2 I/O in C++¶

- **You can download the template for the programming tasks of the module as a `zip` file from this link.**

Contents

The past examples have used the standard input and output streams, **cin** and **cout**, that by default read and write to the terminal console. cin is a readily created object of `istream` type, stream is the class that implements input streams. cout is an object of `ostream` type, which is for output streams. In addition, there is a class `iostream` that is for both input and output. This class inherits both `istream` and `ostream` (this is called multiple inheritance, which we haven't discussed yet).

2.1 File streams¶

C++ Primer Chapter 8.2 (File input and output)

File streams is a stream that is intended for reading or writing to a file. The file streams are `ifstream`, `ofstream` and `fstream`; for input, output or both, respectively. These inherit from `istream`, `ostream` or `iostream`. The file stream classes are defined in a header file `fstream`, so you should include **fstream** header in order to use them.

Because file streams inherit from basic I/O streams, they can use the same operations as the normal streams, including the stream operators `<<` and `>>`. If you have overloaded these operators for your class, you can use them also with files. They also work normally with the built-in data types and standard library types.

Below is a modification to the GeomVector class in the previous module. We modify the main function such that it writes the output to a file.

```
1#include <iostream>
2#include <fstream>
3
4class GeomVector {
5public:
6   GeomVector(double x, double y, double z) : x_{x},y_{y},z_{z} { }
7
8   // this is member function of GeomVector
9   GeomVector operator+(const GeomVector& a);
10
11  // these are external functions, but have access to private parts
12  friend GeomVector operator*(double a, const GeomVector& b);
13  friend std::ostream &operator<<(std::ostream &out,
14                                  const GeomVector& a);
15
16private:
17   double x_,y_,z_;
18};
19
20GeomVector GeomVector::operator+(const GeomVector& a)
21{
22   return GeomVector(a.x_ + x_, a.y_ + y_, a.z_ + z_);
23}
24
25GeomVector operator*(double a, const GeomVector& b)
26{
27   return GeomVector(a * b.x_, a * b.y_, a * b.z_);
28}
29
30std::ostream &operator<<(std::ostream &out, const GeomVector& a) {
31   out << "[" << a.x_ << ", " << a.y_ << ", " << a.z_ << "]";
32   return out;
33}
34
35int main() {
36   GeomVector vec{2,3,4}, vec3{1,1,1}, vec{0.1,0,0};
37   GeomVector vec5{0,0,0}, vec6{0,0,0};
38
39   vec5 = vec4 + vec3;
40   vec6 = 5 * vec3;
41
42   // open a file for writing
43   std::ofstream oos("outfile");
44
45   // write to the file
46   os << "vec4: " << vec4 << std::endl
47   << "vec5: " << vec5 << std::endl
48   << "vec6: " << vec6 << std::endl;
49
50   // close the file
51   os.close();
52
53   // open the file for reading
54   std::ifstream ios("outfile");
55   // write the contents of the file to standard output
56   std::cout << ios.rdbuf();
57}
```

The above example opens file *outfile* (on line 9) for writing. The rest of the main function is very similar to the example in Module 3. The only difference is that we use the os object in place of std::cout when outputting data.

There is a close function for file streams, as with traditional C streams, but the destructor of a file stream automatically closes the file. Therefore, separately calling the close function is not usually needed, but in this case, as we reopen the same file, we need to close it first. By default, an ofstream object truncates (i.e., deletes the contents) any existing file of the same name. For different behaviour, we can specify different file modes as the second parameter of the *ofstream* constructor (i.e. it is overloaded, or rather has defaulted for the second argument). For example the following will not truncate the file, but append new content at the end of the file:

```
std::ofstream oos("outfile", std::ofstream::app)
```

More information about the different modes in the constructor documentation. Note that even though it says basic_ofstream it can be used with just ofstream like in the example on that page.

2.2 Condition bits and manipulators¶

The current I/O stream status bits can be queried using the rdstate function. The function returns an object of type *iostate* (defined in the stream's namespace), that can be handled like a bit mask. The following values are in use:

- **badbit**: the stream is corrupted
- **failbit**: an I/O operation has failed
- **eofbit**: stream is at the end of file
- **goodbit**: stream is ok. This value is equivalent to zero (useful to know in condition tests).

