Reading and Writing Variable Length Records

1. How do we know how long the record is?
2. How do we store the length?

All techniques use more space than the actual data

The designers have to understand how data will be used so they can choose the implementation with the least wasted space

Read about file dumps so you can use them in your program

The stuff about numbers is particularly important because you may want to store them as strings of digits rather than as numbers

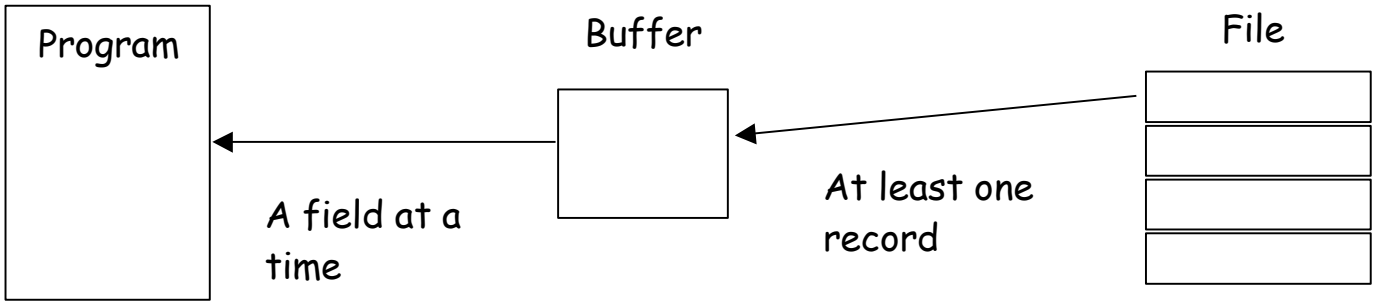
Buffering: We need to do this first

When you read information from a file you don't read a field as a time you read a buffer

This buffer usually contains at least one record

It may contain an integral multiple of records

It is in the buffer that the records are looked at as containing fields.



C++ Implementation of Buffers

Uses Inheritance for Record Buffer Classes

But first a reminder:

Public: Can be referenced directly from outside the class. Usually we make methods that change the state of the object public. We do not make the state public.

Private: Can be accessed only by methods within the class. The state variables in the class should be private.

Protected: Can be referenced directly by the class and all classes in the derived classes. We use this for methods. We do not make the state variables protected.

Virtual: Derived classes will supply their own version of the method. Binding occurs at run time. (i.e. the runtime system decides which version to run not the compiler)

