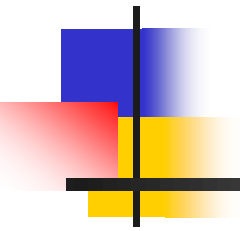
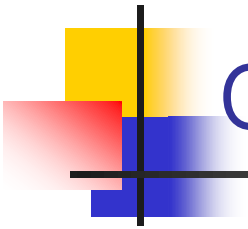


Chapter 4. Fundamental File Structure Concepts

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



Chapter Outline

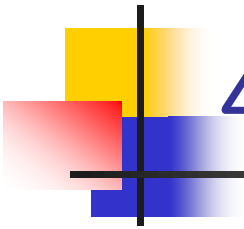
4.1 Field and Record Organization

4.2 Using Classes to Manipulate Buffers

4.3 Using Inheritance for Record Buffer Classes

4.4 Managing Fixed-Length, Fixed-Field Buffers

4.5 An Object-Oriented Class for Record Files



4.1 Field and Record Organization

- Field
  - *the smallest logically meaningful unit of information in a file*
- Aggregates of fields

  (1) Array
  - many copies of a single field

  (2) Record
  - a list of different fields
- Object (vs. record)
  - data residing in memory
- Members (vs. fields)
  - object's fields in memory

4.1.1 A Stream File

- writestr.cpp (operator <<) => Appendix D
  - write the fields of a *Person* to a file as a stream of bytes containing no added information => no way to get field apart
  - *keep the information divided into fields*

```
ostream & operator << (ostream & outputFile, Person & p)
{ // insert (write) fields into stream
   outputFile << p.Lastname
      << p.FirstName
      << p.Address
      << p.City
      << p.State
      << p.ZipCode;
   return outputFile;
}
```

Mary Ames	Alan Mason
123 Maple	90 Eastgate
Stillwater, OK 74075	Ada, OK 74820

AmesMarry123 MapleStillwaterOK74075MasonAlan90 EastgateAdaOK74820

4.1.2 Field Structures (1/5)

- Structure of fixed-length fields

(1) In C:

```
Struct Person{
    char last [11];
    char first [11];
    char address [16];
    char city [16];
    char state [3];
    char zip[10];
};
```

(2) In C++:

```
class Person { public:
    char last [11];
    char first [11];
    char address [16];
    char city [16];
    char state [3];
    char zip [10];
};
```

4.1.2 Field Structures (2/5)

- Methods for structuring fields

1. Fix the length of fields : Fig. 4.3(a)
   - force the fields into a predictable length (i.e., fixed-length fields)
   - disadv.
     - adding all the padding required to bring the fields up to a fixed length makes the file much larger
     - data can be too long to fit into the allocated amount of space
   - inappropriate
     - a large amount of variability in the length of fields
   - appropriate
     - every field is already fixed in length, or very little variation in field lengths

4.1.2 Field Structures (3/5)

2. Begin each field with a length indicator : Fig. 4.3(b)
   - store the field length just ahead of the field
   - one byte for up to 256-byte field

3. Separate the fields with delimiters : Fig. 4.3(c)
   - use some special character or sequence of characters that will not appear within a field as a delimiter
   - delimiter character
     - *white-space characters* (e.g., blank, new line, tab) or *special characters* (e.g., vertical bar character "|")

4.1.2 Field Structures (4/5)

4. Use a "Keyword=Value" expression to identify fields
   : Fig. 4.3(d)

- adv.
  - self-describing structure (i.e., a field provides information about itself)
  - good format for dealing with missing fields
- disadv.
  - waste a lot of space for the keywords

4.1.2 Field Structures (5/5)

```
Ames      Mary     123 Maple       Stillwater     OK74075
Mason     Alan     90 Eastgate     Ada            OK74820
```

(a)

```
04Ames04Mary09123 Maple10Stillwater02OK0574075
05Mason04Alan1190 Eastgate03Ada02OK0574820
```

(b)

```
Ames|Mary|123 Maple|Stillwater|OK|74075|
Mason|Alan|90 Eastgate|Ada|OK|74820|
```

(c)

```
last=Ames|first=Mary|address=123 Maple|city=Stillwater|
state=OK|zip=74075|
```

(d)

Figure 4.3 Four methods for organizing fields within records. (a) Each field is of fixed length. (b) Each field begins with a length indicator. (c) Each field ends with a delimiter |. (d) Each field is identified by a key word.

