Lab session 4

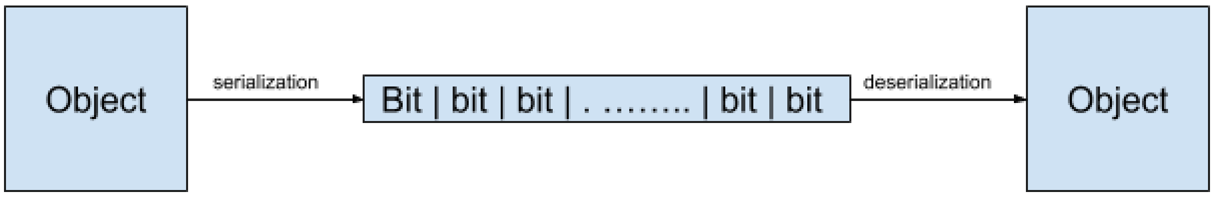
Data serialization

To transmit objects among several instances of a piece of software distributed over the network (e.g. in a multiplayer video game), the program has to format the data it needs to transmit in a proper format before sending packets through a network layer protocol (e.g. UDP or TCP).

This document describes a robust and easy to use data serialization system for the real time transmission of data through the network.

Data serialization

Serializing data means converting the data as it exists in main memory into a linear series of bits. Imagine a game, or a game editor running, with plenty of instances of game objects such as characters, buildings, animated objects, lights, assets, etc. We need a way to put all this data together so that it can be sent through the network or saved into a file.



Why is serialization important?

Picture the following class:

```
class Spaceship : public GameObject
{
  ...
private:
  int32_t mHealth;
  int32_t mPower;
};
```

As seen in previous sessions, when working with sockets (e.g. TCP sockets), we have available a couple of functions to send and receive data to/from the network (*send* and *recv*, respectively). An instance of the class *Spaceship* could be sent using these functions:

```
void NaivelySendSpaceship(SOCKET sock, const Spaceship *spaceship) {
  send(sock, reinterpret_cast<const char*>(spaceship), sizeof(Spaceship), 0);
}

void NaivelyRecvSpaceship(SOCKET sock, SpaceShip *spaceship) {
  recv(sock, reinterpret_cast<char*>(spaceship), sizeof(Spaceship), 0);
}
```

Let's say we have two hosts communicating through a connected socket at both ends. To send an instance of *Spaceship*, In one of the two hosts, the function *NaivelySendSpaceship* was called. The other host called *NaivelyRecvSpaceship* in order to receive it. This way, the spaceship instance will be transferred from one host to the other. This is of course, assuming that both machines have the same kind of hardware so they interpret the basic data types equally.

As *mHealth* and *mPower* are primitive data types, these functions behave as expected, and the instance of the spaceship in the destination host has the appropriate values after having been received.

However, if we need to transmit more complex objects (as it is often the case), the previous code is barely useful. Let us see the following implementation of the *Spaceship* class:

```
class Spaceship : public GameObject
{
  ...
private:
  int32_t mHealth;
  int32_t mPower;
  GameObject *mHomeBase;
  std::vector<LaserShots> mLaserShots;
};
```

The Spaceship class is now far more complex. If we directly execute the naive functions to send and receive data shown before, it is not likely that instances of the class *Spaceship* are going to be properly transferred. The previous functions copy the memory of the passed objects directly into the send buffer. Let's focus on the attribute *mHomeBase,* which is a pointer to a *GameObject*. After receiving it in the remote host, it will be probably pointing to an invalid region of memory. Something similar would happen with the attribute *mLaserShots,* which is a *std::vector* of *LaserShots*. We don't know its internal implementation, which may probably contain pointers to dynamic memory. It is almost for sure that *mLaserShots* will contain erroneous data after receiving it into the remote host.

Conclusion

So far, we have seen the main reasons why directly sending objects' memory is not a good idea. For a correct transmission of potentially complex objects, we need an individual serialization of its internal primitive-type attributes. To ease the serialization process, the concept of *streams* is introduced in the next section.

Streams

In computer programming, a *stream* is a data structure that encapsulates an ordered set of elements that allows programmers either reading or writing those elements in sequence.

Streams can either be intended to write data into them (*output streams*), or to read data from them (*input streams*). There can also exist *dual streams* that perform both operations. *Output streams* can be used to write or insert elements in sequence into an internal buffer, but never to read them. On the other hand, *input streams* are used to read elements in sequence from an internal buffer, not for writing.

Memory streams

A *memory stream* encapsulates a buffer in memory (usually a buffer allocated with dynamic memory) and provides methods to easily write or read elements sequentially.

