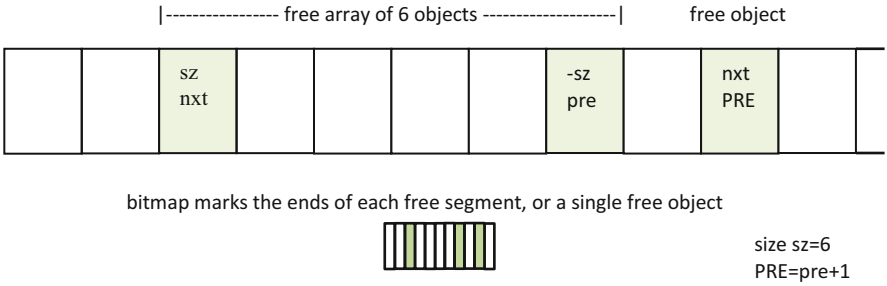

Fig. 2.14 Space representation of free objects and arrays. For a single object, the bitmap is 1, and PRE is an odd number. Left end of the array stores the positive size, right hand stores the negative size. Indexes nxt (=next) and pre (=previous) implement the doubly-linked lists

Example: Let's assume that:

(a) Free object is a segment of memory which we can temporarily use for storing any information such as its size or pointers linking it to other free memory segments.
(b) We can keep a bitmap of our memory space, with one bit per object - see Fig. 2.14.[61]
(c) We allocate all objects so that their space can accommodate at least two integers—at least 8 bytes on 32-bit hardware.
(d) Unstructured memory as text strings will be allocated in chunks of 8 bytes

Figure 2.14 shows the representation of one array and one object. Note that even though `nxt` and `pre` form a doubly linked list, they are not pointers but integer indexes—thus they are persistent.

Typically, all single free objects are in one list, and free arrays in another. Another variation is to keep multiple lists, each for certain array size or range of sizes. There can also be a special class for non-structured data, with one object 8 bytes long. Any text, arrays of integers, etc. would be represented as objects or arrays of this class.

Figure 2.15 shows how this data organization detects abutting objects and allows one to combine them into a larger array. It also allows fast splitting of arrays or pulling a single objects from the free array, depending on your allocation strategy.

Until now, we assumed that we pull individual objects from an array of preallocated objects. But what are we going to do if we run out of preallocated objects? Allocating a bigger array and copying the original array in it would not work, because the existing objects would move and pointers to them would become invalid.

Instead, we can use a virtual array which is composed of pages, each a shorter array by itself, but all managed together and indexed as a single array—see

[61] Note that bitmaps used in previous chapters kept one bit for every 4-byte location. Here the bitmap is even smaller—how much smaller depends on the size of the object. Object must be large enough to store at least two values: SIZE and PRE, i.e. at least 8 bytes, but is usually much larger.

INITIAL SITUATION: pool=17, free objects=13, free arrays=4

base[ ]	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
size	x	1	x	4			-4	x	x	x	x	1	1	x	x	x
next	x	12	x	0				x	x	x	x	0	2	x	x	x
prev	x		x	0			0	x	x	x	x			x	x	x

bitmap

…….		1		1			1					1	1			

AFTER FREEING ARRAY 8-11, it combines with array 4-7 and single 12 and single 13, resulting in pool=17, free objects=2, free arrays=4

base[ ]	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
size	x	1	x	10									-10	x	x	x
next	x	0	x	0										x	x	x
prev	x	0	x										0	x	x	x

bitmap

…….		1		1								1	1			

AFTER FREEING ARRAY 14-16, it combines with free array 4-13, and thus reduces pool to 4, with free objects=2 and free arrays=0

base[ ]	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
size	x	1	x	-	-	-	-	-	-	-	-	-	-	-	-	-
next	x	0	x	-	-	-	-	-	-	-	-	-	-	-	-	-
prev	x	0	x	-	-	-	-	-	-	-	-	-	-	-	-	-

bitmap

…….		1		-								-				

Fig. 2.15 Allocation and free storage of objects: Array base[ ] for each class . x marks currently active objects. Index 0 is reserved for NULL pointer, and object at that location is never used. "pool" is the index of the first so far unused location; the bitmap is the same bitmap as in Fig. 2.14

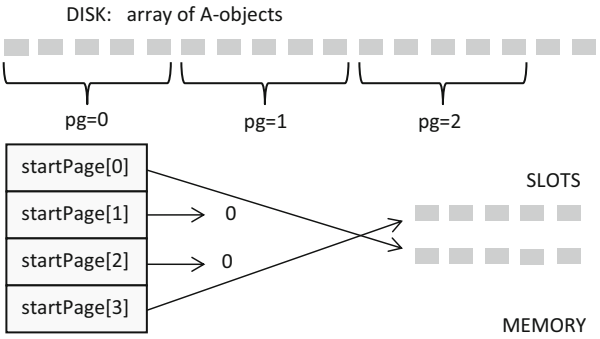

Fig. 2.16 Paging an array of objects from disk to memory. Pages move in and out of a few pre-assigned slots

Fig. 2.16. This is similar to what we discussed in Sect. 2.2.2, except that now we need such a paging system for every application class. That may appear complicated, but the overhead—both in the lines of code and in the required space—is practically the same as when we paged the entire memory.