4.1.3 Reading a Stream of Fields

- Extraction operator (operator >>) => Appendix D
  - reads the stream of bytes, breaking the stream into fields and storing it as a *Person* object

```
istream & operator >> (istream & stream, Person & p)
{ // read delimited fields from file
   char delim;
   stream.getline (p.LastName, 30, ' | ');
   if (strlen (p.LastName) ==0) return stream;
   stream.getline (p.FirstName, 30, ' | ');
   stream.getline (p.Address, 30, ' | ');
   stream.getline (p.City, 30, ' | ');
   stream.getline (p.State, 15, ' | ');
   stream.getline (p.ZipCode, 10, ' | ');
   return stream;
}
```

```
Last Name: 'Ames'
First Name: 'Mary'
Address: '123 Maple'
City: 'Stillwater
State: 'OK'
Zip Code: '74075'
Last Name: 'Mason'
First Name: 'Alan'
Address: '90 Eastgate'
City: 'Ada'
State: 'OK'
Zip Code: '74820'
```

4.1.4 Record Structures (1/5)

- Record
  - *a set of fields that belong together when the file is viewed in terms of a higher level of organization*

- Methods for structuring records

1. Make records a predictable number of bytes (*fixed-length records*) : Fig. 4.5(a)
   - each record contains the same number of bytes
   - the most commonly used methods
   - not imply that the size or number of fields in the record must be fixed
   - frequently used to hold variable numbers of variable-length fields : Fig. 4.5(b)

4.1.4 Record Structures (2/5)

2. Make records a predictable number of fields : Fig. 4.5(c)
   - each record contains a fixed number of fields

3. Begin each record with a length indicator : Fig. 4.6(a)
   - each record contains a field indicating how many bytes there are in the record
   - commonly used for handling variable-length records

4. Use an index to keep track of addresses : Fig. 4.6(b)
   - use an *index* to keep a byte offset for each record in the original file
   - two-file mechanism

4.1.4 Record Structures (3/5)

5. Place a delimiter(e.g., “#”) at the end of each record : Fig. 4.6(c)
   - use *end-of-line character* (e.g., \n on UNIX or CR/NL on other O.S.) as a record delimiter

4.1.4 Record Structures (4/5)

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)

Ames¦John¦123 Maple¦Stillwater¦OK¦74075¦Mason¦Alan¦90 Eastgate¦Ada¦OK . . .

(c)

FIGURE 4.5 Three ways of making the lengths of records constant and predictable. (a) Counting bytes: fixed-length records with fixed-length fields. (b) Counting bytes: fixed-length records with variable-length fields. (c) Counting fields: six fields per record.

