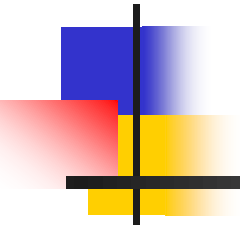
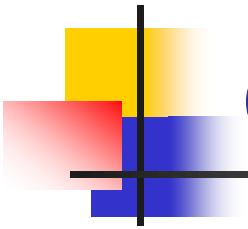
Chapter 5. Managing Files of Records



Ki-Joon Han
School of Computer Science & Engineering
Konkuk University
kjhan@db.konkuk.ac.kr



Chapter Outline

5.1 Record Access

5.2 More about Record Structures

5.3 Encapsulating Record I/O Operations in a Single Class

5.4 File Access and File Organization

5.5 Beyond Record Structures

5.6 Portability and Standardization

5.1 Record Access
5.1.1 Record Keys (1/2)

- Key
  - to identify an individual record based on the contents
  - standard form (*canonical form*) with associated *rules* and *procedures* for converting keys into the standard form (e.g., AMES, ames, or Ames)

- Canonical form of key
  - single representation for the key that conforms to the rule
  - (e.g.) key consists solely of uppercase letters and has no extra blanks at the end (Ames => AMES)

5.1.1 Record Keys (2/2)

- Primary key
  - key used to identify a record uniquely
  - not contain a real data value (dataless) => unchanging
- Secondary key
  - key that does not uniquely identify a record

- Good rule

  (1) avoid putting data into primary keys (e.g., 부서명)

  (2) assign data content to secondary keys

5.1.2 A Sequential Search (1/4)

- Sequential search
  - read through the file, record by record, looking for a record with a particular key
- Performance measure

1. Memory
   - number of comparison required for the search
   - comparison operations are more expensive than fetch operations to bring data in from memory
2. DISK
   - number of disk I/O (= low-level Read call)
   - assumption
     - each Read call requires a seek (multiuser environment) and is equal-cost

5.1.2 A Sequential Search (2/4)

- Work for sequential search (*n* records)

  maximum : *n* comparison (Read calls)
  minimum : *1* comparison (Read call)
  average : *n/2* comparison (Read calls)
  } *O(n)*

- Performance improvement of sequential search
  - by reading in a *block* (2 to 64 kilobytes) of several records all at once and then processing that block of records in memory
- Grouping
  - a stream of bytes --> fields --> records --> blocks

<-for physical organization-> <-for logical organization-><-for performance

5.1.2 A Sequential Search (3/4)

- Block size

1. sector-oriented disk drive
   - related to the physical properties of the disk drive
   - always some multiple of the sector size
2. block-oriented disk drive
   - related to the content of the data
   - some multiple of the record size

- (ex) 4,000 512-byte records with 512-byte sector

(1) unblocked sequential search
- average 2,000 Read calls

(2) blocked (bf=16) sequential search
- average 125 Read calls

slightly more time*

5.1.2 A Sequential Search (4/4)

- Adv. of sequential search
  - extremely easy to program
  - require the simplest file structures
- Disadv. of sequential search
  - too expensive for most serious retrieval situations

- Many situations for sequential search

1. ASCII files in which you are searching for some pattern (e.g., *grep*)
2. files with few records (e.g., 10 records)

⋮

=> Identify better ways to access individual record

5.1.3 UNIX Tools for Sequential Processing

- Most common file structure in UNIX
  - an ASCII file with the new-line character as the record delimiter and, when possible, white space (e.g., blanks or tabs) as the field delimiter
- UNIX tools (sequential search)
  - variable-length record

    field : ending with a tab

    record : ending with a new-line

  (1) cat (concatenation) : *cat myfile*

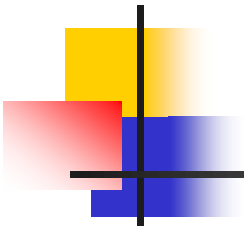
  (2) wc (word count) : *wc myfile*

  (3) grep (*egrep* and *fgrep*) (generalized regular expression)
    - *grep Ada myfile*
    - *grep Ada myfile | wc*

5.1.4 Direct Access (1/4)

- Direct access
  - seek directly to the beginning of the record and read it in
  - O(1) (v.s. sequential search : O(n))
  - index file : information about record locations

```
int IOBuffer::DRead (istream & stream, int recref)
// read specified record
{
        stream.seekg (recref, ios::beg);
        if (stream.tellg() != recref) return -1;
        return Read (stream);
}
```



5.1.4 Direct Access (2/4)

- Relative record number (RRN)
  - record position relative to the beginning of the file
  - view a file as a collection of records
  - RRN0, RRN1, …

5.1.4 Direct Access (3/4)

- Direct access by RRN

1. variable-length record : O(n)
   - have to read sequentially through the file, counting records as we go, to get to the desired record
2. fixed-length record : O(1)
   - calculate the *byte offset* of the desired record

- Byte offset
  - record position (byte count) relative to the start of the file
  - byte offset = n * r { n : RRN, r : record size
  - (ex) RRN : 546, fixed-length record size : 128 bytes
    => byte offset = 546 * 128 = 69,888