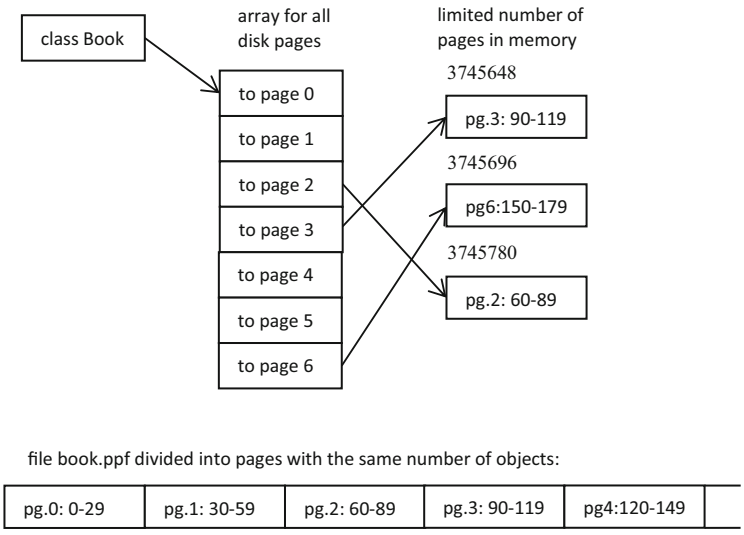

Fig. 2.17 Disk as the prime data storage. Pages store the same number of objects (here 30), and are paged to memory as needed

2.4.2 Array on Disk, Paged to Memory on Demand

An example of this approach is the Persistent Pointer Factory (PPF).[62] PPF assumes that the primary storage of the data is an array of objects on a disk file, with a separate file for each class. This array is paged to memory as needed—see Fig. 2.16. There is a limited number of slots to where pages can move. A page is not assigned any slot permanently. This removes the problem with increasing the size of the preallocated array without moving existing objects. We can expand the file gradually by adding pages without any limitation.

Smart generic pointer, `PersistPtr<T>`, stores the disk address of the target object, and its `operator ->` calculates the address each time it is invoked. For example, if we want to dereference the pointer which stores index 103 in Fig. 2.17, the program does the following calculation:

Page size is 30, so the object is on page 103/30 = 3, and at position 13 within this page (103 − 30*3 = 13). At this moment, this page starts at the address 3745648. Figure 2.18 does not say how large one object is; if it were, for example, 24 bytes, then the memory address at which the object resides at this moment is 3745648 + 13*24 = 3745860

[62] A commercial product distributed since 1997; the full source is now available on this book's website.

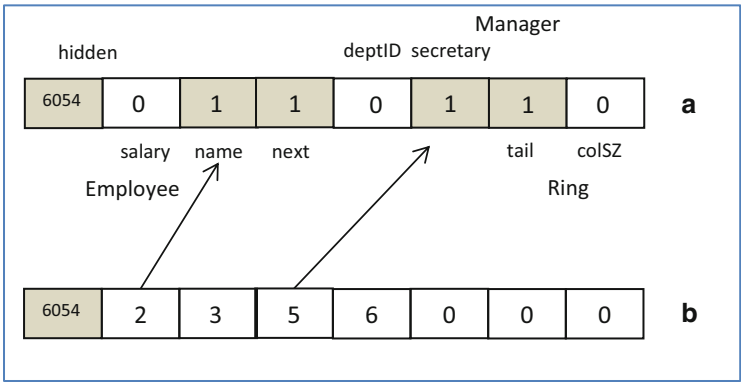

Fig. 2.18 Compared to Fig. 2.2, each class in Objective-C can keep just one mask (**a**). The hidden pointer inserted by the compiler is always in the first field, and pointers of any kind are marked by 1. For faster access, the same information can be converted to format (**b**), with hidden pointer in position 0 followed by the list of indexes for the pointer locations

An interesting question is whether it is better to store the disk address or the object index in the virtual disk array. Listing 2.16 shows the implementation of operator -> for both cases, and storing disk address results in a slightly faster access.[63] However, storing index makes it possible to handle larger data, potentially many times[64] the size of the virtual memory.