4.1.4 Record Structures (5/5)

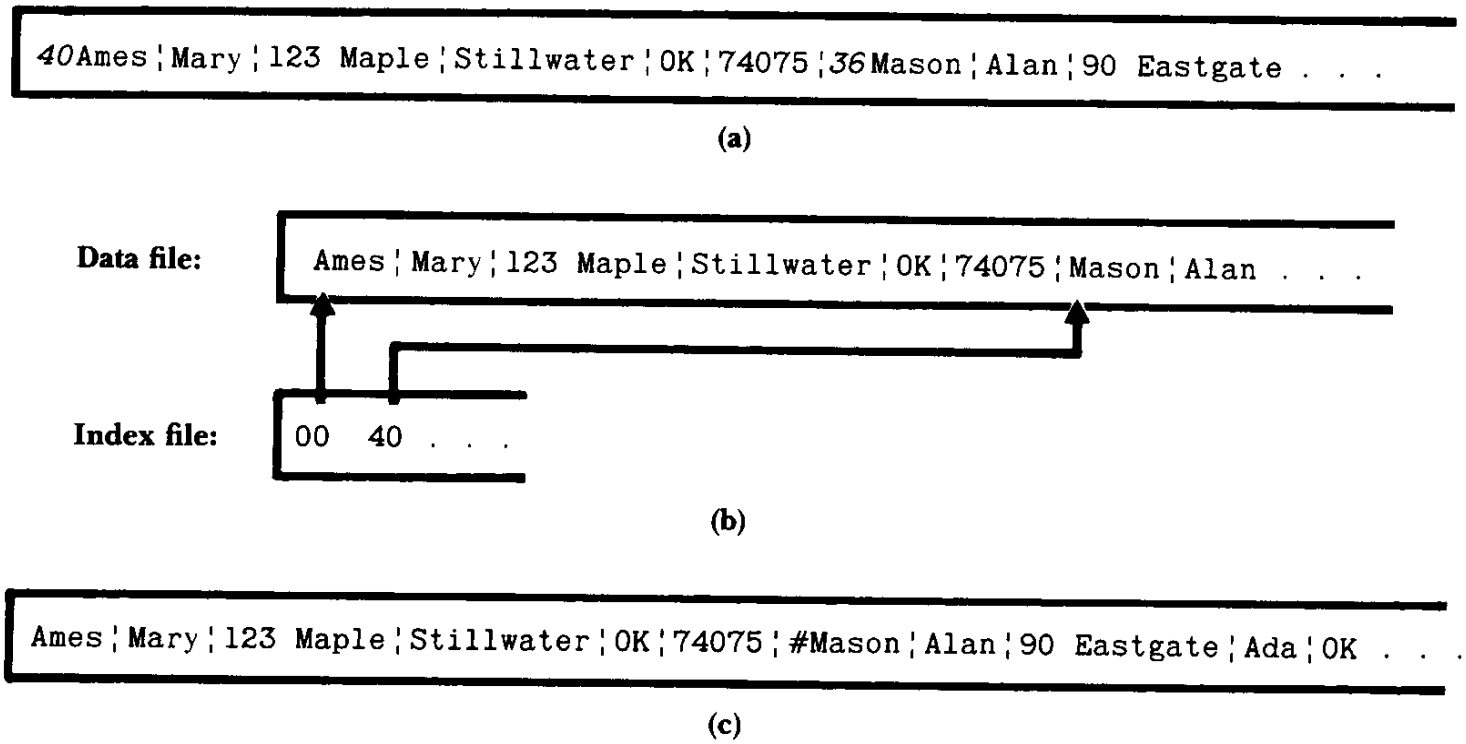


Figure 4.6 Record structures for variable-length records. (a) Beginning each record with a length indicator. (b) Using an index file to keep track of record addresses. (c) Placing the delimiter # at the end of each record.

4.1.5 A Record Structure That Uses a Length Indicator (1/4)

- Method for selecting record organization
  - the nature of data
  - what you need to do with it
- Writing the variable-length records to the file
  - a record-length field at the beginning of the record

  => the sum of the lengths of the fields in each record
  - form of the record-length field

  => binary integer or ASCII characters

4.1.5 A Record Structure That Uses a Length Indicator (2/4)

- WritePerson function
  - write a variable-length, delimited buffer to a file
  - buffer : character array for fields and filed delimiters

```
const int MaxBufferSize = 200
int WritePerson (ostream & stream, Person & p)
{   char buffer [MaxBufferSize] ; // create buffer of fixed size
    strcpy (buffer, p.LastName) ; strcat (buffer, " | ") ;
    strcat (buffer, p.FirstName) ; strcat (buffer, " | ") ;
    strcat (buffer, p.Address) ; strcat (buffer, " | ") ;
    strcat (buffer, p.City) ; strcat (buffer, " | ") ;
    strcat (buffer, p.State) ; strcat (buffer, " | ") ;
    strcat (buffer, p.ZipCode) ; strcat (buffer, " | ") ;
    short length=strlen (buffer)
    stream.write (&length, sizeof(length)) ; //write length
    stream.write(&buffer, length) ; //write buffer
}
```

4.1.5 A Record Structure That Uses a Length Indicator (3/4)

- Representing the record length
  - write the length in the form of a 2-byte binary integer in C
  - convert the length into a character (decimal) string using formatted output
    - fprintf (file, "%d ", length); // with C stream
    - stream << length << ' '; // with C++ stream classes
- readvar.cpp => Appendix D
  - implement record structure using binary field for length

40Ames|Marry|123 Maple|Stillwater|OK|74075|36Mason|Alan|90Eastgate|Ada|OK|74820|

4.1.5 A Record Structure That Uses a Length Indicator (4/4)

- ReadVariablePerson function
  (i) read the length of a record
  (ii) move the characters of the record into a buffer
  (iii) break the record into fields

```
int ReadVariablePerson (istream & stream, Person & p)
{ // read a variable sized record from stream and store it in p
   short length;
   stream.read (&length, sizeof(length));
   char * buffer = new char[length+1]; // create buffer space
   stream.read (buffer, length);
   buffer[length] = 0; // terminate buffer with null
   istrstream strbuff (buffer); // create string array input stream
   strbuff >> p; // use the istream extraction operator (See Fig.4.4)
   return 1;
}
```

buffer

Ames|Marry|123 Maple|Stillwater|OK|74075|

Length : 40

4.1.6 Mixing Numbers and Characters : Use of a File Dump (1/3)

- Number 40, 36

	Decimal value of number	Hex value stored in bytes		ASCII character form	
(a) stored as	40	34	30	'4'	'0'
ASCII characters:	36	33	36	'3'	'6'
(b) stored as	40	00	28	'\0'	'('
a 2-byte integer:	36	00	24	'\0'	'$'