5.1.4 Direct Access (4/4)

- Two kinds of view

1. low-level view of files (C++)
   - file : a sequence of bytes
   - application program does the byte offset calculation and uses the *seekp* and *seekg* methods to jump to the desired record
2. high-level view of files (Cobol)
   - file : a collection of records that are accessed by keys
   - operating system takes care of the translation between a key and a record's location (not programmer's concern)

5.2 More about Record Structures
5.2.1 Choosing a Record Structure and Record Length (1/4)

- Record length for a fixed-length record

1. lengths of the fields are fixed
   - sum of the fixed lengths
   - 512-byte sector : 30 bytes (spanning) --> 32 bytes (no spanning)
2. lengths of the fields can vary
   - sum of the estimates of the largest possible values for all the fields

     => waste a lot of space

5.2.1 Choosing a Record Structure and Record Length (2/4)

- Fields organization within a fixed-length record

1. fixed-length fields : Figure 5.1(a)
   - simplicity
2. variable-length fields : Figure 5.1(b)
   - averaging-out effect
   - efficient use of a fixed amount of space
3. fixed-length and variable-length fields

Ames	John	123	Maple	Stillwater	OK74075
Mason	Alan	90	Eastgate	Ada	OK74820

(a)

Ames | John | 123 Maple | Stillwater | OK | 74075 | ← Unused space →

Mason | Alan | 90 Eastgate | Ada | OK | 74820 | ← Unused space →

(b)

5.2.1 Choosing a Record Structure and Record Length (3/4)

- Distinguishing methods (real data portion ~ unused space portion)

1. record-length count at the beginning of the record
2. special delimiter at the end of the record

(ex) (1) Figure 5.2(a)

- fill out the unused portion of the record with null characters("00": null)

(2) Figure 5.2(b)

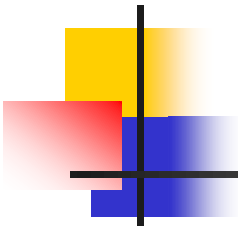
- place a fixed-length field (an integer) for a length count at the start of the record("20": space)

=> Header record : *header size, record count, record size*

5.2.1 Choosing a Record Structure and Record Length (4/4)–hexadecimal

```
0000000  0020  0002  0040  0000  0000  0000  0000  0000  ................  Header: header size (32),
0000020  0000  0000  0000  0000  0000  0000  0000  0000  ................  record count (2), record size (64)
0000040  416d  6573  7c4d  6172  797c  3132  3320  4d61  Ames|Mary|123 Ma  First record
0000060  706c  657c  5374  696c  6c77  6174  6572  7c4f  ple|Stillwater|O
0000100  4b7c  3734  3037  357c  0000  0000  0000  0000  K|74075|........
0000120  0000  0000  0000  0000  0000  0000  0000  0000  ................
0000140  4d61  736f  6e7c  416c  16e   7c39  3020  4561  Mason|Alan|90 Ea  Second record
0000160  7374  6761  7465  7c41  6461  7c4f  4b7c  3734  stgate|Ada|OK|74
0000200  3832  307c  0000  0000  0000  0000  0000  0000  820|............
0000220  0000  0000  0000  0000  0000  0000  0000  0000  ................
                                                  (a)
0000000  0042  0002  0044  0000  0000  0000  0000  0000  ................  Header: header size (66)
0000020  0000  0000  0000  0000  0000  0000  0000  0000  ................  record count (2), record size (68)
0000040  0000  0000  0000  0000  0000  0000  0000  0000  ................
0000060  0000  0000  0000  0000  0000  0000  0000  0000  ................
0000100  0000
0000102        0028  416d  6573  7c4d  6172  797c  3132    (.Ames|Mary|12  First record
0000120  3320  4d61  706c  657c  5374  696c  6c77  6174  3 Maple|Stillwat  Integer in first
0000140  6572  7c4f  4b7c  3734  3037  357c  0020  2020  er|OK|74075|      two bytes contains
0000160  2020  2020  2020  2020  2020  2020  2020  2020                    the number of
0000200  2020  2020                                                        bytes of data in the record
0000204              0024  4d61  736f  6e7c  416c  616e  $.Mason|Alan      Second record
0000220  7c39  3020  4561  7374  6761  7465  7c41  6461  |90 Eastgate|Ada
0000240  7c4f  4b7c  3734  3832  307c  0020  2020  2020  |OK|74820|
0000260  2020  2020  2020  2020  2020  2020  2020  2020
0000300  2020  2020  2020
                                                  (b)
```

Figure 5.2 Two different record structures that carry variable-length fields in a fixed-length record. (a) File containing a 32- (20_{16}) byte header and two fixed-length records (64 bytes each) containing variable-length fields that are terminated by a null character. (b) File containing a 66- (42_{16}) byte header and fixed-length records (68 bytes each) beginning with a fixed-length (2-byte) field that indicates the number of usable bytes in the record's variable-length fields.



