Reading and writing raw data from/to binary files

Some of the data programs work with has to be persisted to disk files in various ways, that can include storing it in a database or to flat files, either as text or binary data. This recipe and the next one are focused on persisting and loading both raw data and objects from and to binary files. In this context, raw data means unstructured data, and in this recipe, we will consider writing and reading the content of a buffer (that is, a contiguous sequence of memory, that can be either a C-like array, an `std::vector`, or an `std::array`).

Getting ready

For this recipe, you should be familiar with the standard stream input/output library, though some explanations, to the extent required to understand this recipe, are provided below. You should also be familiar with the difference between binary and text files.

In this recipe, we will use the `ofstream` and `ifstream` classes, available in the namespace `std` in the `<fstream>` header.

In the following examples, we will consider the following data to write to a binary file (and consequently to read back):

```
    std::vector<unsigned char> output {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
```

How to do it...

To write the content of a buffer (in our example, an `std::vector`) to a binary file, you should perform the following steps:

1. Open a file stream for writing in binary mode by creating an instance of the `std::ofstream` class:

```
std::ofstream ofile("sample.bin", std::ios::binary);
```

2. Ensure that the file is actually open before writing data to the file:

```
if(ofile.is_open())
{
  // streamed file operations
}
```

3. Write the data to the file by providing a pointer to the array of characters and the number of characters to write:

```
ofile.write(reinterpret_cast<char*>(output.data()),
            output.size());
```

4. Flush the content of the stream buffer to the actual disk file; this is automatically done when you close the stream:

```
ofile.close();
```

To read the entire content of a binary file to a buffer, you should perform the following steps:

1. Open a file stream to read from a file in the binary mode by creating an instance of the `std::ifstream` class:

```
std::ifstream ifile("sample.bin", std::ios::binary);
```

2. Ensure that the file is actually opened before reading data from it:

```
if(ifile.is_open())
{
  // streamed file operations
}
```

3. Determine the length of the file by positioning the input position indicator to the end of the file, read its value, and then move the indicator to the beginning:

```
ifile.seekg(0, std::ios_base::end);
auto length = ifile.tellg();
ifile.seekg(0, std::ios_base::beg);
```

4. Allocate memory to read the content of the file:

```
std::vector<unsigned char> input;
input.resize(static_cast<size_t>(length));
```

5. Read the content of the file to the allocated buffer by providing a pointer to the array of characters for receiving the data and the number of characters to read:

```
ifile.read(reinterpret_cast<char*>(input.data()), length);
```

6. Check that the read operation is completed successfully:

```
auto success = !ifile.fail() && length == ifile.gcount();
```

7. Finally, close the file stream:

```
ifile.close();
```

How it works...

The standard stream-based input/output library provides various classes that implement high-level input, output, or both input and output file stream, string stream and character array operations, manipulators that control how these streams behave, and several predefined stream objects (`cin`/`wcin`, `cout`/`wcout`, `cerr`/`wcerr`, and `clog`/`wclog`).

These streams are implemented as class templates and, for files, the library provides several classes:

- `basic_filebuf` implements the input/output operations for a raw file and is similar in semantics with a C `FILE` stream.
- `basic_ifstream` implements the high-level file stream input operations defined by the `basic_istream` stream interface, internally using a `basic_filebuf` object.
- `basic_ofstream` implements the high-level file stream output operations defined by the `basic_ostream` stream interface, internally using a `basic_filebuf` object.
- `basic_fstream` implements the high-level file stream input and output operations defined by the `basic_iostream` stream interface, internally using a `basic_filebuf` object.