***40*Ames|John|123 Maple|Stillwater|OK|74075|*36*Mason|Alan|90 Eastgate ……**

(Ames|John|123 Maple|Stillwater|OK|74075| $Mason|Alan|90 Eastgate ……

0x28 **0x00** **0x24** **0x00**

4.1.6 Mixing Numbers and Characters : Use of a File Dump (2/3)

	Dec.	Oct.	Hex.		Dec.	Oct.	Hex.		Dec.	Oct.	Hex.		Dec.	Oct.	Hex.
nul	0	0	0	sp	32	40	20	@	64	100	40	‘	96	140	60
sol	1	1	1	!	33	41	21	A	65	101	41	a	97	141	61
stx	2	2	2	“	34	42	22	B	66	102	42	b	98	142	62
etx	3	3	3	#	35	43	23	C	67	103	43	c	99	143	63
eot	4	4	4	$	36	44	24	D	68	104	44	d	100	144	64
enq	5	5	5	%	37	45	25	E	69	105	45	e	101	145	65
ack	6	6	6	&	38	46	26	F	70	106	46	f	102	146	66
bel	7	7	7	’	39	47	27	G	71	107	47	g	103	147	67
bs	8	10	8	(	40	50	28	H	72	110	48	h	104	150	68
ht	9	11	9	)	41	51	29	I	73	111	49	i	105	151	69
nl	10	12	A	*	42	52	2A	J	74	112	4A	j	106	152	6A
vt	11	13	B	+	43	53	2B	K	75	113	4B	k	107	153	6B
np	12	14	C	,	44	54	2C	L	76	114	4C	l	108	154	6C
cr	13	15	D	–	45	55	2D	M	77	115	4D	m	109	155	6D
so	14	16	E	.	46	56	2E	N	78	116	4E	n	110	156	6E
si	15	17	F	/	47	57	2F	O	79	117	4F	o	111	157	6F
dle	16	20	10	0	48	60	30	P	80	120	50	p	112	160	70
dc1	17	21	11	1	49	61	31	Q	81	121	51	q	113	161	71
dc2	18	22	12	2	50	62	32	R	82	122	52	r	114	162	72
dc3	19	23	13	3	51	63	33	S	83	123	53	s	115	163	73
dc4	20	24	14	4	52	64	34	T	84	124	54	t	116	164	74
nak	21	25	15	5	53	65	35	U	85	125	55	u	117	165	75
syn	22	26	16	6	54	66	36	V	86	126	56	v	118	166	76
etb	23	27	17	7	55	67	37	W	87	127	57	w	119	167	77
can	24	30	18	8	56	70	38	X	88	130	58	x	120	170	78
em	25	31	19	9	57	71	39	Y	89	131	59	y	121	171	79
sub	26	32	1A	:	58	72	3A	Z	90	132	5A	z	122	172	7A
esc	27	33	1B	;	59	73	3B	[	91	133	5B	{	123	173	7B
fs	28	34	1C	<	60	74	3C	\	92	134	5C	|	124	174	7C
gs	29	35	1D	=	61	75	3D	]	93	135	5D	}	125	175	7D
rs	30	36	1E	>	62	76	3E	^	94	136	5E	~	126	176	7E
us	31	37	1F	?	63	77	3F	–	95	137	5F	del	127	177	7F

4.1.6 Mixing Numbers and Characters : Use of a File Dump (3/3)

- Common file structure
  - each record has both binary and ASCII data (mixing data type), and consists of a fixed-length field (byte count) and several delimited, variable-length fields.

4.2 Using Classes to Manipulate Buffers

- Buffer class
  - to encapsulate the pack, unpack, read, and write operations of buffer objects

1. For output
   (i) start with an empty buffer object
   (ii) pack field values into the object one by one
   (iii) write the buffer contents to an output stream
2. For input
   (i) initialize a buffer object by reading a record from an input stream
   (ii) extract(unpack) the object's field values

4.2.1 Buffer Class for Delimited Text Fields (1/3)

- DelimitedTextBuffer class => Appendix E
  - support variable-length buffers whose fields are represented as delimited text
  - full class in *deltext.h and deltext.cpp*

```
class DelimTextBuffer
{public:
      DelimTextBuffer (char Delim = '|', int maxBytes = 1000);
      int Read (istream & file);
      int Write (ostream & file) const;
      int Pack (const char * str, int size = -1);
      int Unpack (char *str);
private:
      char Delim;          // delimiter character
      char * buffer;       // character array to hold field values
      int BufferSize;      // current size of packed fields
      int MaxBytes;        // max # of characters in buffer
      int NextByte;        // packing, unpacking position in buffer
}
```