5.2.2 Header Records

- Header record
  - placed at the beginning of the file to hold some general information about a file
  - the number of records, the length of the data records, the date and time of the file's most recent update, the name of the file, etc.
  - make a file a *self-describing object*
  - if a tree-structured index is used, it is often placed at the beginning of the index

5.2.3 Adding Headers to C++ Buffer Classes

- Header processing in the IOBuffer class hierarchy
  - see Appendix F
  - virtual int ReadHeader ( );
    - read the header and check for consistency
  - virtual int WriteHeader ( );
    - write a header to a file and return the number of bytes in the header
- Header support in the BufferFile class
  - BufferFile::Create
  - BufferFile::Open
  - BufferFile::ReadHeader
  - BufferFile::WriteHeader
  - BufferFile::Rewind

5.2.3 Adding Headers to C++ Buffer Classes

```
Class BufferFile
{public:
 BufferFile (IOBuffer &);                    //create with a buffer
 int Open (char * filename, int MODE);  //open an existing file
 int Create(char * filename, ..);            //create a new file
 int Close();
 int Rewind();                    // reset to the first record
 int Read (int recaddr = -1);   // read a record into a buffer
 int Write(int recaddr = -1);   // write the buffer contents
 int Append(); // write the current buffer at the end of file
protected:
 IOBuffer & Buffer; // reference to the file's buffer
 fstream File;          // the C++ stream of the file
 int HeaderSize;       // size of header
 int ReadHeader();
 int WriteHeader();
};
```

5.3 Encapsulating Record I/O Operations in a Single Class (1/5)

- Issues in class RecordFile

```
//for class Person
Person p;
RecordFile pFile;                    => RecordFile <Person> pFile;
pFile.Read(p);

//for class Recording
Recording r;
RecordFile rFile;                    => RecordFile <Recording> rFile;
rFile.Read(r);
```

=> **support files for different object types**

(1) Virtual function call : No
   - Do not have a common base type

(2) C++ template : Yes
   - Support parameterized function and class definitions

5.3 Encapsulating Record I/O Operations in a Single Class (2/5)

- RecordFile template class (Figure 5.3)
  - support a read operation that reads an *object* of some class and writes it to a file
  - the use of buffers is hidden inside the class (i.e., *pack into a buffer and write the buffer to a file*)

```
//The template parameter RecType must support the following
//    int Pack (BufferType &); pack record into buffer
//    int Unpack (BufferType &); unpack record from buffer
template <class RecType>
class RecordFile: public BufferFile
{public:
    int Read (RecType & record, int recaddr = -1); //file-> RecType
    int Write (const RecType & record, int recaddr = -1); //RecType->file
    int Append (const RecType & record, int recaddr = -1);
    RecordFile (IOBuffer & buffer): BufferFile (buffer) {}
};
```

5.3 Encapsulating Record I/O Operations in a Single Class (3/5)

```
//template method bodies
template <class RecType>
int RecordFile<RecType>::Read (RecType & record, int recaddr = -1)
{
    int writeAddr, result;
    writeAddr = BufferFile::Read(recaddr);
    if (!writeAddr) return –1;
    result = record.Unpack (Buffer); //RecType::Unpack
    if (!result) return –1;
    return writeAddr:
}
template <class RecType>
int RecordFile<RecType>::Write (const RecType & record, int recaddr=-1)
{
    int result;
    result = record.Pack(Buffer); //RecType::Pack
    if (!result) return –1;
    return BufferFile::Write (recaddr);
}
```

5.3 Encapsulating Record I/O Operations in a Single Class (4/5)

- Issues in class RecordFile

```
//for class Person
Person p;
RecordFile <Person> pFile;
pFile.Read(p);

//for class Recording
Recording r;
RecordFile <Recording> rFile;
rFile.Read(r);
```

=> pack, unpack methods of Person, Recording are used for pFile and rFile, respectively

- **pFile.Read(p) calls Person::Unpack**
- **rFile.Read(r) calls Recording::Unpack()**

5.3 Encapsulating Record I/O Operations in a Single Class (5/5)

PersonFile:

RecordFile<Person> PersonFile (Buffer);

- Object PersonFile is a RecordFile that operates on Person objects
- All of the operations RecordFile<Person> are available, including those from the parent class BufferFile

```
Person person;
PersonFile.Create ("person.dat", ios::in); // create a file
PersonFile.Open ("person.dat", ios::in);  // open and check header
PersonFile.Read (person);                 // read a record into person
PersonFile.Append (person);               // write person at the end of file
```

- testfile.cpp in Appendix F
  - use RecordFile to test all of the buffer I/O classes

5.4 File Access and File Organization

- File organization
  1. file-handling facilities of the language
  2. the use you want to make of the file
- File access
  1. sequential access
     - fixed-length or variable-length records : O.K.
  2. direct access
     (1) fixed-length records
     - by using RRN

     (2) variable-length records
     - by keeping a list of the byte offsets for each record

     ① C++ : allow access to any byte in the file

     ② Pascal : see a file as a collection of records that are the same type => difficult

