Software Development

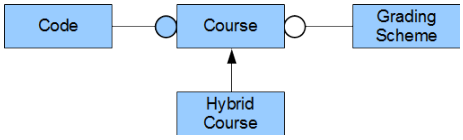
Part C - Encapsulation

Input and Output Operators

Use stream objects to interact with users and access persistent data
Introduce association relationships between classes
Overload the extraction and insertion operators as helper operators

"You don't need to modify istream or ostream to add new << and >> operators" (Stroustrup, 1997)

The relationships between classes that object-oriented languages support include compositions and associations. Both of these relationships are more loosely coupled than friendship. A *composition* is a relationship in which one class *has* another class, while an *association* is a relationship in which one class *uses* another class. In relationship diagrams, associations appear as open circles and compositions as filled circles. The diagram below shows that **`Code`** uses the **`Calendar`** (to determine availability), while each **`Course`** has a Course **`Code`** class.



Associations are more loosely coupled than compositions. Typical examples of associations are relationships between the stream-based input and output library classes that support the C++ language and our own custom classes. To create these relationships, we overload the insertion and extraction operators as helper operators that take stream objects as their left operands and objects of our class type as their right operands.

This chapter describes the stream-based input/output library and shows how to overload the insertion and extraction operators for objects of our custom classes. This chapter shows how to define file objects and use them to store and retrieve variables of fundamental type. The chapter concludes by introducing the standard library's **`string`** class, which manages character stream input of user-defined length.

STREAM LIBRARY OVERVIEW

The stream-based input/output library that supports the core C++ language overloads two operators for extracting values from an input stream and inserting values into an output stream:

- **`>>`** (extract from an input stream)
- **`<<`** (insert into an output stream)

The library embeds its class definitions in the standard namespace (**`std`**). To access those instances of these classes that the library predefines, we prefix their identifiers with the namespace identifier and the scope resolution operator (**`std::`**).

For example,

```
#include <iostream>

int main() {
    int x;
                                                Enter an integer : 3
    std::cout << "Enter an integer : ";
    std::cin >> x;                              You entered 3
    std::cout << "You entered " << x << std::endl;
}
```

Practice

Resources

Standard I/O and File I/O

Standard I/O

The **iostream** system header file contains the definitions for streaming from and to standard devices.

```
#include <iostream>
```

This header file includes the definitions of the

- **std::istream** class - for processing input from the standard input device
- **std::ostream** class - for processing output to the standard output devices

This header file also predefines the standard input and output objects:

- **std::istream**
  - **std::cin** - standard input
- **std::ostream**
  - **std::cout** - standard output
  - **std::cerr** - standard error
  - **std::clog** - standard log

We use these objects directly and don't need to redefine them.

File I/O

The **fstream** system header file contains the definitions for streaming from and to files.

```
#include <fstream>
```

This header file includes the definitions of the

- **std::ifstream** class - for processing input from a file stream
- **std::ofstream** class - processing output to a file stream
- **std::fstream** class - processing input from and output to a file stream

These three classes manage communication between file streams containing 8-bit characters and system memory. They provide access to a file stream through separate input and output buffers.

Design Considerations

In overloading the insertion and extraction operators for our class types, good design suggests:

- providing flexibility in the selection of stream objects
- resolving scope on classes and objects defined in header files
- enabling cascading as implemented for fundamental types

Selection of Stream Objects

To enable selection of the stream objects by the client code, we upgrade our **display()** member function to receive a modifiable reference to an object of **std::ostream** type. The changes to the header file are shown on the left and the

implementation file on the right:

```
// Student.h

#include <iostream> // for
                    // std::ostream
const int NG = 20;

class Student {
    int no;
    float grade[NG];
    int ng;
public:
    Student();
    Student(int);
    Student(int, const float*, int);
    void display(std::ostream&)
     const;
};
```

```
// Student.cpp

#include <cstring>
#include "Student.h"
using namespace std;

// ...

void Student::display(ostream&
 os) const {
    if (no > 0) {
        os << no << ":\n";
        os.setf(ios::fixed);
        os.precision(2);
        for (int i = 0; i < ng; i++) {
            os.width(6);
            os << grade[i] << endl;
        }
        os.unsetf(ios::fixed);
        os.precision(6);
    } else {
        os << "no data available"
         << endl;
    }
}
```

With this upgrade the client code can choose the destination stream (`cout`, `cerr`, or `clog`).