The following example shows the condition bits in use.

```
1#include <iostream>
2#include <fstream>
3#include <string>
4
5int main() {
6   std::string fname = "testfile";
7
8   std::ifstream istr(fname);
9   if (istr.rdstate() & (istr.failbit | istr.badbit)) {
10      // output error to stderr stream
11      std::cerr << "Failed" << std::endl;
12  } else {
13      // repeat until end of file
14      while (istr.rdstate() == 0) {
15          // read a line from file, output it to screen
16          std::string line;
17          std::getline(istr, line);
18          std::cout << line << std::endl;
19      }
20  }
21}
```

The comments in the above example hopefully explain the logic sufficiently, but it is worth noticing that the condition bits belong to the namespace of the stream, and are referred to accordingly (see line 9). They are not variables, but constant values. On line 17 there is the function getline that, on the other hand, is not member of the stream, but directly under *std* namespace. The stream to operate is given as the parameter of the function. There is also a getline variant for C-style strings, but it is more difficult to use one needs to allocate a sufficiently sized char buffer for reading.

The output can be affected using **manipulators**, that are mixed together with the actual output using the `<<` operator. We have already seen one manipulator, **endl** that changes the line of output. In addition, there are the following manipulators:

- **ends**: inserts a null character to the buffer
- **flush**: output all data currently in stream buffer. For example `std::cout << "Hey!" << std::flush;`
- **unitbuf**: changes the buffering behavior to output each write directly. For example `std::cout << std::unitbuf;`
- **nounitbuf**: reverts the buffering behavior back to normal system behavior.

2.3 String streams¶

C++ Primer Chapter 8.3 (Strings streams)

String streams (*stringstream*) are sometimes useful in when formatted output operations need to be applied to string object. When string stream is created, it can be used normally like any stream. All written content will be stored as a string. that can be used normally like any *[string]* object. String streams can be used for input, and a string stream can be initialized with a predefined string. Stringstreams are defined in **sstream** header.

Below we modify the main function of the previous GeomVector once again.

```
1#include <iostream>
2#include <sstream>
3
4class GeomVector {
5public:
6   GeomVector(double nx, double ny, double nz) : x{nx},y{ny},z{nz} { }
7
8   // this is member function of GeomVector
9   GeomVector operator+(const GeomVector& a);
10
11  // these are external functions, but have access to private parts
12  friend GeomVector operator*(double a, const GeomVector& b);
13  friend std::ostream &operator<<(std::ostream &out,
14                                  const GeomVector& a);
15
16private:
17   double x,y,z;
18};
19
20GeomVector GeomVector::operator+(const GeomVector& a)
21{
22   return GeomVector(a.x + x, a.y + y, a.z + z);
23}
24
25GeomVector operator*(double a, const GeomVector& b)
26{
27   return GeomVector(a * b.x, a * b.y, a * b.z);
28}
29
30std::ostream &operator<<(std::ostream &out, const GeomVector& a)
31{
32   out << "[" << a.x << ", " << a.y << ", " << a.z << "]";
33   return out;
34}
35
36int main() {
37   // create a few vectors and demonstrate their use
38   GeomVector vec2{2,4,6}, vec3{1,1,1}, vec{0.1,0,0};
39   GeomVector vec5{0,0,0}, vec6{0,0,0};
40
41   vec5 = vec4 + vec3;
42   vec6 = 5 * vec3;
43
44   // now, empty string stream
45   std::stringstream ss;
46
47   // write something to the stream
48   ss << "vec4: " << vec4 << std::endl
49   << "vec5: " << vec5 << std::endl
50   << "vec6: " << vec6 << std::endl;
51
52   // copy the string object from string stream
53   // print the length (48) and string itself
54   std::string str= ss.str();
55   std::cout << "length: " << str.length() << std::endl;
56   std::cout << "String: " << str << std::endl;
57}
```

There is an overloaded version of the *str* function that takes a string as a parameter, and sets the stream's contents to the given string. Then the content of the stream can be parsed with the stream input operator (`>>`).

2.4 Overloading the input operator¶

C++ Primer Chapter 14.2.2 (Overloading the Input Operator >>)

Overloading the input operator `>>` is useful when we want to create objects based on user input or read them from a file. Below is an example how the `GeomVector` class could be used together with the overloaded input operator. The example shows how the default constructor is used to create the `GeomVector` that is then filled with the input from a string stream.