5.5 Beyond Record Structures 5.5.1 Abstract Data Models for File Access (1/2)

- Sound, image, and document data
  - envision such a data as objects we manipulate in ways that are specific to the *objects* themselves (rather than as *fields* and *records* on a disk)

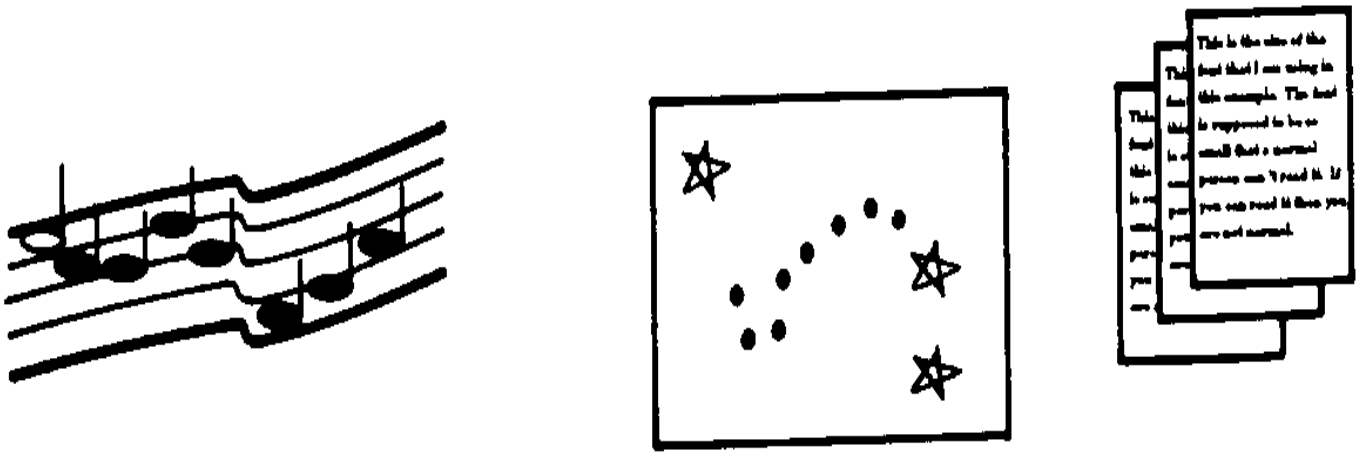


Figure 5.4 Data such as sound, images, and documents do not fit the traditional metaphor of data stored as sequences of records that are divided into fields.

5.5.1 Abstract Data Models for File Access (2/2)

- Abstract data model
  - an in-memory, *application-oriented view* of data (as a object) rather than a medium-oriented one (as fields and records on a disk)
  - describe the organization and access methods in terms of how an application views the data rather than how the data might physically be stored
  - put distance between the physical and the logical organization of files (i.e., ignore the physical format of objects in files)
  - focus more on the information content of files and less on physical format

=> *put file structure information in a header*

5.5.2 Headers and Self-Describing Files

- Header (record)
  - the number of records in the file
  - a name for each field
  - the width of each field
  - the number of fields per record

  } *self-describing file*

  => the more file structure information is in the header, the less our software needs to know about the specific structure

- FixedFieldBuffer class : Appendix F
  - *NumFields* and *FieldSize* members
  - WriteHeader method
  - ReadHeader method

5.5.3 Metadata (1/4)

- File structure for images : Fig. 5.6
  - store one image per file
  - need information(≈ metadata) about each image

- Metadata
  - data that describes the primary data in a file
  - can be incorporated into any file (in the *header record*)
  - (e.g.) ① where the image is from
    ② when it was made
    ③ what camera was used
    ④ references to related images

5.5.3 Metadata (2/4)

Figure 5.6 To make sense of this 2-megabyte image, an astronomer needs such metadata as the kind of image it is, the part of the sky it is from, and the telescope that was used to view it. Astronomical metadata is often stored in the same file as the data itself. (This image shows polarized radio emission from the southern spiral galaxy NGC 5236 [M83] as observed with the Very Large Array radio telescope in New Mexico.)

5.5.3 Metadata (3/4)

- Standard format for holding metadata
  - FITS (Flexible Image Transport System)
    ① FITS header : Fig. 5.7
    - a collection of 2,880-byte blocks (bf=36) of 80-byte ASCII records for *metadata*

    ② actual binary numbers that describe the image
- FITS image
  - a good example of an abstract data model
  - the data itself is meaningless without the interpretive information contained in the header
  - FITS-specific methods : convert FITS data into an understandable image