OutputMemoryStream (output streams)

An *output memory stream* has methods to write data sequentially into its internal buffer, thus facilitating the serialization of objects. Furthermore, it provides a method to access its internal buffer pointer and the number of bytes (size) written so far. Those methods are useful whenever we need to share this memory with other parts of the application (such as passing this data to other functions). For instance, the data written into the *stream* could be passed to the function send, to send this data through the network.

```
class OutputMemoryStream
{
public:
  // Constructor and destructor
  OutputMemoryStream(uint32_t capacity);
  ~OutputMemoryStream();

  // Get pointer to the data in the stream
  const char *GetBufferPtr() const { return mBuffer; }
  uint32_t GetCapacity() const { return mCapacity; }
  uint32_t GetSize() const { return mHead; }

  // Clear the stream state
  void Clear() { mHead = 0; }

  // Generic Write method
  void Write(const void *inData, size_t inByteCount);
  // Write methods for primitive types
  void Write(bool elem);
  void Write(char elem);
  void Write(int elem);
  void Write(float elem);
  // others ...
  // Write methods for complex types
```

```
    void Write(const std::string& s);
    // others ...

  private:
    // Resize the buffer
    // (called from Write methods, when there is no space enough)
    void ReallocBuffer(uint32_t inNewLength);

    char *mBuffer; /* Dynamic buffer to store elements */
    uint32_t mCapacity; /* Size of the dynamic buffer */
    uint32_t mHead; /* Position of the next element to write */
  };
```

As you can see, the class *OutputMemoryStream* encapsulates a buffer (*mBuffer*) that will contain the data written into it. The attribute *mCapacity* contains the size (in bytes) of this buffer, and the attribute *mHead*, will contain the position of the next byte to be written (or what is the same, the number of bytes written so far). In the beginning, *mHead* will equal 0, and it will increment with each call to *Write*. How much it increments depends on the size of the written data (that is, *sizeof(char)*, *sizeof(int)*, *sizeof(float)*, etc).

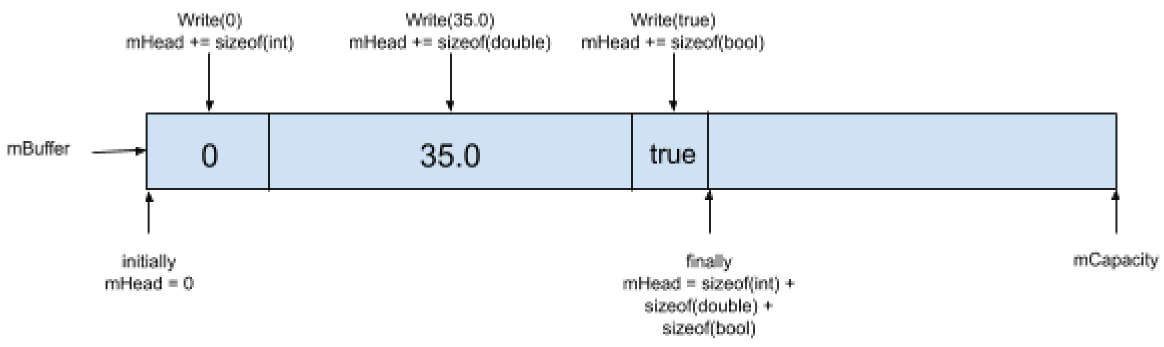
The following method *Write* is for integer numbers, and it provides an example on how to use the generic *Write* function:

```
  void Write(uint16_t val) {
    Write(&val, sizeof(val)); // Using the generic Write function
  }
```

The generic *Write* method (see the class definition above), before inserting any element into the stream, checks whether or not there is enough space for its insertion, and if not, it calls *ReallocBuffer* with a new buffer size as a parameter (for instance, twice the current size). The new buffer size needs to ensure that the data to insert will fit. This generic *Write* method, is the one responsible for incrementing the attribute *mHead*.

The function *ReallocBuffer*, as its name indicates, reallocates the buffer, usually to make it larger, and it must ensure that the previously written data is conserved.