4.2.1 Buffer Class for Delimited Text Fields (2/3)

```
Person MaryAmes;          // declare objects of class Person
DelimTextBuffer buffer;  // declare objects of class DelimTextBuffer
buffer.Pack (MaryAmes.Lastname);  // pack the person into the buffer
buffer.Pack (MaryAmes.FirstName); // copy the characters to the buffer
                          ....                    // and add the delimiter
buffer.Pack (MaryAmes.ZipCode);
buffer.Write (stream);                    // write the buffer to a file

int DelimTextBuffer :: Read (istream & stream)  // Read method
{           Clear();  //clear the current buffer contents
            stream.read((char *)&BufferSize, sizeof(BufferSize)); //read the record size
            if (stream.fail()) return FALSE;
            if (BufferSize > MaxBytes) return FALSE; //buffer overflow
            stream.read(Buffer, BufferSize);
           return stream.good();
}
```

BufferSize : 40

Buffer → **Ames|Marry|123 Maple|Stillwater|OK|74075|**

4.2.1 Buffer Class for Delimited Text Fields (3/3)

```
int DelimTextBuffer :: Pack (const char * str, int size)
// set the value of the next field of the buffer;
// if size = -1 (default) use strlen(str) as Delim of field
{       short len; // length of string to be packed
        if (size >= 0) len = size;
        else len = strlen (str);
        if (len > strlen(str)) // str is too short!
                return FALSE;
        int start = NextByte; // first character to be packed
        NextByte += len + 1;
        if (NextByte > MaxBytes) return FALSE;
        memcpy (&Buffer[start], str, len);
        Buffer [start+len] = Delim; // add delimeter
        BufferSize = NextByte;
        return TRUE;
}
```

4.2.2 Extending Class Person with Buffer Operations

- Buffer for an object(e.g., Person)
  - specify the order in which the members of the object are packed and unpacked

```
Int Person::Pack (DelimTextBuffer & Buffer) const
{// pack the fields into a DelimTextBuffer
   int result;
   result = Buffer.Pack (LastName);
   result = result && Buffer.Pack (FirstName);
   result = result && Buffer.Pack (Address);
   result = result && Buffer.Pack (City);
   result = result && Buffer.Pack (State);
   result = result && Buffer.Pack (ZipCode);
   return result;
}
```

4.2.3 Buffer Classes for Length-Based and Fixed-length Fields (1/3)

- LengthTextBuffer class => Appendix E
  - full class in *lentext.h* and *lentext.cpp*

```
class LengthTextBuffer
{public:
      LengthTextBuffer (int maxBytes = 1000);
      int Read (istream & file);
      int Write (ostream & file) const;
      int Pack (const char * field, int size = -1);
      int Unpack (char *field);
 private:
      char * buffer;        // character array to hold field values
      int BufferSize;       // current size of packed fields
      int MaxBytes;         // max # of characters in buffer
      int NextByte;         // packing, unpacking position in buffer
}
```

4.2.3 Buffer Classes for Length-Based and Fixed-length Fields (2/3)

- FixedTextBuffer class
  - full class in *fixtext.h* and *fixtext.cpp*

```
class FixedTextBuffer
{public:
        FixedTextBuffer (int maxBytes = 1000);
        int AddField (int fieldSize);
        int Read (istream & file);
        int Write (ostream & file) const;
        int Pack (const char * field, int size = -1);
        int Unpack (char *field);
private:
        char * buffer;          // character array to hold field values
        int BufferSize;         // current size of packed fields
        int MaxBytes;           // max # of characters in buffer
        int NextByte;           // packing, unpacking position in buffer
        int NumFields;          // actual number of defined fields
        int MaxFields;          // maximum number of fields
        int * FieldSizes;       // array of field sizes
}
```

4.2.3 Buffer Classes for Length-Based and Fixed-length Fields (3/3)

- FixedTextBuffer class(cont'd)
  - use a fixed collection of fixed-length fields
  - use fixed-length records
  - AddField : support the specification of the fields and their size

```
int Person::InitBuffer (FixedTextBuffer & buffer)
{
    buffer.Init(6, 61);          // 6 fields, 61 bytes total
    buffer.AddField (10);        // LastName[11];
    buffer.AddField (10);        // FirstName[11];
    buffer.AddField (15);        // Address[16];
    buffer.AddField (15);        // City[16];
    buffer.AddField (2);         // State[3];
    buffer.AddField (9);         // ZipCode[10];
    return 1;
}
```

4.2.3 Buffer Classes for Length-Based and Fixed-length Fields (3/3)

```
int FixedTextBuffer :: AddField (int fieldSize)
{           if (NumFields == MaxFields) return FALSE;
            if (BufferSize + fieldSize > MaxBytes) return FALSE;
            FieldSize[NumFields] = fieldSize;
            NumFields ++;
            BufferSize += fieldSize;
            return TRUE;
}
int FixedTextBuffer :: Read (istream & stream)
{           stream . read (Buffer, BufferSize);
            return stream . good ();
}
int FixedTextBuffer :: Write (ostream & stream)
{           stream . write (Buffer, BufferSize);
            return stream . good ();
}
```