5.5.3 Metadata (4/4)

```
SIMPLE  =                    T /CONFORMS TO BASIC FORMAT
BITPIX  =                   16 / BITS PER PIXEL
NAXIS   =                    2 / NUMBER OF AXES
NAXIS1  =                  256 / RA AXIS DIMENSION
NAXIS2  =                  256 / DEC AXIS DIMENSION
EXTEND  =                    F / T MEANS STANDARD EXTENSIONS EXIST
BSCALE  =          0.000100000 / TRUE = [TAPE*BSCALE]<pl>BZERO
BZERO   =          0.000000000 / OFFSET TO TRUE PIXEL VALUES
MAP_TYPE= 'REL EXPOSURE'       / INTENSITY OR RELATIVE EXPOSURE MAP
BUNIT   = '                    ' / DIMENSIONLESS PEAK EXPOSURE FRACTION
CRVAL1  =              0.625 / RA     REF POINT VALUE (DEGREES)
CRPIX1  =            128.500 / RA     REF POINT PIXEL LOCATION
CDELT1  =      -0.006666700 / RA     INCREMENT ALONG AXIS (DEGREES)
CTYPE1  = 'RA--TAN'            / RA     TYPE
CROTA1  =              0.000 / RA     ROTATION
CRVAL2  =             71.967 / DEC    REF POINT VALUE (DEGREES)
CRPIX2  =            128.500 / DEC    REF POINT PIXEL LOCATION
CDELT2  =        0.006666700 / DEC    INCREMENT ALONG AXIS (DEGREES)
CTYPE2  = 'DEC--TAN'           / DEC    TYPE
CROTA2  =              0.000 / DEC    ROTATION
EPOCH   =             1950.0 / EPOCH OF COORDINATE SYSTEM
ARR TYPE=                  4 / 1=DP, 3=FP, 4=I
DATAMAX =              1.000 / PEAK INTENSITY (TRUE)
DATAMIN =              0.000 / MINIMUM INTENSITY (TRUE)
ROLL ANG=            -22.450 / ROLL ANGLE (DEGREES)
BAD ASP =                  0 / 0=good, 1=bad(Do not use roll angle)
TIME LIV=             5649.6 / LIVE TIME (SECONDS)
OBJECT  = 'REM6791             ' / SEQUENCE NUMBER
AVGOFFY =              1.899 / AVG Y OFFSET IN PIXELS, 8 ARCSEC/PIXEL
AVGOFFZ =              2.578 / AVG Z OFFSET IN PIXELS, 8 ARCSEC/PIXEL
RMSOFFY =              0.083 / ASPECT SOLN RMS Y PIXELS, 8 ARCSC/PIX
RMSOFFZ =              0.204 / ASPECT SOLN RMS Z PIXELS, 8 ARCSC/PIX
TELESCOP= 'EINSTEIN            ' / TELESCOPE
INSTRUME= 'IPC                 ' / FOCAL PLANE DETECTOR
OBSERVER= '2                   ' / OBSERVER #: 0=CFA; 1=CAL; 2=MIT; 3=GSFC
GALL    =            119.370 / GALACTIC LONGITUDE OF FIELD CENTER
GALB    =              9.690 / GALACTIC LATITUDE OF FIELD CENTER
DATE_OBS= '80/238              ' / YEAR & DAY NUMBER FOR OBSERVATION START
DATE_STP= '80/238              ' / YEAR & DAY NUMBER FOR OBSERVATION STOP
TITLE   = 'SNR SURVEY: CTA1
ORIGIN  = 'HARVARD-SMITHSONIAN CENTER FOR ASTROPHYSICS
DATE    = '22/09/1989          ' / DATE FILE WRITTEN
TIME    = '05:26:53            ' / TIME FILE WRITTEN
END
```

Figure 5.7 Sample FITS header. On each line, the data to the left of the / is the actual metadata (data about the raw data that follows in the file). For example, the second line (BITPIX = 16) indicates that the raw data in the file will be stored in 16-bit integer format. Everything to the right of a / is a comment, describing for the reader the meaning of the metadata that precedes it. Even a person uninformed about the FITS format can learn a great deal about this file just by reading through the header.

5.5.4 Color Raster Images

- Color raster image
  - a rectangular array of colored dot ("*pixels*") that are displayed on a screen
- Metadata for a raster image

  (1) dimensions of the image (1024 * 768, 1280 * 1024)

  (2) number of bits per each pixel (8-bit : 256 colors, 16-bit : 65,000 colors)

  (3) color lookup table ("*palette*") : pixel values ~ colors
- Methods for images (≡ abstract data type)
  - read an image and store the image on disk
  - display an image in a window on a screen
  - associate an image with a particular color lookup table
  - overlay one image onto another (-->composite image)
  - display several images in succession (-->video)

5.5.5 Mixing Object Types in One File (1/7)

- File
  - a mixture of objects that may be very different in *content*
  - all records are not the same *structure*
  - (e.g.) *several* FITS images together with metadata and lab notes (Fig. 5.8)

