It is convenient to define a common data structure to hold an incoming/outgoing message/packet/frame as it is pro-
cessed. Such a structure should make it easy to add/remove headers and otherwise manipulate the information associ-
ated with a packet. This section describes a type called BUF.

Each instance of the BUF abstraction contains:

- A reference (in C, a pointer) to the data area of the buffer. Note that in general, not all of a buffer’s data area
  is in use at any time, because all messages/frames/packets are not the same size, and buffers generally come in
  “standard” sizes for efficiency reason.
- The total size of the data area, in bytes. (This will generally be the same for all buffers, but it is best not to
  depend on that fact.)
- A pointer to the beginning of the actual (“in use”) data in the buffer. This may be the same as the beginning of
  the data area, or it may differ, for example if a header has been stripped from an incoming packet.
- A count of the data bytes actually stored in the buffer. Data stored in a buffer must be contiguous.
- Two pointers to other buffer structures. These can be used to implement two kinds of chains. The first is a
  chain of buffers that logically make up a single unit—typically a packet. If a packet is too big to fit in a single
  BUF, multiple BUFs can be chained together to hold the packet’s information. This also provides a convenient,
  if perhaps wasteful, way to add a header to a packet: simply prepend a BUF containing (only) the header to the
  chain of BUFs containing the rest of the packet.
  The second kind of chain is a list of complete packets (each of which is a contained in a chain of BUFs).
- Space for auxiliary layer-specific information. Such information is sometimes useful for communication be-
tween layers. For example, incoming packets to the network layer may have associated auxiliary information
identifying the interface on which they arrived; this information is handy to detect when packets are being
forwarded out the same interface they came in on. As another example, it is nice to have the total number of
bytes in a multi-BUF packet stored in the first BUF of the chain. The auxiliary information is opaque to the BUF
subsystem—that is, the BUF routines never read or write it.
- A reference (known as a “callback”) to a deallocation method. When a layer or module is finished processing a
  buffer, it calls this function (giving a reference to the BUF as argument) to deallocate the buffer. By overriding
  (replacing) this reference with a pointer to its own method, a layer can arrange to be informed when a BUF that
  it originally allocated is freed by another layer—for example, so it can also free any auxiliary data structure
  associated with the BUF. When created, BUFs refer to a default deallocation method, which should ultimately
  be called by any layer-provided deallocation method.

Figure 1 illustrates the layout of data in the data area; this scheme makes it easy to trim data from the front or back
of the message: just increase the pointer to the beginning of the valid data, or decrease the size of the valid data,
respectively.

Figure 2 illustrates the two kinds of chains. The first packet in the list (labeled “Packet 0”) is contained in three BUFs;
Packet 1 is made up of a single BUF, and Packet 2 is made up of two BUFs.

You will be given C code (files buf.h and buf.c) that will allow you to allocate and deallocate BUFs. In either case
you may want to add auxiliary methods to those provided — but you must do this in your own code.
The code will contain a constant `STDBUFSIZE`, which is the default size of the data area of a newly created/allocated `BUF`. You should not make any assumptions about the value of `STDBUFSIZE`. Anything in your code that depends on the value of `STDBUFSIZE` must use the symbolic constant. (In fact, your code should contain no numeric constants, except maybe 0 or 1.)