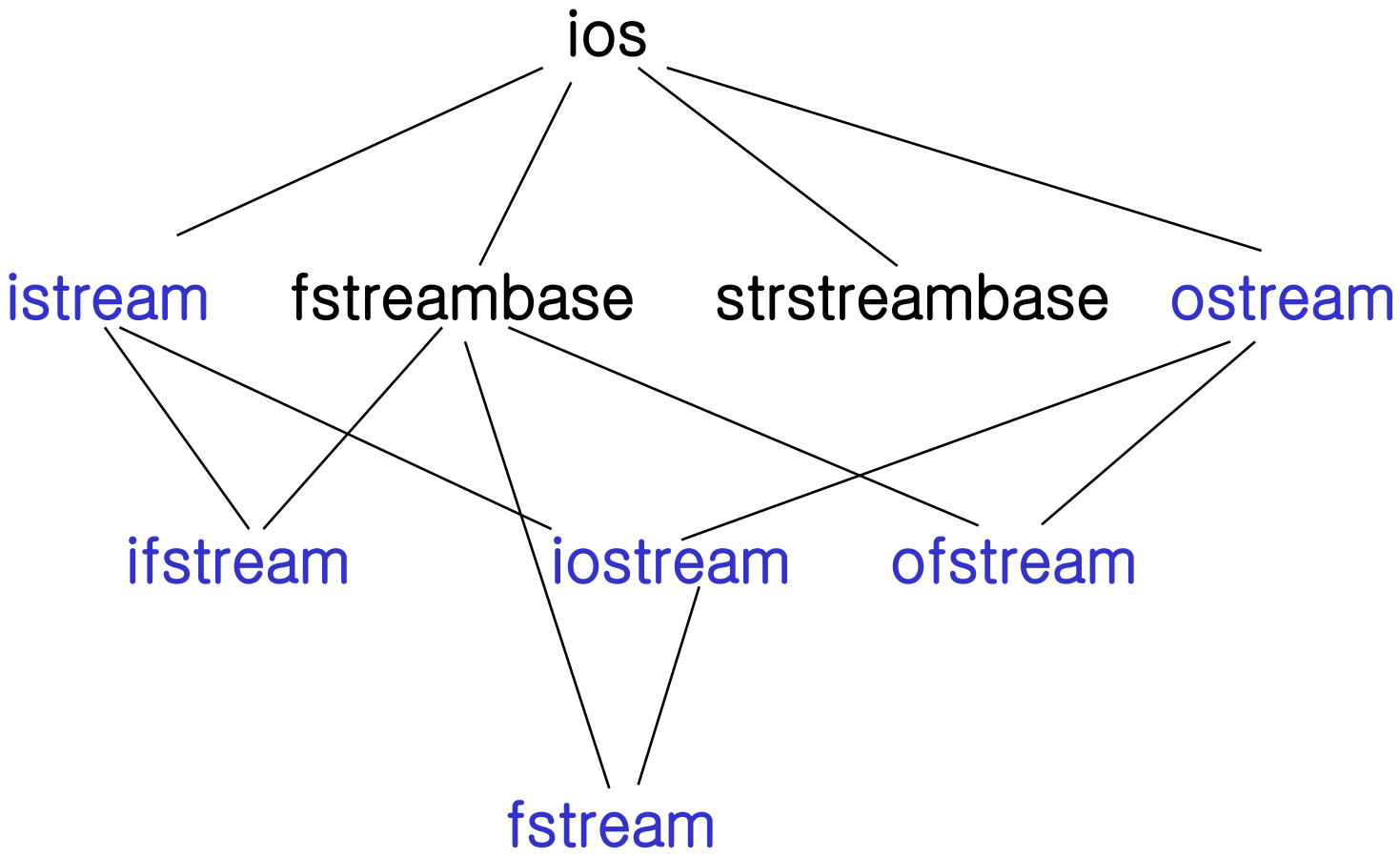
4.3 Using Inheritance for Record Buffer Classes

- cpp files for the three classes
  - a large percentage of the code is duplicated

  => use of the *inheritance* to eliminate almost all of the duplication

4.3.1 Inheritance in the C++ Stream Classes (1/3)

- Multiple inheritance in C++



4.3.1 Inheritance in the C++ Stream Classes (2/3)

```
class istream: virtual public ios {  . . .
class ostream: virtual public ios {  . . .
class iostream: public istream, public ostream { . . .
class ifstream: public fstreambase, public istream {  . . .
class ofstream: public fstreambase, public ostream { . . .
class fstream: public fstreambase, public iostream { . . .
```

- Class istream
  - define the read operations, the extraction operators
- Class ostream
  - define the write operations
- Class ios
  - define basic stream operations
- Class fstreambase
  - for access to operation system file operations