Listing 2.16 Disk as the prime storage—what to store in PersistPtr

```
// When storing disk address, the dereferencing is faster
template<class T> class PersistPtr {
     undisgned diskAddr; // disk address in bytes
     T* operator->() const{
         if(index)return pageArray[diskAddr/pageSz]+diskAddr%pageSz;
         return NULL;
     }
};

// When storing disk index, the disk size may exceed virtual memory
// but dereferencing includes additional multiplication
template<class T> class PersistPtr {
     undisgned diskIndex; // object index on disk
     T* operator->() const{
         if(index)return pageArray[diskInd/numOnPage]
                                 +(diskInd%numOnPage)*sizeof(T);
         return NULL;
     }
};
```

[63] PPF stores disk addresses.

[64] The factor increases with the size of the object size.

This is all simple and straightforward if the application classes do not use inheritance. The mask tells us the type of the target object for each pointer, in other words in which file to look for the object. The persistent pointer then stores the address or index in this file.

If the application class uses inheritance, the type recorded in the mask may have multiple meanings. For example:

```
class Shape {...};                      // class index=7
class Rectangle : public Shape {...}; // class index=8
class Circle : public Shape {...};     // class index=9
class Twin {                            // class index=4
    Shape *s1;
    Shape *s2;
};
```

If all these classes are persistent with the `PERSIST_CLASS()` statements, then the mask for the Twin class is [7,7]. However, pointers s1 and s2 can lead to a Rectangle or Circle, and these are stored in different files!!! When inheritance is involved, in addition to the disk address, the smart pointer must also store the true target type.

For this reason, PPF uses two types of pointers:

- `PersistPtr<T>` which stores only the disk address, and can be used only when class T is not using inheritance.[65] It has the same size as regular pointer.
- `PersistVPtr<T>` stores the target type and the disk address, and can be used under all circumstances. It is the only choice when class T is involved in inheritance. It takes the space of two pointers.

In PPF, user selects the page size and how many pages should be resident in memory for each specific class. Listing 2.17 shows the overall setup and the interface. Macro `PERSIST_CLASS(T)` overloads operator new() for each class. Allocating one object simply means adding 1 to the pool, and then asking the pager assigned to this class for the memory address. For allocating arrays, we need operator `new[]`, which does the same thing except that it derives the number of the objects from the given size.

Note that pool starts from 1, not from 0. Index 0 is the equivalent of a NULL pointer.

[65] Typically when no class is using inheritance.

Listing 2.17 Persistent pointers storing the object index

```
#define PERSIST_CLASS(T) \
friend PersistPtr<T>; \
void* operator new(size_t size){ \
    unsigned u=pool; pool++; 
    return pgr->getAddress(u);} \
void* operator new[](size_t size){ \
    unsigned u=pool; pool=pool+size/sizeof(T); 
    return pgr->getAddress(u);}\
static unsigned pool; \
static Pager *pgr

#define INIT(T,N) \
T* T::pool=1; \
Pager* T::pgr=NULL

// ------------------------------------------------------
class Library {
PERSIST_CLASS(Library);
  ...
}
INIT(Library,1);

class Book {
PERSIST_CLASS(Book);
  ...
}
INIT(Book,10000);
// ------------------------------------------------------
int main() {
    PersistPtr<Library> lib = new Library;
    PersistPtr<Book>      bk = new Book[250];
```

After a long journey we reached yet another variation of mapping a file to memory. The difference from the mapping discussed in Sect. 2.3 is that all objects there were saved in one file. Now each class has its own data file, but there is no fixed limit on the size of the data.

2.5 Quasi-Single Page (QSP)

When designing a new persistent system that would work with Objective-C, we combined the best ideas from all the existing approaches. The result is simple and efficient. It starts with one page of memory which expands to more pages if more space is required. During the run, discarded objects are reused, and when saving the data to disk, the memory space is all collapsed to a single page, with all unused space eliminated. This is the only completely automatic persistent system—pointer locations are retrieved through reflection, and calls to new() does not require any modifications.

In some chapters we tell you that they are not critical and that you can skip them or skim through them quickly. This chapter is just the opposite. It includes new ideas and algorithms, and we recommend you read it carefully and consider every detail.

Objective-C is the language used by Apple and is thus important for iPhone applications. Under the label of "archiving" it has serialization which requires more manual input than most other languages, and its main flaw—just like in Java and C# serializations—is the recursive internal algorithm which walks through all active objects. If the data includes a long chain of pointers, the system stack overflows easily and the program crashes. The workaround is to maintain a collection of all active objects and to write them individually to disk, which complicates the matters and is a potential source of errors.

The purpose of this chapter is to develop a new approach to automatic persistence which would be simpler to implement yet be as fast, if not faster, than the existing persistent systems. It should run in Objective-C, but the idea should be applicable to C++, C# and possibly other languages.