Header Files

A header files may be included alongside other header files written by other developers. To avoid conflicts between the header files included in an implementation file, we follow certain guidelines:

- include system header files before custom header files
- insert namespace directives after all header files
- resolve the scope of any identifier in a header file at the identifier itself

The preferred method of coding header files is shown on the right:

```
// Student.h

#include <iostream>
using namespace std; // POOR DESIGN
const int NG = 20;

class Student {
    int no;
    float grade[NG];
    int ng;
public:
    Student();
    Student(int);
    Student(int, const float*, int);
    void display(ostream& os) const;
};
```

```
// Student.h

#include <iostream>  // GOOD DESIGN
const int NG = 20;

class Student {
    int no;
    float grade[NG];
    int ng;
public:
    Student();
    Student(int);
    Student(int, const float*, int);
    void display(std::ostream& os)
     const;
};
```

Exposing all of the names in any namespace as on the left may lead to unnecessary conflicts with new names or conflicts when several header files are included in an implementation file. Resolving scope in the `display()` function's parameter list identifies the class used with its namespace, without exposing any name in that namespace.

Cascading

The following expression is a cascaded expression

```
std::cout << x << y << z << std::endl;
```

Cascading support enables concatenation of operations where the leftmost operand serves as the left operand for every operation in a compound expression.

The cascaded expression above expands to two simpler sub-expressions executed in the following order:

```
std::cout << x;
std::cout << y << z << std::endl;
```

The cascaded sub-expression

```
std::cout << y << z << std::endl;
```

expands to two simpler sub-expressions executed in the following order:

```
std::cout << y;
std::cout << z << std::endl;
```

Finally, the cascaded sub-expression

```
std::cout << z << std::endl;
```

expands into two simpler sub-expressions executed in the following order:

```
std::cout << z;
std::cout << std::endl;
```

Enabling cascading requires that we return a modifiable reference to the left operand.

Returning a modifiable reference from a function lets the client code use the return value as the left operand for the operator on its right.

STANDARD I/O OPERATORS

The prototypes for the overloaded insertion and extraction operators for standard input and output on objects of our own classes take the form

```
std::istream& operator>>(std::istream&, Type&);
std::ostream& operator<<(std::ostream&, const Type&);
```

where ***Type*** is the name of the class.

The header file for our **Student** class that includes their declarations is:

```
// Student.h

#include <iostream> // for std::ostream, std::istream
const int NG = 20;
```

```
class Student {
    int no;
    float grade[NG];
    int ng;
public:
    Student();
    Student(int);
    Student(int, const float*, int);
    void read(std::istream&);
    void display(std::ostream& os) const;
};

std::istream& operator>>(std::istream& is, Student& s);
std::ostream& operator<<(std::ostream& os, const Student& s);
```