Several typedefs for the class templates mentioned in the preceding classes are also defined in the `<fstream>` header. The `ofstream` and `ifstream` objects are the type synonyms used in the preceding examples:

```
typedef basic_ifstream<char>    ifstream;
typedef basic_ifstream<wchar_t> wifstream;
typedef basic_ofstream<char>    ofstream;
typedef basic_ofstream<wchar_t> wofstream;
typedef basic_fstream<char>     fstream;
typedef basic_fstream<wchar_t>  wfstream;
```

In the previous section, we saw how we can write and read raw data to and from a file stream. The way that works is explained in more detail here:

- To write data to a file, we instantiated an object of the type `std::ofstream`. In the constructor, we passed the name of the file to be opened and the stream open mode, for which we specified `std::ios::binary` to indicate binary mode.

Opening the file like this discards the previous file content. If you want to append content to an existing file, you should also use the flag `std::ios::app` (that is, `std::ios::app | std::ios::binary`). This constructor internally calls `open()` on its underlying raw file object, that is, a `basic_filebuf` object. If this operation fails, a fail bit is set. To check whether the stream has been successfully associated with a file device, we used `is_open()` (that internally calls the method with the same name from the underlying `basic_filebuf`). Writing data to the file stream is done with the `write()` method that takes a pointer to the string of characters to write and the number of characters to write. Since this method operates with strings of characters, a `reinterpret_cast` is necessary if data is of another type, such as `unsigned char` in our example. The write operation does not set a fail bit on failure, but may throw an `std::ios_base::failure` exception. However, data is not written directly to the file device, but stored in the `basic_filebuf` object. To write it to the file, the buffer needs to be flushed, which is done by calling `flush()`. This is done automatically when closing the file stream, as in the preceding example.

- To read data from a file, we instantiated an object of time `std::ifstream`. In the constructor, we passed the same arguments we used for opening the file for writing, the name of the file and the open mode, that is, `std::ios::binary`. The constructor internally calls `open()` on the underlying `std::basic_filebuf` object. To check whether the stream has been successfully associated with a file device, we used `is_open()` (that internally calls the method with the same name from the underlying `basic_filebuf`). In this example, we read the entire content of the file to a memory buffer, in particular, an `std::vector`. Before we can read the data, we must know the size of the file in order to allocate a buffer large enough to hold that data. To do so, we used `seekg()` to move the input position indicator to the end of the file, then we called `tellg()` to return the current position, which in this case indicates the size of the file, in bytes, and then we moved the input position indicator to the beginning of the file to be able to start reading from the beginning. Calling `seekg()` to move the position indicator to the end can be avoided by opening the file with the position indicator moved directly to the end. This can be achieved using the `std::ios::ate` opening flag in the constructor (or the `open()` method). After allocating enough memory for the content of the file, we copied the data from the file into memory using the `read()` method. This takes a pointer to the string of characters that receives the data read from the stream and the number of characters to be read. Since the stream operates on characters, a

`reinterpret_cast` expression is necessary if the buffer contains other types of data, such as `unsigned char` in our example. This operation throws an `std::basic_ios::failure` exception if an error occurs. To determine the number of characters that have been successfully read from the stream, we can use the `gcount()` method. Upon completing the read operation, we close the file stream.

The operations shown in these examples are the minimal ones necessary to write and read data to and from file streams. It is important, though, that you perform appropriate checks for the success of the operations and catch possible exceptions that could occur.

The example code discussed so far in this recipe can be reorganized in the form of two general functions for writing and reading data to and from a file:

```
bool write_data(char const * const filename,
                char const * const data,
                size_t const size)
{
  auto success = false;
  std::ofstream ofile(filename, std::ios::binary);

  if(ofile.is_open())
  {
    try
    {
      ofile.write(data, size);
       success = true;
    }
    catch(std::ios_base::failure &)
    {
      // handle the error
    }
    ofile.close();
  }

  return success;
}

size_t read_data(char const * const filename,
                 std::function<char*(size_t const)> allocator)
{
  size_t readbytes = 0;
  std::ifstream ifile(filename, std::ios::ate | std::ios::binary);

  if(ifile.is_open())
  {
    auto length = static_cast<size_t>(ifile.tellg());
    ifile.seekg(0, std::ios_base::beg);

    auto buffer = allocator(length);

    try
```

```
        {
          ifile.read(buffer, length);

          readbytes = static_cast<size_t>(ifile.gcount());
        }
        catch (std::ios_base::failure &)
        {
          // handle the error
        }

        ifile.close();
      }

      return readbytes;
    }
```