The method is new, and we believe that it will be highly competitive in performance, flexibility[66] of use, and simplicity of internal design—not only in Objective-C, but also in C++ and other languages.

For the benefit of the majority of readers, this chapter uses code samples mostly in C++. Final implementation in Chap. 6 is written entirely in Objective-C.

This design is a good example of how persistence, data structures, and allocation—when not treated as orthogonal, can be most efficient while working jointly toward the same *objective*.

BUILDING BLOCK 1: Pointer Mask For each class, we can generate the mask showing pointer locations, either by using the approach described in Fig. 2.2 and Listing 2.3, or through reflection as will be explained in Chaps. 5 and 6.

However, this mask will be simpler than that shown in Fig. 2.2. Objective-C is a dynamically typed language and, once we have a pointer, we can determine the type of the target object. We do not need to give ID to each class; we simply mark the pointer locations by 1—see Fig. 2.18. Objective-C uses only one hidden pointer at the beginning of the object. We do not have to record it in the mask but, for convenience, we can keep its value there.

BUILDING BLOCK 2: Making Classes Persistent We will use the simple interface at which we arrived in Sect. 2.1.2, with two statements added to each class[67]:

`PersistInterface;` will insert additional methods[68] needed for the persistence.

`PersistImplementation;` will insert static members and the implementation of the added functions.

[66] It supports both storing entire pages and serialization, including the existing Objective-C format.

[67] The names are slightly changed to fit Objective-C terminology.

[68] Mostly static, "messages" in Objective-C lingo.

These two statements will be macros, but clean macros, simply a section of code that repeats for every class. All pointer members will be registered as a `PTR()` statement which, in Sect. 2.2, was in the default constructor:

```
class Library {
PersistInterface;
     Book *books;
     Library *next;
     int telephone;
     char *libName;
public{
     Library(){
        PTR(books,Book); PTR(next,Library); PTR(libName);
        ... anything else you want here
     }
};
PersistImplementation;
```

Some things are simpler in Objective-C. For example, `PersistInterface` and `PersistImplementation` do not need the class parameter.[69] Also, we can use `PTR()` for any kind of pointer, while in C++ we were using `PTR()` for object pointers and `STR()` for strings.

Other things are more complicated in Objective-C. There is no equivalent to the C++ default[70] constructor, which automatically invokes default constructors of its superior (base) classes. In Objective-C, any method can serve as a constructor, but the traversal of the superior classes must be introduced explicitly.[71]

BUILDING BLOCK 3: Replacing Allocation The simplest and most efficient memory management for persistent data is one block (or page) of virtual memory, from which we allocate all the objects. Storing data to disk is reduced to dumping the entire page to the disk, and loading the data back to memory is also very fast. Pointer conversion (swizzling) requires only to add the same offset to all pointers on the page.

We described this method in Sect. 1.4.1 (Old Smalltalk Model), and we pointed out its main weakness—if the data grows beyond the page size, allocation of a larger page and copying of the old page into it would be as complicated operation as storing data to disk. If we move any objects, we have to swizzle all the pointers!

In our new algorithm we will replace one page by a set of pages, but will control these pages and their sizes in such a way that all the data will be on a single page most of the time. We will also collapse multiple pages into a single page any time we'll be saving the data to disk.

Figure 2.19 shows the architecture. Typically, there is only one or a few pages; multiple pages are only a temporary measure for additional data. All pages have the

[69] The equivalent from Sect. 2.2 would be `PersistInterface(Library).`

[70] Constructor without parameters, e.g. `Library(){...}`.

[71] We will show in Chap. 7 how you do that.

