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*