The reading would fail, for example, if the string stream doesn't have enough numbers to fill the three vectors, which you can try out by modifying the example yourself. It is customary that the values are first read into some temporary variables, and only after checking that the reading succeeded, the values are stored in the object that the operator is supposed to modify.

```
1class GeomVector
2{
3public:
4   // these are external functions, but have access to private parts
5   friend std::istream &operator>>(std::istream &out, const GeomVector& a);
6   friend std::ostream &operator<<(std::ostream &ss, const GeomVector& a);
7
8private:
9   double x, y, z;
10};
11
12std::ostream &operator<<(std::ostream &out, const GeomVector& a)
13{
14   out << "[" << a.x << ", " << a.y << ", " << a.z << "]";
15   return out;
16}
17
18std::istream &operator>>(std::istream &is, GeomVector& a) {
19   double x, y, z;
20   is >> x >> y >> z;
21
22   // Check that reading succeeded
23   if (is) {
24      a.x = x;
25      a.y = y;
26      a.z = z;
27   } else {
28      // Set the stream instate to fail if reading failed
29      is.clear(is.rdstate() & ~std::ios::goodbit) ; std::ios::failbit);
30   }
31
32   return is;
33}
34
35int main() {
36   std::istringstream iss("2 3 4 1 3.45 5 0.1 0 2.8");
37
38   // Create a few vectors with the overloaded >> operator
39   GeomVector vec4, vec5, vec6;
40
41   if(iss >> vec4 >> vec5 >> vec6) {
42      std::cout << "Successfully read GeomVectors from istringstream"
43                << std::endl;
44   } else {
45      std::cout << "Failed to read GeomVectors!" << std::endl;
46      return 0;
47   }
48
49   // print the vectors to stdout
50   std::cout << "vec4: " << vec4 << std::endl
51             << "vec5: " << vec5 << std::endl
52             << "vec6: " << vec6 << std::endl;
53}
```

2.4.1 Programming tasks¶

Points 0/2 / 2 | My submissions 47 | Deadline Friday, 15 October 2021, 19:59 | To be submitted alone

This course has been archived (Saturday, 17 December 2022, 19:59).

Vehicle register

Objective: Practicing I/O together with class inheritance and polymorphism.

You will need to implement a vehicle register. You will also implement a way to serialize the instances stored in the register and output the serialization to a file / read serialized data from a file and store the instances in the register.

In the register there may be 2 different kinds of vehicles: boats and aircrafts. Every type of these 3 has their own specific variables:

Boat:

- name (string)
- draft (double)
- power (double)

Aircraft:

- model (string)
- wingspan (double)
- cruise speed (unsigned int)

These vehicle types also share common member variables, such as register number and owner. These are implemented already in the Vehicle class. So these 2 types of vehicle classes inherit from the Vehicle class.

You need to implement the *Register*, *Vehicle*, *Boat* and *Aircraft* classes. More information in the *.hpp* files.

Error

The documentation of the `Read` functions indicates that the objects should be constructed from a line that has the format specified in the `write` functions. Since the extraction operator of istream class `>>` extracts arguments separated by whitespace characters, you cannot use it in the `Read` function.

Hint

1. Check istream::getline reference.
2. See non-member function overloads in <string>.

aircraft.cpp

Choose File | No file chosen

boat.cpp

Choose File | No file chosen

register.cpp

Choose File | No file chosen

vehicle.cpp

Choose File | No file chosen

Submit

Points 0/2 / 2 | My submissions 14 | Deadline Friday, 15 October 2021, 19:59 | To be submitted alone

This course has been archived (Saturday, 17 December 2022, 19:59).

Polynomial

Objective: Learn to overload various operators.

Polynomial is a mathematical expression consisting of terms summed to each other. In this exercise you will implement class *Poly*, holding a single polynomial where each term is of form **mul*x^pow**. For example, the parabola $4x^2+2x+7$ would be stored in the Poly object in a *std::map<int, int>* of pairs of exponent and multiplier: {2, 4}, {1, 2}, {0, 7}.

The comments in file *poly.hpp* provide more detailed instructions of the functions you will need to implement. You will only need to modify *poly.cpp*.

poly.cpp

Choose File | No file chosen

Submit