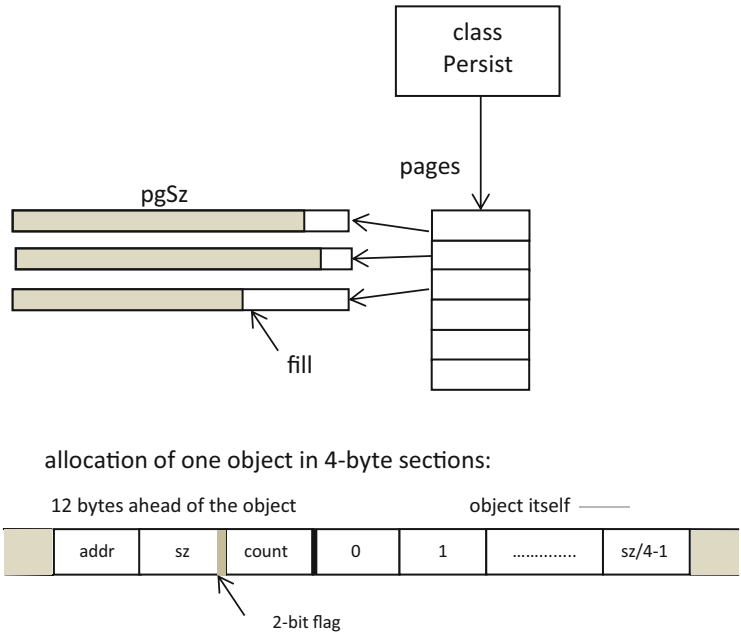

Fig. 2.19 QSP allocation usually involves one or a few pages, which are converted to one page while storing to disk. The lowest two bits of the `sz` field are used as a type flag: 0 = object or array with no pointers, 1 = object/array of objects with a hidden pointer, 2 = array of pointers. Field `addr` is a temporary space for internal algorithms

same size `pgSz`, but different `fill`. Before saving to disk, pages are sorted by their starting address and converted to a single page.

Before each object or array, the allocation inserts a 12-byte header,[72] which includes the retain count that Objective-C expects to be there. Field `addr` is a temporary variable that QSP needs in some algorithms. For example, when traversing all active objects, it takes the role of the "next" pointer which builds the queue. Later on, when merging all pages into one, it stores the new address of the object before it actually moves there.

Instances of classes derived from NSObject start with a hidden pointer at the beginning of the actual object. The overhead is 12 bytes per object,[73] which is less than 16 bytes in the standard Objective-C heap.

[72] Note that this is still less than the 16-byte header Objective-C uses when allocating from the heap.

[73] Temporarily, while tArr is used, the overhead is 20 bytes.

Field (sz) stores the size of the object, except for its two lowest bits which are used as a special flag. Sizes of objects are always multiples of 4, so the two lowest bits in this field are unused, and we can use them for this flag with values between 0 and 3.

We will allocate all objects—both instances of application classes and large, irregular objects such as blocks of text or pictures from the same data space.

Using one large page instead of smaller multiple pages has a potential flaw. Quoting Mark Bales: *One of our systems worked this way but as designs grew, it required a *very* large block of memory. When we tried to re-load the data back from disk, the read operation failed. This was because there wasn't enough space between various smaller blocks still in use. As a result, I have become convinced that page-based techniques should remain page-based even on re-read.*

Note that this problem can occur for very large data (VLSI design in Mark's case) and for programs that re-read the data within the same run. A simple cure is to provide a smart-read function, which in cases of read failure breaks the data into multiple pages. That by itself is simple but, for multiple pages, pointer swizzling becomes more complicated and time consuming. Instead of applying the same offset to all pointers, we must first find the proper page, and then apply the offset.

Representation of Arrays

The representation of arrays is critical and deserves more explanation. We have three possible styles of arrays. In all three cases, sz stores the overall size of the array in bytes.

(a) Arrays that do not include any pointers are stored and represented in the same way as a block of text, with flg=0.
(b) Arrays of objects that are instances of persistent classes, each object starting with a hidden pointer, are stored with flg=1. A single object derived from NSObject is a special case—an array of objects with only one object.
(c) Arrays of pointers are special, because they do not need any mask. We know that every 4 bytes represent a pointer, and their signature is flg=2.

Listing 2.18 shows the differences in how we use or allocate these different arrays, and what we do with them when we either traverse all objects or swizzle the pointers.

Listing 2.18 Using various types of arrays

```
class Chapter;
class Author;
class Store;
class Book {
     PersistInterface(Book); // registration of the class
     char *name;
     int ISBN;
     Book *next;
     Chapter **chapters; // array of pointers to Chapters
     Author *authors;    // array of Author objects
     Book(){
          PTR(next); PTR(chapters); PTR(authors); PTR(stores);
          ... anything else you want here
     }
};

// Different ways of allocation store the object differently.
// Persist is the persistent utility
bk->name=Persist.palloc(sz); // text or other no-pointer data
bk->next=new Book;  // overloaded new() for a single object
bk->chapters=Persist.allocPtrArr(sz); // pointer array
bk->authors=Book.allocArr(sz); // method automatically added to B
```

SPECIAL RULE FOR ARRAYS OF OBJECTS:

If you stop using any elements of an array, for example when reducing the number of elements, you must set all the pointers in the released elements to NULL or, safer and easier, simply overwrite these elements with 0s.

The program that controls the persistency has no information about how big a part of the array is actually used. If you don't follow the Special Rule, the program may crash when swizzling or traversing pointers. We recommend that raw arrays of objects such as shown in Listing 2.16 are not used, but instead that they are encapsulated in a special Array class that takes care of overwriting discarded objects with 0.

