Object-oriented Programming

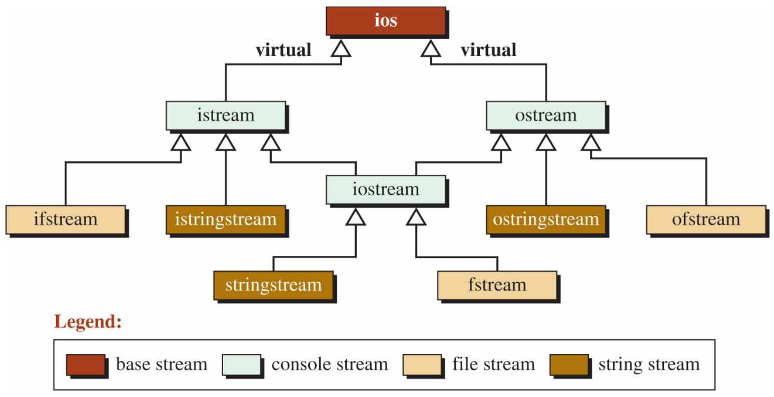
Input / Output Streams Part 3

YoungWoon Cha

Computer Science and Engineering

Stream Classes

To handle input/output operations, the C++ library defines a hierarchy of classes.



The ios Class

At the top of the hierarchy is the *ios* class that serves as a virtual base class for all input/output classes.

It defines data members and member functions that are inherited by all input/output stream classes.

Since the *ios* class is never instantiated, it does not use its data members and member functions. They are used by other stream classes.

Other Classes

For convenience, we refer to *istream*, *ostream*, and *iostream* as console classes (they are used to connect our program to the console).

We refer to *ifstream*, *ofstream*, and *fstream* as file streams (they are used to connect our program to files).

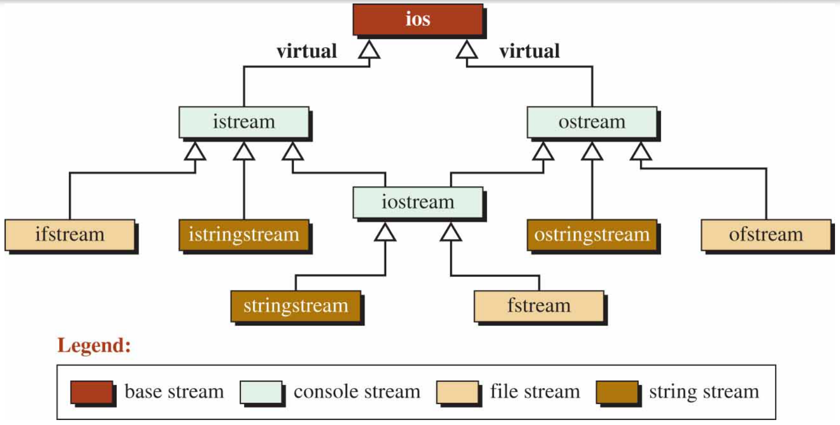
We refer to *istringstream*, *ostringstream*, and *stringstream* as string streams.

String Streams

STRING STREAMS

String streams use three classes *istringstream*, *ostringstream*, and *stringstream*.

These classes are used to input data from or output data to strings. They are inherited respectively from *istream*, *ostream*, and *iostream*.



To use the string streams, we need the <sstream> header file.

In a string stream the source or destination is in fact a string inside the program itself.

In other words, we read from an existing string in the program; we write to a string in the program.

We cannot open or close these streams because the source or sink entities connected to these streams are not external; they are created and destroyed inside our programs.

For this reason, there is no *open*() or *close*() functions for these streams.