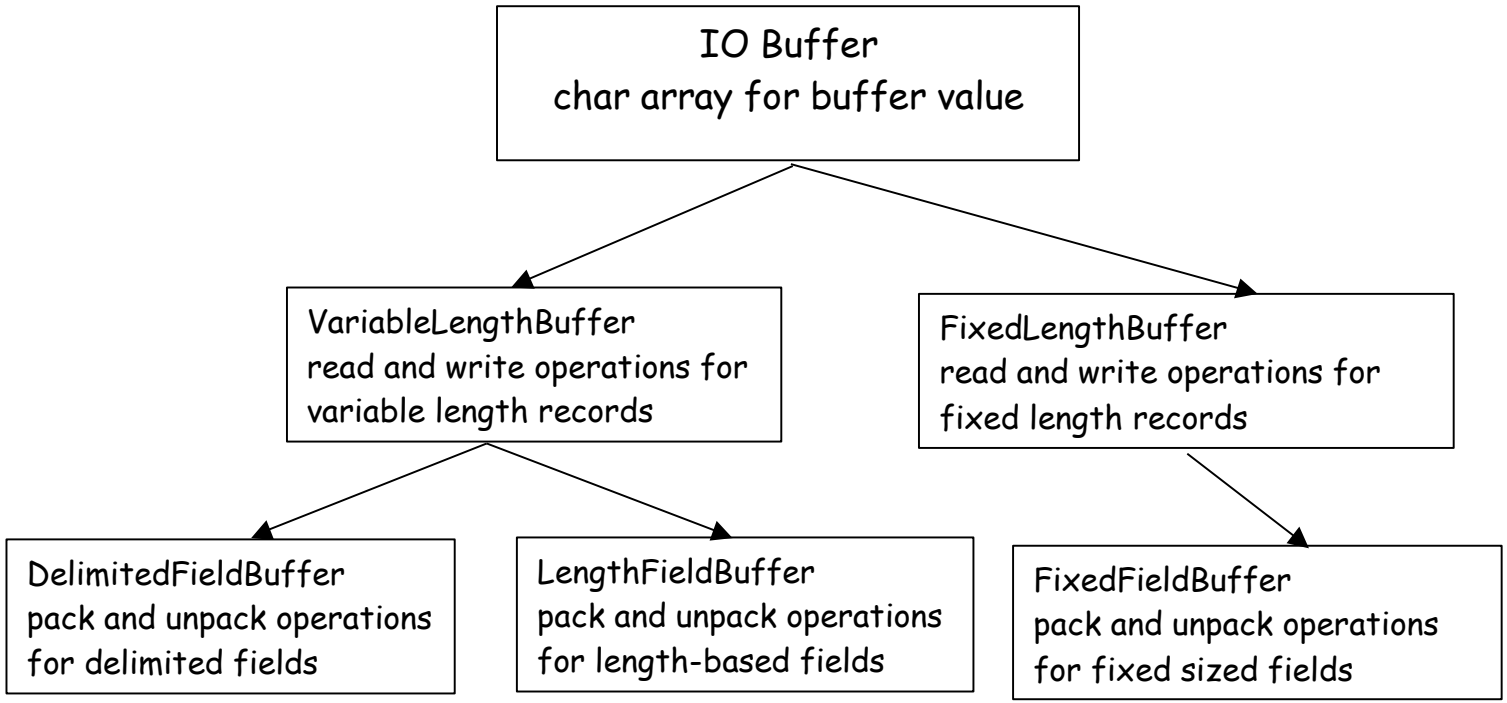
=0 means the derived class must give its own version

Friend: if a class designates a method as a friend then that method has access to the private data of the class but is not a member of the classes. Classes can also be friends. (You see this in redefining <<)

Static: when data members are static this means there is one per class and all objects in the class have access to that one data member

Static: when methods are static they are used to access static member data. They belong to the class and not the object so they are referenced using the class name.

Buffer Class Hierarchy



If we create 3 classes, one for each of the buffers we can make the implementation opaque to the user.

If we are lucky we can reuse some of the code.

Notice that VariableLengthBuffer and FixedLengthBuffer don't really know anything about the individual fields in the record.

They just know how to write the entire record

So, we can change the fields without needing to change these classes

Class BufferFile in appendix F

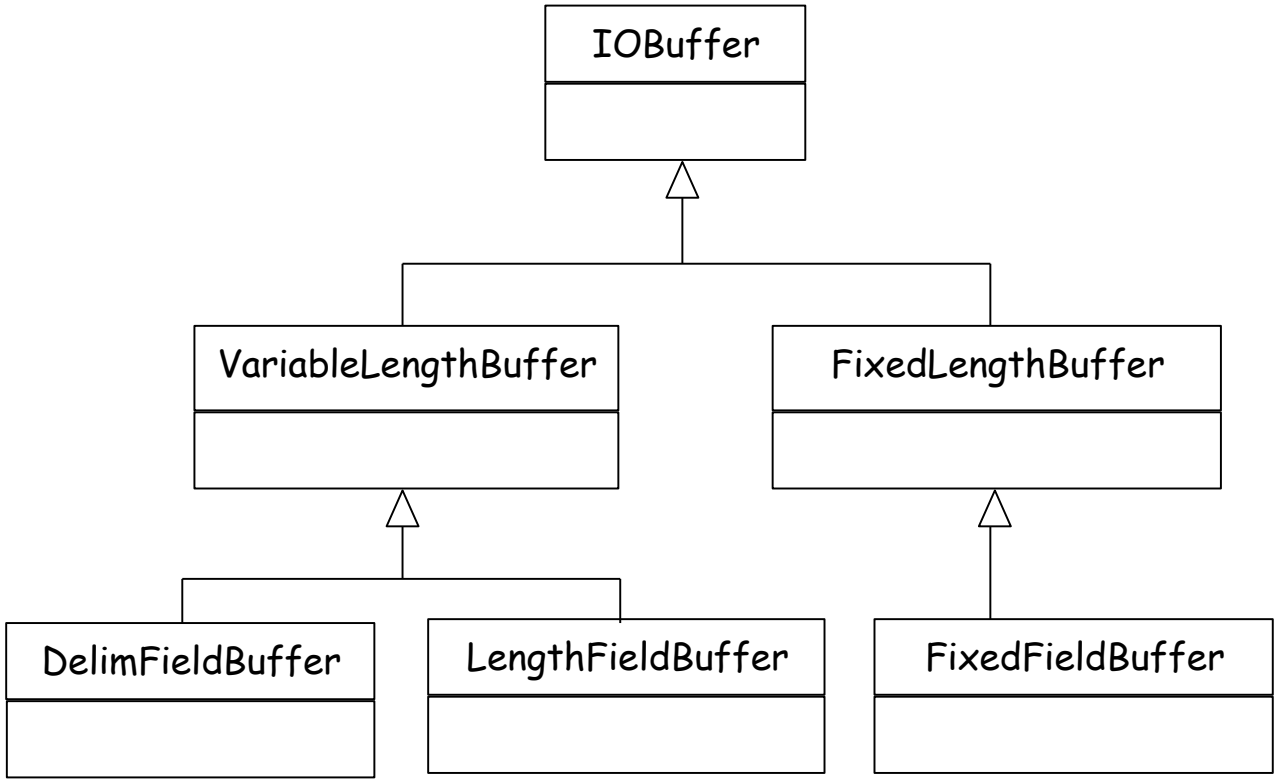
Supports manipulation of files that are tied to a specific buffer type