There are two situations when the algorithms have to traverse pointers of all objects: when we traverse the pointer graph starting from the root in order to find all the active (connected) objects, and then when we are swizzling[74] the pointers of these objects. Assume that we have object `obj`, and that we want to report all pointers that lead to other objects. The following code which finds all pointers `ptr`

[74] Resetting pointer values after all the objects move to a different place in memory.

in `obj` and calls `fun(ptr)` has interesting logic,[75] which works particularly well in Objective-C[76]:

```
int i,k,mySz; char *obj,**ptr,**msk;
int ptrSz=sizeof(char*);
int flg=getFlg(obj); // get the flag from the allocation record
if(flg==0)return; // there are no pointers in obj
int sz=getAllocatedSize(obj); // get the allocated size of obj
if(flg==1){
    mySz=getClassSize(obj); else mySz=0;
    char *mask=getMask(obj);
}
for(i=k=0; i<sz; i=i+ptrSz, k=k+ptrSz){ // i for obj, k for mask
    ptr=(char**)(obj+i); // *ptr is the value at location i
    if(flg==2){fun(*ptr); continue;} // every location is a pointer
    // k is used only in the next part
    if(k>=mySz)k=0; // repeat the mask for the next section
    if(k==0)continue; // hidden pointer, we exclude them
    msk=(char**)(mask+k);
    if(*msk)fun(*ptr); // if mask is not 0
}
```

BUILDING BLOCK 4: Algorithms A and B The following two algorithms are the heart of this entire approach.

Algorithm A traverses all active objects without recursion. It uses field `addr` as the "next" pointer when building two stacks: One for the objects still to be expanded, the second for those already expanded. A simple check whether `addr==0` prevents the same object expanding again.

Algorithm A can be used for two purposes:

(a) To serialize the data in any of the existing Objective-C formats, simply calling function `writeSingleObject(void* ptr)` which saves the object without expansion.

(b) To set up the data for Algorithm B which eliminates dead space and collapses the data space into a single page. Condition `addr==0` marks an object as a dead space.

For all application classes, the standard allocation is replaced[77] by allocation from our special pages, regardless whether we store the data with QSP or serialization. Being able to alternate between the two styles of saving the data has many advantages.

[75] This is more an algorithm description than a functional code.

[76] You will see this in Chap. 7.

[77] The new alloc() method is hidden under `PersistInterface`.

ALGORITHM A: Serialization
(mark all active objects with ind=1, possibly write them to disk)

```
Stage:
ptrSz is the size of pointer, in bytes
The algorithm assumes addr=0 for all objects. Allocator sets
it that way, but if this function is called several times,
this needs a special management.

Algorithm:
  void *stackBeg, *stackEnd; // stack, objects to be expanded
  void *listBeg; // list of active objects
  void *nxtObj; // next object to expand
  void **targPtr; // leads to the next object
  char *p;
  sort pages; // Algorithm B will rely on it
  { place root in the stack, set addr=1}
  stackBeg=stackEnd=root;
  while(stack not empty){
      nxtObj=stackBeg;
      if(option)writeSingleObject(nextObj); // serialization
      int flg=getFlg(nxtObj);
      if(flg>0){ // skip for 0: no pointers, no expansion
          if(flg==1){ // object with hidden pointer
             {get pointer mask, 1 marks regular pointers}
          }
          for(int i=0; i<sz; i=i+ptrSz){ // all ptr locations
              p=(char*)nxtObj+i;
              if(flg==1 && mask[i]!=1)continue;
              targPtr=(char**)p;
              if(*targPtr)==NULL)continue;
              if(addr for *targPtr > 0)continue;
              {add *targPtr to the end of the stack}
          }
      }
      { move nxtObj from stack to list }
      activeSpace=activeSpace+12+sz; // sz for nxtObj
  }
  returns:
      listBeg; // list of active objects
      activeSpace; // space needed for all active objects
  NOTE: Stack and List use addr=1 instead of NULL at the end
```

The Purpose of Algorithm B Is:

- To concatenate all the pages into one.
- To remove discarded or lost objects.

If, after running Algorithm A, we have a single page and the total space is equal to the active space, there is no need to call Algorithm B (Fig. 2.20).

It also does not make any sense to do a big cleanup if there are just a few unused objects. Algorithm B has a flexible, user defined cutoff—for example, it can be

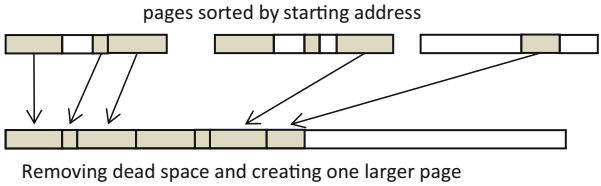

Fig. 2.20 Pass 3 of Algorithm B

bypassed if there is only one page and the dead space is not more than 10 % of the allocated space (cutoff specified as 0.1 for 10 %).