`write_data()` is a function that takes the name of a file and a pointer to an array of character and its length and writes it to the specified file. `read_data()` is a function that takes the name of a file and a function that allocates a buffer and reads the entire content of the file to the buffer returned by the allocated function. The following is an example of how these functions can be used:

```
    std::vector<unsigned char> output {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
    std::vector<unsigned char> input;

    if(write_data("sample.bin",
                  reinterpret_cast<char*>(output.data()),
                  output.size()))
    {
      if(read_data("sample.bin",
                   [&input](size_t const length) {
        input.resize(length);
        return reinterpret_cast<char*>(input.data());}) > 0)
        {
          std::cout << (output == input ? "equal": "not equal")
                    << std::endl;
        }
      }
```

Alternatively, we could use a dynamically allocated buffer, instead of the `std::vector`; the changes required for that are small in the overall example:

```
    std::vector<unsigned char> output {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
    unsigned char* input = nullptr;
    size_t readb = 0;

    if(write_data("sample.bin",
                  reinterpret_cast<char*>(output.data()),
                  output.size()))
    {
      if((readb = read_data(
         "sample.bin",
         [&input](size_t const length) {
        input = new unsigned char[length];
        return reinterpret_cast<char*>(input); })) > 0)
        {
          auto cmp = memcmp(output.data(), input, output.size());
```

```
          std::cout << (cmp == 0 ? "equal": "not equal")
                    << std::endl;
        }
      }

      delete [] input;
```

There's more...

The way of reading data from a file to memory shown in this recipe is only one of the several possible alternatives. Compared to the others, it is, however, the fastest method, even though the alternatives may look more appealing from an object-oriented perspective. It is beyond the purpose of this recipe to compare the performance of these alternatives, but the reader can take it as an exercise.

The following are possible alternatives for reading data from a file stream:

- Initializing an `std::vector` directly using `std::istreambuf_iterator` iterators (similarly, this can be used with `std::string`):

```
std::vector<unsigned char> input;
std::ifstream ifile("sample.bin", std::ios::binary);
if(ifile.is_open())
{
  input = std::vector<unsigned char>(
     std::istreambuf_iterator<char>(ifile),
     std::istreambuf_iterator<char>());
  ifile.close();
}
```

- Assigning the content of an `std::vector` from `std::istreambuf_iterator` iterators:

```
std::vector<unsigned char> input;
std::ifstream ifile("sample.bin", std::ios::binary);
if(ifile.is_open())
{
  ifile.seekg(0, std::ios_base::end);
  auto length = ifile.tellg();
  ifile.seekg(0, std::ios_base::beg);

  input.reserve(static_cast<size_t>(length));
  input.assign(
    std::istreambuf_iterator<char>(ifile),
    std::istreambuf_iterator<char>());
  ifile.close();
}
```

- Copying the content of the file stream to a vector using `std::istreambuf_iterator` iterators and an `std::back_inserter` adapter to write to the end of the vector:

```
std::vector<unsigned char> input;
std::ifstream ifile("sample.bin", std::ios::binary);
if(ifile.is_open())
{
  ifile.seekg(0, std::ios_base::end);
```

```
  auto length = ifile.tellg();
  ifile.seekg(0, std::ios_base::beg);

  input.reserve(static_cast<size_t>(length));
  std::copy(std::istreambuf_iterator<char>(ifile),
            std::istreambuf_iterator<char>(),
            std::back_inserter(input));
  ifile.close();
}
```

See also

- *Reading and writing objects from/to binary files*
- *Using I/O manipulators to control the output of a stream*

Reading and writing objects from/to binary files

In the previous recipe, we saw how to write and read raw data (that is, unstructured data) to and from a file. Many times, however, we have to persist and load objects. Writing and reading in the manner shown in the previous recipe works for POD types only. For anything else, we must explicitly decide what is actually written or read, as writing or reading pointers, virtual tables, and any sort of meta data is not only irrelevant, but also semantically wrong. These operations are commonly referred to as serialization and deserialization. In this recipe, we will see how we can serialize and deserialize both POD and non-POD types to and from binary files.