Each object of class BufferFile represents a specific file that uses a specific buffer object. So each file has only one buffer.

We use this object of type BufferFile to create, open, read, and write records to the file.

The records are written to the file through the buffer

```
DelimFieldBuffer (buffer);        //Creates buffer
BufferFile file(buffer);          //Attaches the buffer to file
file.open( Myfile );              // opens the file named MyFile
file.read( );                     // puts a copy of the next record
                                  // into the buffer
buffer.Unpack( MyObject);         // Unpacks the record just read into
                                  // MyObject
```



The buffer class hierarchy includes a class called FixedLengthBuffer even thought it may look like it is unnecessary.

IOBuffer	
+IOBuffer(int maxBytes) +IOBuffer & operator = (const IOBuffer &) +virtual void Clear() +virtual int Pack (const void * field, int size int size = -1) = 0 +virtual int Unpack (void * field, int maxBytes = -1) = 0 +virtual void Print (ostream &) const +int Init (int maxBytes) +virtual int Read (istream &) = 0 +virtual int Write (ostream &) const = 0 +virtual int DRead (istream &, int recref) +virtual int DWrite (ostream &, int recref) const +virtual int ReadHeader (istream &) +virtual int WriteHeader (ostream &) const	This class also indicates that packing routines for putting the fields into the buffer Notice that the fields are all derived from the super class field.
-int Initialized -char * Buffer -int BufferSize -int MaxBytes -int NextBytes -int Packing	

FixedLenghtBuffer	
+FixedLengthBuffer(int recodSize = 1000) +FixedLengthBuffer (const FixedLengthBuffer & Buffer) +void Clear() +int Read (istream &) +int Write (ostream &) const +int ReadHeader (istream &) +int WriteHeader (ostream &) const +void Print (ostream &) const +int sizeOfBuffer () const #int Init (int recordSize) #int changeRecordSize (int recordSize)	Notice this class has no data members It is an abstract class

FixedFieldBuffer	
+FixedFieldBuffer(int maxFields, int recodSize = 1000) +FixedFieldBuffer(int maxFields, int * fieldSize) +FixedFieldBuffer (const FixedLengthBuffer &) +FixedFieldBuffer & operator = (const FixedLengthBuffer &) +void Clear() +int addField (int fieldSize) +int ReadHeader (istream &) +int WriteHeader (ostream &) const +int pack (const void* field, int size = -1) +int unpack (void* field, int maxBytes = -1) +void Print (ostream &) const +int numberOfFields () const #int Init (int maxFields) #int Init (int numFields, int * fieldSize)	
# int * fieldsize #int maxFields #int numFields #int nextField	Why are these protected?