Algorithm B allocates a new, single page to store all the data without the dead space. Then it traverses the list of active objects that Algorithm A left behind, and calculates the future address of each object in this new page. Because the objects are already sorted, this calculation involves only a gradual accumulation of the memory shift for the dead space and gaps between pages. In the second pass, the algorithm traverses the active objects again, and swizzles their pointers to the values stored in `addr` of the target object. There are no searches or dictionaries. In the third pass all the active objects are copied into the new page.

The second and third pass could be combined into a single pass, but we would not gain much. Leaving them separate allows one to copy the data into an existing page without allocating a new one, assuming that the old page is large enough to receive the data.[78]

Algorithm C stores the single page to disk, and it is trivial. It first writes the header with the overall parameters:

`fill` = total space required to receive the data,

`root` = old address of the root,

`pageAddress` = address on which the old page started and then the table of registered classes, each entry

```
class name
size of each instance
pointer mask (includes the value of the hidden pointer)
```

The hidden pointer is needed for the conversion when loading the data back to memory, and the mask is for a rough check that the class has not changed.

The page size is not passed; the receiving program can chose any size which is not smaller than `fill`.

Finally, there is a binary dump of the entire data space[79] and of the bitmap.

Algorithm D allocates a large-enough page and bitmap, and fills both with the data from the disk. Without looking at individual objects, it runs through the bitmap looking for pointer locations, and swizzles them including the root pointer by the same offset `(startOfNewPage - startOfOldPage)`. When the original pointer does not fit the old page, it must be a hidden pointer, and it is replaced using the old-to-new conversion table of hidden pointers. Note that if there are any dead objects,

[78] This option is not in the first version of this code.

[79] As a single page.

converting their pointers does no harm, and because there should not be many of them, it may be faster to traverse bitmap than to analyze individual objects.

The conversion allows for some changes of the schema:

(a) If the new data contain additional classes, it does not matter. We can still read the old data.

(b) If some old classes are missing in the new set, but there are no instances of these classes in the old set, that is also fine, but this may require an additional check during the swizzling of pointers.

ALGORITHM B: Cleanup and compression
(eliminates discarded objects and compresses all data into one page).

```
Stage:
Algorithm A was performed, addr!=0 marks active objects.

Algorithm:
    void *nxtObj,*root,*newAddr; int n;
    Using the cutoff condition, decide whether to bypass.
    Sort the array of pages by the increasing page address.
    Allocate a new large page, page0.
    Disassemble the list of active objects.
    // Pass 1: calculate future addresses
    newAddr=page0 + 12;
    Using sz and headerSz, traverse objects page by page {
       if(nxtObj->addr==0)continue; // not active object
       nxtObj.copy(newAddr); // including 12 bytee header
       nxtObj->addr=newAddr;
       newAddr=newAddr+nxtObj->sz+12;
    } // also collect all active objects in char** active[].
    //Pass 2: Swizzle page0 pointers using addr from orig. data
    for(i=0; active[i]; i++){
        nxtObj=active[i];
       newAddr=nxtObj->addr; // swizzle pointers in page0
       {based on the 2-bit flag, decide how to access pointers}
       {get mask from the class, when appropriete}
       traverse pointers of newAddr {
          if(the pointer is neither NULL nor hidden pointer){
              replace pointer value by addr of the target object
          }
       }
    }
    convert the root reference to addr stored on the root object
    // Pass 3: Concatenate active objects (see Fig.2.17)
    newAddr=page0;
    for(i=0; active[i]; i++){
        nxtObj=active[i];
        n=nxtObj->sz + 12;
        memcpy(newAddr,nxtObj-12,n);
        newAddr=newAddr+n;
    }
    delete all old pages;
    set page0 as page[0];
    newAddr is its 'fill'
```

BUILDING BLOCK 5: Special Arrays So far we have been working with single instances of application classes. What are we going to do with arrays? Arrays of characters or of other basic types such as *int* or *long long*, or even arrays of structures (*struct*) as long as they do not include any pointers are no problem. We allocated each array as an object of the appropriate size, and other objects can refer to it with a pointer. The bitmap corresponding to such an array will be full of 0s, so we do not have to do anything special for these arrays except that we have to make sure they get allocated with the allocator from our Utility class.

Objective-C does not allow arrays of instances, only arrays of pointers. For example:

```
class A {
     int id;
     A *next;
}
int main(){
     A *arr=new A[100]; // array of A instances, not in Objective C
     A **pArr=new (A*)[100]; // array of pointers to A instances
     for(int i=0; i<100; i++){
          pArr[i]=new A;
     {
     printf("arr=%d pArr%d\n",arr[17].id,pArr[35]->id);
```

Instead of calls such as

```
A **pArr=new (A*)[100];
```

the application code will have to call a special function which will not only allocate the appropriate memory from our pages but also fill its assigned area of bitmap with 1s.