The implementation file for our upgraded `Student` class contains:

```
// Student.cpp

#include "Student.h"
using namespace std;

Student::Student() {
    no = 0;
    ng = 0;
}

Student::Student(int n) {
    *this = Student(n, nullptr, 0);
}

Student::Student(int sn, const float* g, int ng_) {
    bool valid = sn > 0 && g != nullptr && ng_ >= 0;
    if (valid)
        for (int i = 0; i < ng_ && valid; i++)
            valid = g[i] >= 0.0f && g[i] <= 100.0f;

    if (valid) {
        // accept the client's data
        no = sn;
        ng = ng_ < NG ? ng_ : NG;
        for (int i = 0; i < ng; i++)
            grade[i] = g[i];
    } else {
        *this = Student();
    }
}

void Student::read(istream& is) {
    int no;          // will hold the student number
    int ng;          // will hold the number of grades
    float grade[NG]; // will hold the grades

    cout << "Student Number : ";
    is >> no;
    cout << "Number of Grades : ";
    is >> ng;
    if (ng > NG) ng = NG;
    for (int i = 0; i < ng; i++) {
        cout << "Grade " << i + 1 << " : ";
        is >> grade[i];
    }

    // construct a temporary Student
    Student temp(no, grade, ng);
```

```
    // if data is valid, the temporary object into the current object
    if (temp.no != 0)
        *this = temp;
}

void Student::display(ostream& os) const {
    if (no > 0) {
        os << no << ":\n";
        os.setf(ios::fixed);
        os.precision(2);
        for (int i = 0; i < ng; i++) {
            os.width(6);
            os << grade[i] << endl;
        }
        os.unsetf(ios::fixed);
        os.precision(6);
    } else {
        os << "no data available" << endl;
    }
}

std::ostream& operator<<(ostream& os, const Student& s) {
    s.display(os);
    return os;
}

std::istream& operator>>(istream& is, Student& s) {
    s.read(is);
    return is;
}
```

The following client code uses our upgraded **`Student`** class accepts the input shown on the right and produces the results shown below:

```
// Standard I/O Operators
// standardIO.cpp

#include "Student.h"

int main () {
    Student harry;

    std::cin >> harry;
    std::cout << harry;
}
```

```
Student Number : 1234
Number of Grades : 2
Grade 1 : 56.7
Grade 2 : 78.9

1234:
 56.70
 78.90
```

FILE I/O OPERATORS

The stream library does not predefine any file objects as instances of the file stream classes. To create file objects, we need to define them ourselves and connect them to a named file. A file object is an instance of one of the file stream classes. When used with the insertion or extraction operators on a connected file, a file object streams the data in formatted form.

File Connections

We can connect a file object to a file for reading, writing or both. The object's destructor closes the connection.

Input File Objects

To create a file object for reading we define an instance of the **`std::ifstream`** class. This class includes a no-argument constructor as well as one that receives the address of a C-style null-terminated string containing the file name.

For example,

```
// Create a File for Reading
// createFileReading.cpp

#include <fstream>

int main() {
    std::ifstream f("input.txt"); // connects fin to input.txt for reading
    // ...
}
```

To connect a file to an existing file object, we call the **`open()`** member function on the object.

For example,

```
// Connect to a File for Reading
// connectFileReading.cpp

#include <fstream>

int main() {
    std::ifstream fin;     // defines a file object named fin
    fin.open("input.txt"); // connects input.txt to fin
    // ...
}
```

Output File Objects

To create a file object for writing we define an instance of the **`std::ofstream`** class. This class includes a no-argument constructor as well as one that receives the address of a C-style null-terminated string containing the name of the file.

For example,

```
// Writing to a File
// writeFile.cpp

#include <fstream>

int main() {
    std::ofstream fout("output.txt"); // connects fout to output.txt for writing
    // ...
}
```

To connect a file to an existing file object, we call the **`open()`** member function on the object.

For example,

```
// Connect to a File for Writing
// connectFileWriting.cpp

#include <fstream>

int main() {
    std::ofstream fout;      // create a file object named fout
    std::ofstream fout("output.txt"); // connects fout to output.txt for writing
    // ...
}
```

Confirming a File Connection

The `is_open()` member function called on a file object returns the current state of the object's connection to a file:

```
#include <iostream>
#include <fstream>

std::ofstream fout("output.txt");  // connects output.txt to fout for output

if (!fout.is_open()) {
    std::cerr << "File is not open" << std::endl;
} else {
    // file is open
    // ...

}
```

Streaming Fundamental Types

The standard input/output library overloads the extraction and insertion operators for each fundamental type for the file stream classes with a file objects as left operands.