=> *need a new kind of file structure*

5.5.5 Mixing Object Types in One File (2/7)

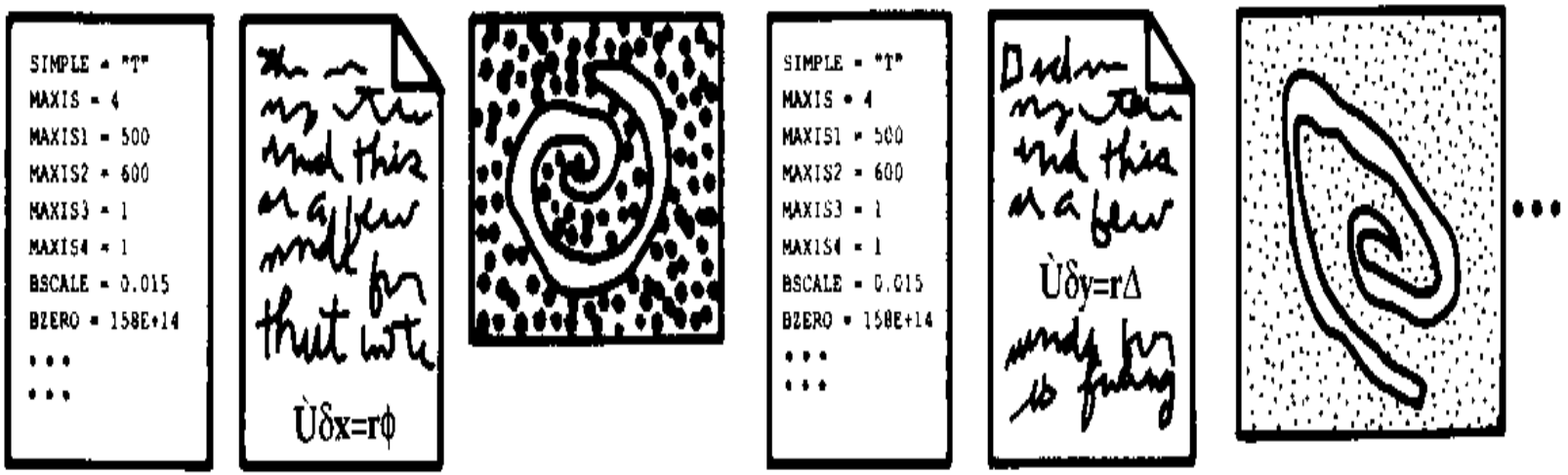


Figure 5.8 Information that an astronomer wants to include in a file.

5.5.5 Mixing Object Types in One File (3/7)

- Methods for new file design

1. Put each type of object into a variable-length record and write file processing programs so they know what each record looks like
   - ex) 1st record : header for the 1st image, 2nd record : image, 3rd record : document, 4th record : header for the 2nd image, and so forth
   - workable and simple
   - drawbacks

   ① object must be accessed sequentially (->*time-consuming*)

   ② file must contain exactly the objects that are described, in exactly the order indicated

   (*changes in the file's structure* -> rewrite all programs)

5.5.5 Mixing Object Types in One File (4/7)

2. Each record begins with a *keyword* that identifies the metadata record as in the FITS header
   - not work at all for objects that vary enormously in size and content
   - keyword : for the records in the headers, not for the headers themselves, images, and any other objects

3. Use an index table with tags
   - let each record be big enough to hold the object that is referenced by the keyword(tag)
   - place the keywords in an index table, together with the byte offset and a length indicator of the actual metadata (or data)

5.5.5 Mixing Object Types in One File (5/7)

(1) index table

- hold descriptive information about the primary data (keyword (*tag*) + offset + length)

(2) tags (= keywords)