Getting ready

It is recommended that you first read the previous recipe, *Reading and writing raw data from/to binary files*, before you continue. You should also know what POD and non-POD types are and how operators can be overloaded.

For the examples in this recipe, we will use the `foo` and `foopod` classes shown in the following:

```
class foo
{
  int i;
  char c;
  std::string s;

public:
  foo(int const i = 0, char const c = 0, std::string const & s = {}):
    i(i), c(c), s(s)
  {}

  foo(foo const &) = default;
  foo& operator=(foo const &) = default;

  bool operator==(foo const & rhv) const
  {
    return i == rhv.i &&
           c == rhv.c &&
           s == rhv.s;
  }

     bool operator!=(foo const & rhv) const
  {
    return !(*this == rhv);
  }
};

struct foopod
{
  bool a;
  char b;
  int c[2];
};
```

```


bool operator==(foopod const & f1, foopod const & f2)
{
  return f1.a == f2.a && f1.b == f2.b &&
         f1.c[0] == f2.c[0] && f1.c[1] == f2.c[1];
}
```

How to do it...

To serialize/deserialize POD types that do not contain pointers, use `ofstream::write()` and `ifstream::read()`, as shown in the previous recipe:

- Serialize objects to a binary file using `ofstream` and the `write()` method:

```
std::vector<foopod> output {
  {true, '1', {1, 2}},
  {true, '2', {3, 4}},
  {false, '3', {4, 5}}
};

std::ofstream ofile("sample.bin", std::ios::binary);
if(ofile.is_open())
{
  for(auto const & value : output)
  {
    ofile.write(reinterpret_cast<const char*>(&value),
                sizeof(value));
  }

  ofile.close();
}
```

- Deserialize objects from a binary file using the `ifstream` and `read()` methods:

```
std::vector<foopod> input;
std::ifstream ifile("sample.bin", std::ios::binary);
if(ifile.is_open())
{
  while(true)
  {
    foopod value;
    ifile.read(reinterpret_cast<char*>(&value),
               sizeof(value));

    if(ifile.fail() || ifile.eof()) break;
      input.push_back(value);
  }

  ifile.close();
}
```

To serialize non-POD types (or POD types that contain pointers), you must explicitly write the value of data members to a file, and to deserialize, you must explicitly read from the file to the data members in the same order. To exemplify this, we will consider the `foo` class defined earlier:

- Add a member function called `write()` to serialize objects of this class. The

method takes a reference to an `ofstream` and returns a `bool` indicating whether the operation was successful or not:

```
bool write(std::ofstream& ofile) const
{
  ofile.write(reinterpret_cast<const char*>(&i), sizeof(i));
  ofile.write(&c, sizeof(c));
  auto size = static_cast<int>(s.size());
  ofile.write(reinterpret_cast<char*>(&size), sizeof(size));
  ofile.write(s.data(), s.size());

  return !ofile.fail();
}
```

- Add a member function called `read()` to deserialize objects of this class. This method takes a reference to an `ifstream` and returns a `bool` indicating whether the operation was successful or not:

```
bool read(std::ifstream& ifile)
{
  ifile.read(reinterpret_cast<char*>(&i), sizeof(i));
  ifile.read(&c, sizeof(c));
  auto size {0};
  ifile.read(reinterpret_cast<char*>(&size), sizeof(size));
  s.resize(size);
  ifile.read(reinterpret_cast<char*>(&s.front()), size);

  return !ifile.fail();
}
```

An alternative to the `write()` and `read()` member functions exemplified above is to overload `operator<<` and `operator>>`. To do this, you should perform the following steps:

1. Add `friend` declarations for non-member `operator<<` and `operator>>` to the class to be serialized/deserialized (in this case, the `foo` class):

```
friend std::ofstream& operator<<(std::ofstream& ofile,
                                 foo const& f);
friend std::ifstream& operator>>(std::ifstream& ifile,
                                 foo& f);
```

2. Overload `operator<<` for your class:

```
std::ofstream& operator<<(std::ofstream& ofile, foo const& f)
{
  ofile.write(reinterpret_cast<const char*>(&f.i),
              sizeof(f.i));
  ofile.write(&f.c, sizeof(f.c));
  auto size = static_cast<int>(f.s.size());
  ofile.write(reinterpret_cast<char*>(&size), sizeof(size));
  ofile.write(f.s.data(), f.s.size());
```

```
  return ofile;
}
```

3. Overload `operator>>` for your class:

```
std::ifstream& operator>>(std::ifstream& ifile, foo& f)
{
  ifile.read(reinterpret_cast<char*>(&f.i), sizeof(f.i));
  ifile.read(&f.c, sizeof(f.c));
  auto size {0};
  ifile.read(reinterpret_cast<char*>(&size), sizeof(size));
  f.s.resize(size);
  ifile.read(reinterpret_cast<char*>(&f.s.front()), size);

  return ifile;
}
```

How it works...

Regardless of whether we serialize the entire object (for POD types) or only parts of it, we use the same stream classes discussed in the previous recipe, `ofstream` for output file streams and `ifstream` for input file streams. Details about writing and reading data using these standard classes have been discussed in that recipe and will not be reiterated here.

When you serialize and deserialize objects to and from files, you should avoid writing values of pointers to a file, and you must not read pointer values from the file since these represent memory addresses and are meaningless across processes, or even in the same process some moments later. Instead, you should write data referred by a pointer and read data into objects referred by a pointer. This is a general principle, and in practice, you may encounter situations where a source may have multiple pointers to the same object, in which case you might want to write only one copy and also handle the reading in a corresponding manner.

If the objects you want to serialize are of a POD type, you can do it just like we did when we discussed raw data. In the example in this recipe, we serialized a sequence of objects of the `foopod` type. When we deserialize, we read from the file stream in a loop until the end of the file is read or a failure occurs. The way reading is done in this case may look counter-intuitive, but doing it differently may lead to duplication of the last read value:

- Reading is done in an infinite loop.
- A read operation is performed in the loop.
- A check for failure or end of file is performed, and if any of these occurred, the infinite loop is exited.
- The value is added to the input sequence and the looping continues.

If reading is done using a loop with an exit condition that checks the end of the file bit, that is, `while(!ifile.eof())`, the last value will be added twice to the input sequence. The reason for that, is that upon reading the last value, the end of file is not yet encountered (as that is a mark beyond the last byte of the file). The end of

file mark is only reached at the next read attempt, which, therefore, sets the of bit of the stream. However, the input variable still has the last value, as it hasn't been overwritten with anything, and this is added for a second time to the input vector.

If the objects you want to serialize and deserialize are of non-POD types, writing/reading these objects as raw data is not possible. For instance, such an object may have a virtual table. Writing the vtable to a file does not cause problems, even though it does not have any value; however, reading from a file, and, therefore, overwriting the vtable of an object will have catastrophic effects on the object and the program.

When serializing/deserializing non-POD types, there are various alternatives, some of them shown in the previous section: either provide explicit methods for writing and reading or overloading the standard << and >> operators. The second approach has the advantage that it enables the use of your class in generic code where objects are written and read to and from stream files using these operators.

When you plan to serialize and deserialize your objects, consider versioning your data from the very beginning to avoid problems if the structure of your data changes over time. How versioning should be done is beyond the purpose of this recipe.

See also

- *Reading and writing raw data from/to binary files*
- *Using I/O manipulators to control the output of a stream*