Reading From a File

A file object reads from a file under format control using the extraction operator in the same way as the standard input object (`cin`) reads using the extraction operator.

Consider a file with a single record: `12 34 45 abc` The output from the following program is shown on the right:

```
// Reading a File
// readFile.cpp

#include <iostream>
#include <fstream>

int main() {
    int i;

    std::ifstream f("input.txt");
    if (f.is_open()) {
        while (f) {
            f >> i;
            if (f)
                std::cout << i << ' ';
            else
                std::cout << "\n**Bad input**\n";
        }
    }
}
```

```
12 34 45

**Bad input**
```

The file stream class definition overload the `bool` conversion operator to return false if the object is not ready for further streaming. A stream object is not ready for further streaming if it has encountered an error and has not been cleared. the topic of error states and clearing errors is covered later in the chapter entitled Input and Output Refinements.

Writing to a File

A file object writes to a file under format control using the insertion operator in the same way as the standard output objects (`cout`, `cerr` and `clog`) write using the insertion operator.

For example, the contents of the file created by the following program are shown on the right

```
// Writing to a File
// writeFile.cpp
```

```
#include <iostream>
#include <fstream>

int main() {
    int i;

    std::ofstream f("output.txt");
    if (f.is_open()) {
        f << "Line 1" << std::endl;   // record 1                Line 1
        f << "Line 2" << std::endl;   // record 2                Line 2
        f << "Line 3" << std::endl;   // record 3                Line 3
    }
}
```

STRING CLASS (OPTIONAL)(OPTIONAL)

The examples in these notes have been limited to input data that fits within pre-allocated memory. In the case of character string input, the user determines the number of characters to enter and pre-allocation of the required memory is not possible. A user entering more characters than allocated memory can accept may cause a stream failure.

The Problem

Consider the user inputting a comment on a student's transcript. Since we only know how much memory to allocate for the comment after receiving the complete text, we cannot allocate that memory at compile-time or run-time before accepting the comment.

The Solution

The standard library's **`string`** class allocates the required amount of memory dynamically during the input process itself. A **`std::string`** object can accept as many characters as the user enters. The helper function **`std::getline()`** extracts the characters from the input stream.

The prototype for this helper function is

```
std::istream& getline(std::istream&, std::string&, char);
```

The first parameter receives a modifiable reference to the **`std::istream`** object, the second parameter receives a modifiable reference to the **`std::string`** object and the third parameter receives the character delimiter for terminating extraction (newline by default).

The **`<string>`** header file contains the class definition with this prototype. The class definition includes two member functions for converting its internal data into a C-style null-terminated string:

- **`std::string::length()`** - returns the number of characters in the string
- **`std::string::c_str()`** - returns the address of the C-style null-terminated version of the string

C-Style Example

The following client code extracts an unknown number of characters from the standard input stream, stores them in a C-style null-terminated string and displays the character string on the standard output object in five steps:

1. define a **`string`** object to accept the input
2. extract the input using the **`std::getline()`** helper function
3. query the **`string`** object for the memory required
4. allocate dynamic memory for the requisite C-style null-terminated string
5. copy the data from the **`string`** object to the allocated memory
6. deallocate the allocated memory

```
// String Class example
// string.cpp

#include <iostream>
```

```
#include <string>

int main( ) {
    char* s;
    std::string str;

    std::cout << "Enter a string : ";
    if (std::getline(std::cin, str)) {
        s = new char [str.length() + 1];
        std::strcpy(s, str.c_str());
        std::cout << "The string entered is : >" << s << '<' << std::endl;
        delete [] s;
    }
}
```

Student Class Example

Let us upgrade our **`Student`** class to store a comment using a **`string`** object as a data member.