4.3.1 Inheritance in the C++ Stream Classes (3/3)

```
int ReadVariablePerson (istream & stream, Person & p)
{
        . . .
        istrstream strbuff (buffer);  //create string array input stream
        strbuff >> p;                 //use the istream extraction operator
        return 1;
}
```

- Use an istrstream object strbuff to contain a string buffer
  - istrstream is derived from istream
  - strbuff >> p : manipulated by istream operation

4.3.2 A Class Hierarchy for Record Buffer Objects (1/5)

- Buffer class hierarchy => Appendix F

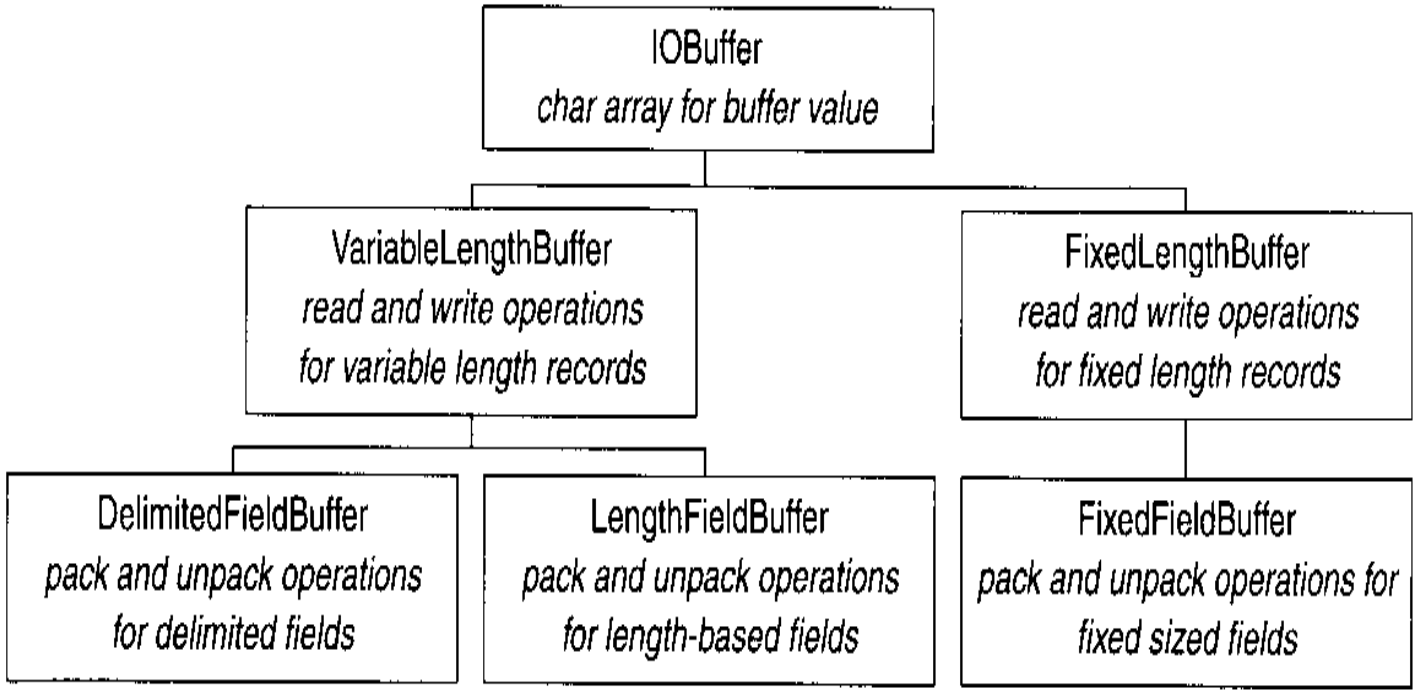


Figure 4.14 Buffer class hierarchy

4.3.2 A Class Hierarchy for Record Buffer Objects (2/5)

```
class IOBuffer
{public:
    IOBuffer (int maxBytes = 1000);
    virtual int Read (istream &) = 0;              //read a buffer
    virtual int Write (ostream &) const = 0;       //write a buffer
    virtual int Pack(const void * field, int size = -1) = 0;
    virtual int Unpack (void * field, int maxbytes = -1) = 0;
 protected:
    char * Buffer;     //character array to hold field values
    int BufferSize;    //sum of the sizes of packed fields
    int MaxBytes;      //max # of char in the buffer
    int NextByte;      // index of next byte to be packed/unpacked
 };
```

- ⇒ Define Read, Write, Pack, Unpack as virtual to allow each subclass to define its own implementation
- ⇒ Use pure virtual function (= 0) : 파생 클래스에서 반드시 재정의해야 하는 함수
- ⇒ Dread, Dwrite, ReadHeader, WriteHeader, …