BUILDING BLOCK 6: Managing Free Objects Any saving of data to disk, even if we don't exit our program after that, will remove all unused objects. We can even call Algorithm B without saving to disk, and the unused objects will be removed and all the data will be compacted to a single page. That, however, may cause a pause in the execution of the program, which in some applications may not be acceptable. Also, some applications continuously destroy objects and create new ones, and being able to reuse destroyed objects would help significantly.

It would[80] be easy to arrange chains of free objects organized by size—for sizes corresponding exactly to the application classes, and approximate sizes of all powers of 2—see Fig. 2.21. Any field in the allocated space can be used for the temporary pointer that creates the chain; the beginning of the object seems most appropriate.

In real-life applications, large blocks of memory that are freed and reused are usually in a relatively small range of sizes. Remembering this range leads to the following performance optimization.

[80] This feature may not be in the first release of this Objective-C persistence.

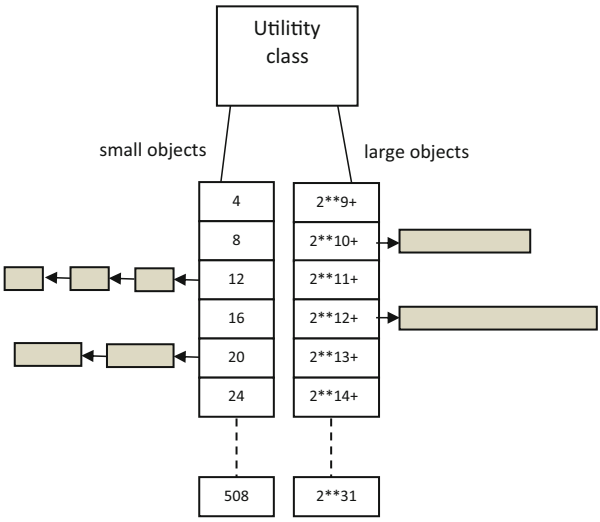

Fig. 2.21 Chains of free objects organized by size. The picture assumes instances of all application classes are not more than 24 bytes in size, and all free objects up to that size can be picked up with appropriate size. Larger objects which can only be arrays or long strings, are organized by power of 2

Useful Trick No. 6
When moving larger blocks to free storage or reusing them (see Fig. 2.21) we need to find the right slot for the given size. The binary search seems to be the best choice here, but remembering the range of the stored values makes it super-fast.

Example

slot	value
0	512
1	1024
2	2048
3	4096
4	8192
5	16384
....	
22	214748364

minRange
maxRange

We start the run with minRange>maxRange. This is a signal of empty free storage.

When block of size 1048 enters free storage, minRange=1024, maxRange=2044.

When allocating block of size 5000, we do search for a free object, because it is out of range.

When entering the 5000 block to free storage, we search in range 1024-214748364, then adjust maxRange to 8188.

When taking the 1048 block out of the free storage, the range remains 1048-8188; removal from free storage does not change the range.

BUILDING BLOCK 7: Persistent Libraries In general, persistent data is only as useful as the data structures it supports. This is truer for Objective-C than for other languages, because the NextStep (NS) library is essentially part of the core language. You cannot program in Objective-C without it.

Making a library class persistent is just as easy as making an application class persistent:

STEP 1: Modify *.h by *adding PersistInterface*.

STEP 2: Modify *.m by adding *PersistImplementation* and method *prtList* which registers all the pointers using the PTR() statement.

STEP 3: Search all methods in *.m for any calls that allocate arrays or unstructured memory, and replace them by *allocArr, allocPtrArr*, or *palloc*. (Allocations of single objects with new() require no changes.)

Section 6.2 will describe implementation details specific to Objective-C, NS classes, and to the conversion of the InCode library to Objective-C.

Note that a program may run with persistent application classes, while using NS classes that are not persistent and are allocated from the Objective-C heap. The serialization algorithm (Algorithm A) will save both parts seamlessly. When reading such data from disk, the application classes will automatically allocate from persistent pages of memory, while the NS classes will allocate from the Objective-C heap.

BUILDING BLOCK 8: The Main() The main() program has to start the persistent utility and all the persistent classes.

```
int main() {
    PersistStart(pageSize);
    myClass1.start();
    myClass2.start();
    ... etc. for all application classes
```

where you normally use pageSize=0 which uses a good default. Selecting this parameter may slightly improve the speed of saving the data to disk especially for very large data sets, but has no impact on the speed of traversing the data. As you save and open the data, the internal algorithms convert all the data to one page anyway, and then the original choice of the page size is irrelevant.