The header file for our **`Student`** class contains:

```
// Student.h

#include <iostream>
#include <string>

const int NG = 20;

class Student {
    int no;
    float grade[NG];
    int ng;
    std::string comment;
public:
    Student();
    Student(int);
    Student(int, const float*, int, const std::string&);
    void read(std::istream&);
    void display(std::ostream&) const;
};

std::istream& operator>>(std::istream& is, Student& s);
std::ostream& operator<<(std::ostream& os, const Student& s);
```

The implementation file contains:

```
// Student.cpp

#include "Student.h"
using namespace std;

Student::Student() {
    no = 0;
    ng = 0;
}

Student::Student(int n) {
    *this = Student(n, nullptr, 0, "");
}

Student::Student(int sn, const float* g, int ng_, const string& c) {
    bool valid = sn > 0 && g != nullptr && ng_ >= 0;
    if (valid)
```

```
            for (int i = 0; i < ng_ && valid; i++)
                valid = g[i] >= 0.0f && g[i] <= 100.0f;

        if (valid) {
            // accept the client's data
            no = sn;
            ng = ng_ < NG ? ng_ : NG;
            for (int i = 0; i < ng; i++)
                grade[i] = g[i];
            comment = c;
        } else {
            *this = Student();
        }
    }

    void Student::read(std::istream& is) {
        int no;          // will hold the student number
        int ng;          // will hold the number of grades
        float grade[NG]; // will hold the grades
        string comment;  // will hold comments

        cout << "Student Number : ";
        is >> no;
        cout << "Number of Grades : ";
        is >> ng;
        if (ng > NG) ng = NG;
        for (int i = 0; i < ng; i++) {
            cout << "Grade " << i + 1 << " : ";
            is >> grade[i];
        }
        is.ignore(); // extract newline
        cout << "Comments : ";
        getline(is, comment, '\n');

        // construct a temporary Student
        Student temp(no, grade, ng, comment);
        // if data is valid, the temporary object into the current object
        if (temp.no != 0)
            *this = temp;
    }

    void Student::display(std::ostream& os) const {
        if (no > 0) {
            os << no << ":\n";
            os.setf(ios::fixed);
            os.precision(2);
            for (int i = 0; i < ng; i++) {
                os.width(6);
                os << grade[i] << endl;
            }
            os.unsetf(ios::fixed);
            os.precision(6);
            os << "Comments:\n" << comment << endl;
        } else {
            os << "no data available" << endl;
        }
    }

    std::ostream& operator<<(std::ostream& os, const Student& s) {
        s.display(os);
        return os;
    }

    std::istream& operator>>(std::istream& is, Student& s) {
        s.read(is);
        return is;
    }
```

The client code on the left receives the input on the right and produces the output listed below:

```
// String Class
// string.cpp

#include <iostream>
#include "Student.h"

int main ( ) {
    Student harry;

    std::cin >> harry;
    std::cout << harry << std::endl;
}
```

```
Student Number : 1234
Number of Grades : 2
Grade 1 : 56.7
Grade 2 : 78.9
Comments : See Coordinator

1234:
 56.70
 78.90
Comments:
See Coordinator
```

SUMMARY

- we associate our own classes with the **iostream** classes by overloading the extraction and insertion operators as helpers to those classes
- the first parameter in the declaration of each overloaded operator is a modifiable reference to the stream object
- the return type of each overloaded operator is a modifiable reference to the stream object, which enables cascading
- the standard library includes overloaded extraction and insertion operators for file objects as left operands and fundamental types as right operands
- an input file object is an instance of an **ifstream** class
- an output file object is an instance of an **ofstream** class
- the **string** class of the standard library manages the memory requirements for storing a user-defined character string of any length

EXERCISES

- Complete the Handout on Custom I/O Operators.

print this page Top
Previous: Helper Functions Next: Inheritance

ICT | ICT | Home | Outline | Outline | Timeline
Notes | IPC Notes | IPC Notes | MySeneca | MySeneca | Workshops | Workshops | Assignments | Instructor

Last Modified: 06/20/2022 06:27Last Modified: 05/20/2017 11:50