4.3.2 A Class Hierarchy for Record Buffer Objects (3/5)

- VariableLengthBuffer and FixedLengthBuffer classes
  - support the read and write operations for different types of records

```
class VariableLengthBuffer: public IOBuffer
{ public:
   VariableLengthBuffer (int MaxBytes = 1000);
   int Read (istream &);
   int Write (ostream &) const;
   int PackFixLen (void *, int);
   int PackDelimeted (void *, int);
   int PackLength (void *, int);
   int SizeOfBuffer () const; // return current size of buffer
};
```

4.3.2 A Class Hierarchy for Record Buffer Objects (4/5)

- LengthFieldBuffer, DelimFieldBuffer, FixedFieldBuffer classes
  - have the pack and unpack methods for the specific field representation

```
class DelimFieldBuffer: public VariableLengthBuffer
{ public:
    DelimFieldBuffer (char Delim = -1, int maxBytes = 1000);
    int Pack (const void*, int size = -1);
    int Unpack (void * field, int maxBytes = -1);
 protected:
   char Delim;
};
```

4.3.2 A Class Hierarchy for Record Buffer Objects (5/5)

- Persistent objects
  - move objects from memory to files
  - ensure that fields are packed and unpacked

```
int Person::Unpack(IOBuffer & Buffer)
{
 Clear();
 int numBytes;
 numBytes = Buffer.Unpack (LastName); //which unpack method ?:not compile time
 if (numBytes == -1) return FALSE;
 LastName[numBytes] = 0;
 numBytes = Buffer.Unpack (FirstName);
 if (numBytes == -1) return FALSE;
 FirstName[numBytes] = 0;
 . . . // unpack the other fields
  return TRUE;
}

Person MaryAmes;
DelimFieldBuffer Buffer;
Read(Buffer);
MaryAmes.Unpack (Buffer); //use the method DelimFieldBuffer::Unpack
```

4.4 Managing Fixed-Length, Fixed-Field Buffers (1/2)

- Read and write of fixed-length records
  - Write method : write the fixed-size record
  - Read method : must know the record size
  - ⇒ Use protected field for field sizes of FixedLengthBuffer object

```
class FixedFieldBuffer: public FixedLengthBuffer
{public:
 . . .
 int AddField (int fieldSize);  //define the next field
 int Pack ( );
 int Unpack ( );
 int NumberOfFields ( ) const; //return # of defined fields
protected:
 int * FieldSize;   //array to hold field sizes
 int MaxFields;    //max # of fields
 int NumFields;   //actual # of defined fields
};
```

4.4 Managing Fixed-Length, Fixed-Field Buffers (2/2)

- Initialization of FixedFieldBuffer

```
int Person::InitBuffer (FixedFieldBuffer & Buffer)
{
 int result;
 result = Buffer.AddField(10);                  //LastName [11];
 result = result && Buffer.AddField(10); //FirstName[11];
     . . .
return result;
}
```

⇒ Add the fields one at a time, each with its own size

- Preparation of a buffer for reading and writing objects

```
FixedFieldBuffer Buffer(6, 61); // 6 fields, 61 bytes total
Person MaryAmes;
MaryAmes.InitBuffer(Buffer);
```

4.5 An Object-Oriented Class for Record Files (1/3)

- Have to know how to transfer objects to and from files

- BufferFile class : file <~> buffer
  - support manipulation of files that are tied to specific buffer types
  - support the creation of an object BufferFile from a specific buffer object
    - to open and create files and to read and write records
  - full class in *buffile.h* and *buffile.cpp* of Appendix F

4.5 An Object-Oriented Class for Record Files (1/3)

- Class BufferFile

```
// Class to represent buffered file operations
//        Used in conjunction with the IOBuffer classes
// Each buffered file is associated with a disk file of a specific
//        record type.
// Each buffered file object has a buffer object which can be
used  for file I/O
// Sequential and random access read and write are supported
//        each write returns the record address of the record
//        this record address can be used to read that record
//        the values of the record address depend on the type of
file and buffer
```

4.5 An Object-Oriented Class for Record Files (2/3)

```
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();
};
```

4.5 An Object-Oriented Class for Record Files (3/3)

DelimFieldBuffer buffer; // a buffer is created

BufferFile file (buffer); // BufferFile object file is attached

file.Open (myfile);

file.Read ( ); // buffer contains the packed record
Person myobject;

myobject.Unpack(buffer); // put the record into myobject

[buffer.Unpack (myobject); // put the record into myobject]

⇒ BufferFile is combined with a fixed-length buffer

- 5.2 "More about Record Structures"
  - Put a header record on the beginning of each file
  - BufferFile::Open reads the record size from the file header