The following picture shows how *mHead* is modified as several elements are written into the *stream*:



Using an *OutputMemoryStream*, we can serialize and send an object like a *Spaceship* in the previous example as follows:

```
// Serialize a spaceship
void Spaceship::Write(OutputMemoryStream &stream) const {
  stream.Write(mHealth);
  stream.Write(mPower);
  mHomeBase->Write(stream);
  stream.Write(mLaserShots.size());
  for (auto laserShot : mLaserShots) {
        laserShot.Write(stream);
  }
}

// Send a spaceship
void SendSpaceship(SOCKET socket, const Spaceship *spaceship) {
  OutputMemoryStream stream;
  spaceship->Write(stream);
  send(socket, stream.GetBufferPtr(), stream.GetSize(), 0);
}
```

InputMemoryStream (input streams)

To receive objects in the remote hosts, we may use the corresponding input stream:

```
class InputMemoryStream
{
public:
  // Constructor and destructor
  InputMemoryStream(uint32_t inSize = DEFAULT_STREAM_SIZE) :
    mBuffer(static_cast<char*>(std::malloc(inSize))), mCapacity(inSize), mHead(0)
  { }
  ~InputMemoryStream() { std::free(mBuffer); }

  // Get pointer to the data in the stream
  char *GetBufferPtr() const { return mBuffer; }
  uint32_t GetCapacity() const { return mCapacity; }

  // Clear the stream state
  void Clear() { mHead = 0; }

  // Generic Read method
  void Read(void *outData, size_t inByteCount);

  // Read methods for primitive types
  void Read(bool &elem);
  void Read(char &elem);
  void Read(int &elem);
  void Read(float &elem);
  // others ...

  // Read methods for complex types
  void Read(std::string& inString);
  // others ...
private:
  char *mBuffer; /* Dynamic buffer with stored elements */
  uint32_t mCapacity; /* Size of the dynamic buffer */
  uint32_t mHead; /* Position of the next element to read */
};
```

In this case, we do not need a method *ReallocBuffer*, as the contents of the buffer are fixed from the beginning and will not change. This stream is only in charge of iterating sequentially over the internal buffer to extract the elements it contains.

Analogously to the *output stream* in the previous section, the *input stream* increments the value of *mHead* with each call to the generic *Read* method, so that it advances as elements are read from it. The specific data type *Read* methods, on the other hand, are in charge of calling the generic *Read* method with the pointer to the element where the result of the read operation must be stored, and the size of the data (in bytes) to be read (*sizeof(elem)*).

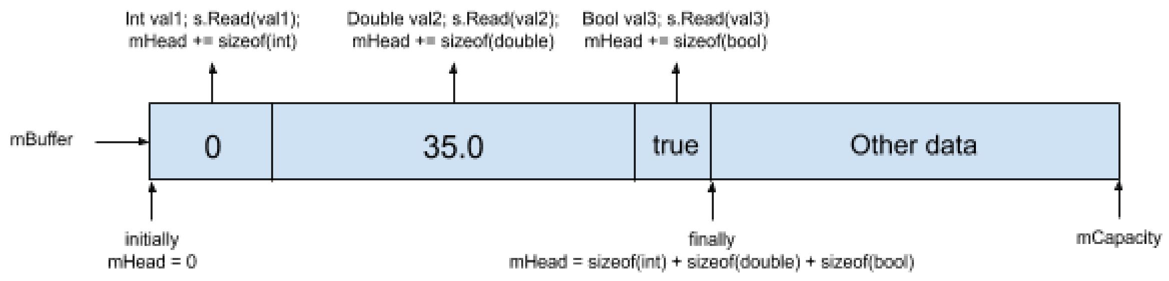
Using an *InputMemoryStream*, we can receive and deserialize objects like the *Spaceship* seen before as follows:

```
// Deserialize a spaceship
void Spaceship::Read(InputMemoryStream &stream) const {
  stream.Read(mHealth);
  stream.Read(mPower);
  mHomeBase->Read(stream);
  size_t vectorSize;
  stream.Read(vectorSize);
  mLaserShots.resize(vectorSize);
  for (auto laserShot : mLaserShots) {
         laserShot.Read(stream);
  }
}

// Receive a spaceship
void ReceiveSpaceship(SOCKET socket, Spaceship *spaceship) {
  InputMemoryStream stream(MAX_PACKET_LENGTH);
  size_t recvBytes = recv(socket, stream.GetBufferPtr(), stream.GetCapacity(), 0);
  if (recvBytes > 0) {
         spaceship->Read(stream);
  } else {
         // Handle error or disconnection
  }
}
```

Before receiving data with the function recv, an *InputMemoryStream* was declared with enough capacity (*MAX_PACKET_LENGTH*) to ensure that the received data will fit the stream.

The following picture shows how *mHead* is modified as elements are read from the *stream*:



IMPORTANT NOTE: We need to make sure that all reading operations (deserialization) are performed **exactly in the same order and using the same data types** as they were done during the corresponding writing operation (serialization).