- distinguish different types of object within a file

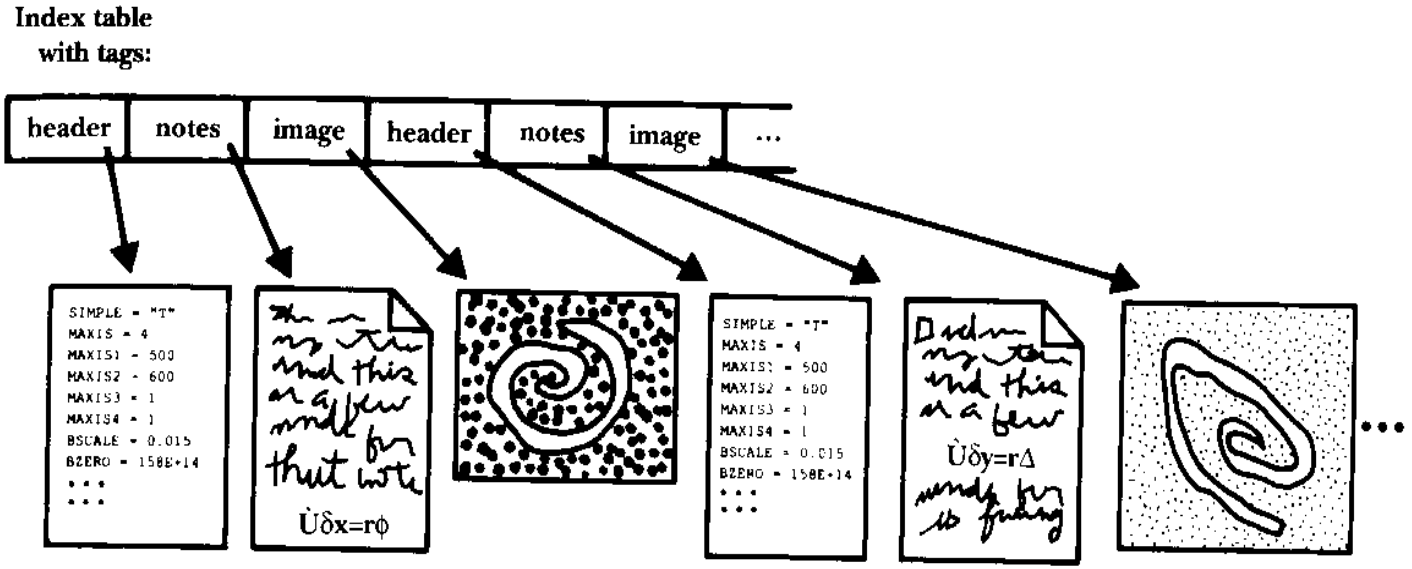


Figure 5.9 Same as Fig. 5.8, except with tags identifying the objects.

5.5.5 Mixing Object Types in One File (6/7)

- Tag structures in standard file formats

1. TIFF (Tagged Image File Format)
   - for storing images
2. HDF (Hierarchical Data Format)
   - for storing many different kinds of scientific data, including images
3. SGML (Standard General Markup Language)
   - *language* for describing document structures and for defining tags used to mark up that structure
4. FITS (Flexible Image Transport System)
   - for storing images

5.5.5 Mixing Object Types in One File (7/7)

- Access for files with mixtures of data objects

1. read an object of a particular type
   - use the index table
2. store an object in the file
   - store tags and put the object
3. determine the correct method for storing or retrieving an object

} Chap.6

5.5.6 Representation-Independent File Access (1/3)

- Software for abstract data model(=>object-oriented access)

1. It delegates to separate modules the responsibility of translating to and from the physical format of the object, letting the application modules concentrate on the task at hand
2. It opens up the possibility of working with objects that at some level fit the same abstract data model, even though they are stored in different format

5.5.6 Representation-Independent File Access (2/3)

- (e.g.) Image processing program (*find_star*)
  - operate in memory on a collection of 8-bit images
  - { some images : stored in FITS files
    other images : stored in TIFF files
  - read_image() for object-oriented access

① determines the format of the images within a file

② invokes the proper procedure to read the image in that format

③ converts it from that format into the 8-bit memory format

5.5.6 Representation-Independent File Access (3/3)

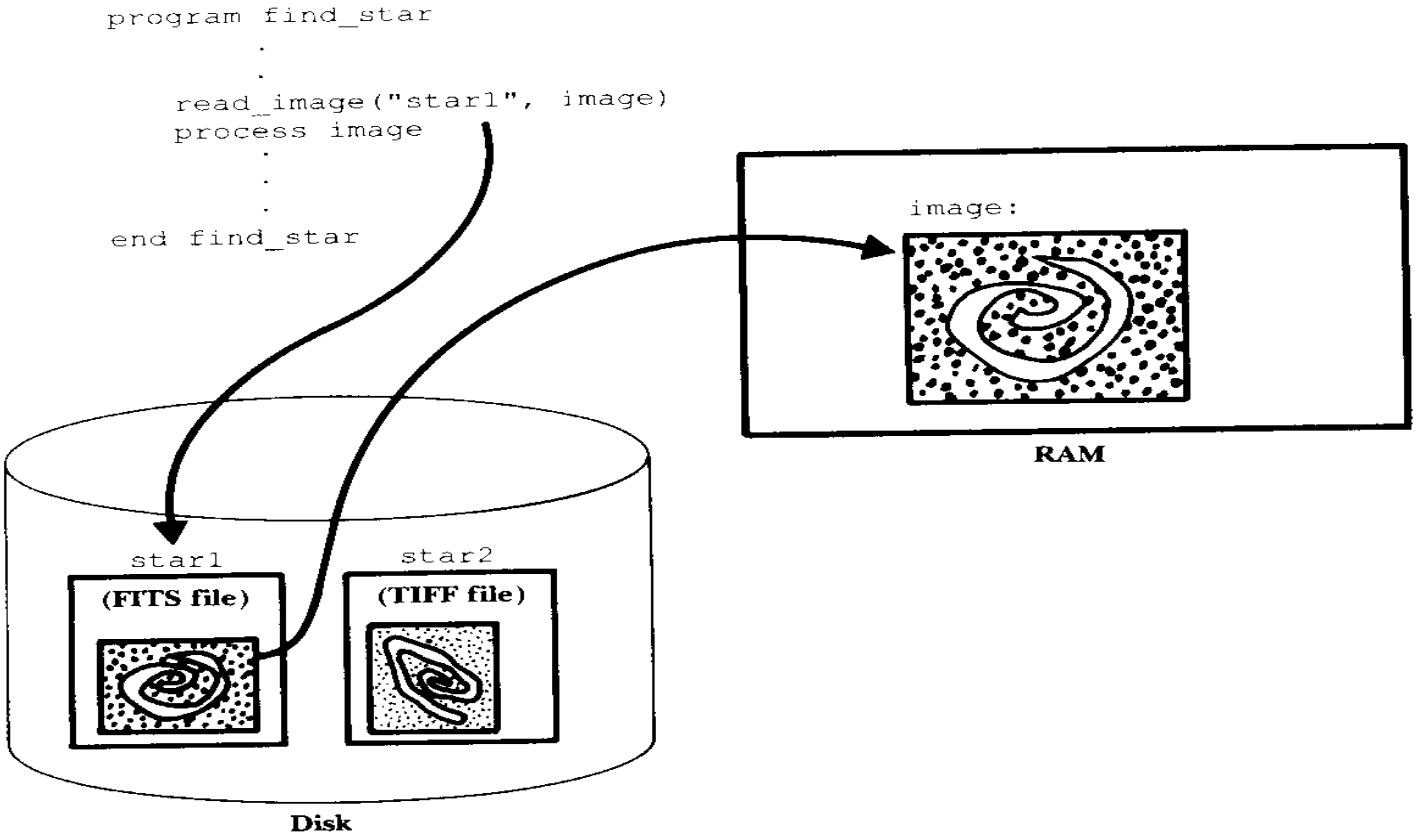
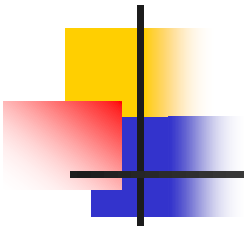


Figure 5.10 Example of object-oriented access. The program `find_star` knows nothing about the file format of the image that it wants to read. The routine `read_image` has methods to convert the image from whatever format it is stored in on disk into the 8-bit in-memory format required by `find_star`.



5.5.7 Extensibility

- Access to a mixture of objects
  - a mechanism for choosing the appropriate *methods* for a given type of object
- Extensibility
  - extend the *types of objects* that the software can support
  - extend the *methods* to the repertoire of methods

5.6 Portability and Standardization
5.6.1 Factors Affecting Portability (1/2)

- Factors

1. differences among operating systems
   - the ultimate physical format of the same logical file
   - MS-DOS
     - adds an extra line-feed character every time it encounters a carriage return character
   - most other file systems (e.g., UNIX) : not the case
2. differences among languages
   - the physical layout of files produced with different languages
   - C, C++
     - can make header records and data records different sizes
     - can mix and match fixed record lengths
   - Pascal
     - must use the same size for every record in the file
     - all records in a nontext file must be the same size

5.6.1 Factors Affecting Portability (2/2)

3. differences in machine architectures
   - the representation of numbers, records, text, etc
   - Sun Ultra
     - the hexadecimal value of $500{,}000{,}000_{10}$ is $1dcd6500_{16}$
   - IBM PC, VAX
     - the hexadecimal value of $500{,}000{,}000_{10}$ is $0065cd1d_{16}$
   - The size of INTEGER is 8 bytes(64-bit word), 4 bytes and 2 bytes in a Cray 2, a Sun 3 and a IBM PC, respectively
   - EBCDIC is a standard created by IBM, and most others support ASCII

5.6.2 Achieving Portability (1/5)

- Guidelines for achieving portability

1. Agree on a *standard physical record format* and stay with it
   - physical standard
     - one that is represented the same physically, no matter what language, machine, or operating system is used
     - (e.g.) FITS : header + images

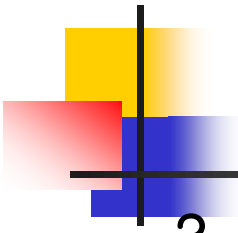
5.6.2 Achieving Portability (2/5)

2. Agree on a *standard binary encoding* for data elements (client/server environments)
   (1) text
   - ASCII, EBCDIC

   (2) number

   ① IEEE Standard formats
   - 32-bit, 64-bit, 128-bit floating point numbers
   - 8-bit, 16-bit, 32-bit integers

   ② External Data Representation (XDR)
   - specify a set of standard encodings for all files (IEEE encodings)
   - provide for a set of routines for each machine for converting from its binary encoding, and vice versa



5.6.2 Achieving Portability (3/5)

3. Number and text conversion
   - conversion among *n* native format

   (1) direct conversion
   - *n(n−1)* translators

   (2) indirect conversion
   - via standard intermediate format (e.g., XDR, GML)
   - *2n* translators
   - need *two* conversions

5.6.2 Achieving Portability (4/5)

4. File structure conversion
   - software packages on images with different file formats

   (1) require that the user supply images in a format that is compatible with the one used by the package => *user*

   (2) process only images that adhere to some predefined standard format (e.g., FITS) => *software developer*

   (3) include different sets of I/O methods capable of converting an image from several different formats into a standard memory structure that the package can work with => *software developer*

5.6.2 Achieving Portability (5/5)

5. File system differences
   - transferring files between file systems
     - differences in the way files are organized physically
     - (e.g.) UNIX system ---> non-UNIX system
       (512-byte block) (2,880-byte block)
- dd utility
  - for copying tape data to and from UNIX systems
  - for converting data from any physical source

① convert from one block size to another
② convert fixed-length records to variable-length records, or vice versa
③ convert ASCII to EBCDIC, or vice versa
④ convert all characters to lowercase (or uppercase)
⑤ swap every pair of bytes