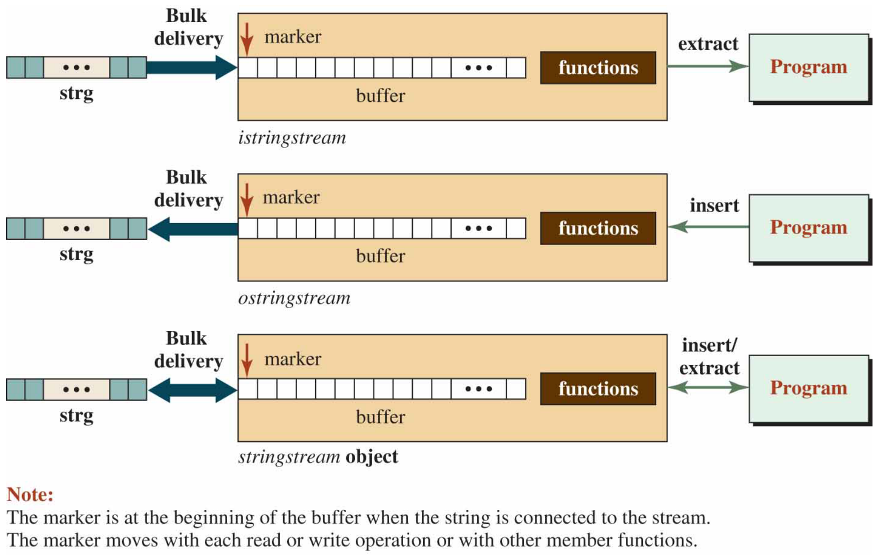
Instantiation

We show the constructors for the *istringstream*, *ostringstream*, and *stringstream* classes with the default values for the open modes.

```
Istringstream (const string strg,  ios:: openmode mod = ios :: in)
Ostringstream (const string strg,  ios:: openmode   mod =  ios :: out)
Stringstream (const string strg,  ios :: openmode   mod =  ios :: in |ios :: out)
```

Each constructor instantiates an object of type *istringstream*, *ostringstream*, or *stringstream* and connects the object to the string object defined as the first parameter.

Note that a string object needs to be included but it can be a null string.



Member Functions

Each class provides two versions of a new member function called *str*.

The first version replaces the string connected to the stream; the second version returns a copy of the string connected to the stream.

```
void str(string strg);   //Connect the parameter to the host object
string str() const;   // Returns the string connected to the host objet
```

Testing String Stream Classes

```
#include <iostream>
#include <string>
#include <sstream>
using namespace std;

int main ()
{
      // Using istringstream object
      istringstream iss ("Hello friends!");
      cout << iss.str () << endl;
      iss.str ("Hello world!");
      cout << iss.str () << endl << endl;
      // Using ostringstream object
      ostringstream oss ("Bye friends!");
      cout << oss.str () << endl;
      oss.str ("Bye world!");
      cout << oss.str () << endl;
      return 0;
}
```

```
Run:
Hello friends!
Hello world!

Bye friends!
Bye world!
```

In the first section, we first create an object of *istringstream* connected to a string and print the string.

We then change the string connected to the same *istringstream* and print the string.

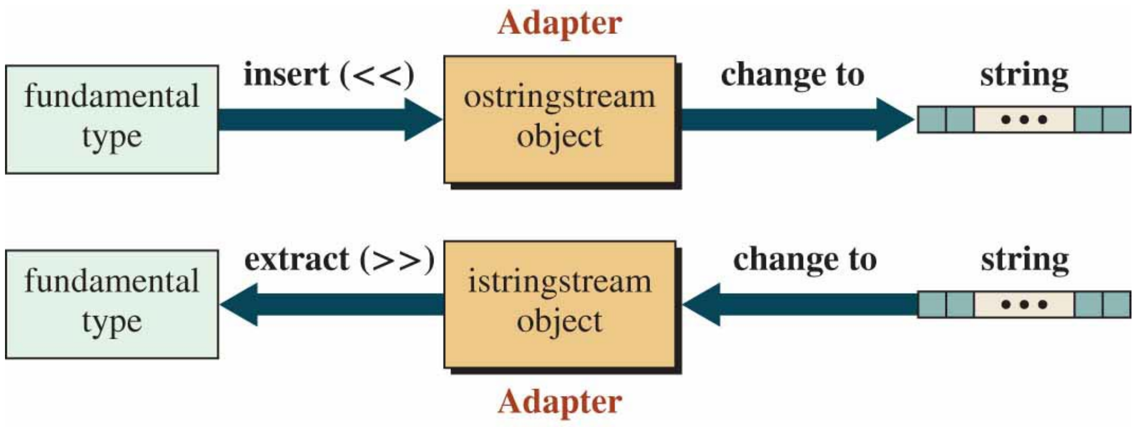
In the second section, we do the same with *ostringstream*.

Application: Adapter

The most common application of a string stream class is to act as an *adapter* for the string class.

In programming we sometimes need to convert a fundamental data type to a string and a string to a fundamental data type.

To do so we use an object of a *string stream* class as an *adapter*.



In wrapping, we insert the fundamental data type (s) into an *ostringstream* object and then change the *ostringstream* object to a string object.

In unwrapping, we change the string to an *istringstream* object and the extract the fundamental data type from the *istringstream* object.

Application Program That Uses Template Functions

```
#ifndef CONVERT_H
#define CONVERT_H
#include <iostream>
#include <string>
#include <sstream>
using namespace std;

// toString function changes any data type to string
template <typename T>
string toString (T data)
{
      ostringstream oss ("");
      oss << data;
      return oss.str ();
}
// toData function takes out the data embedded in a string
template <typename T>
T toData (string strg)
{
      T data;
      istringstream iss (strg);
      iss >> data;
      return data;
}
#endif
```

```
#include "convert.h"

int main ( )
{
      // Converting integer 12 to a string
      string strg = toString (12);
      cout << "String: " << strg << endl;
      // Converting string "15.67" to double
      double data = toData <double> ("15.67");
      cout << "Data: " << data;
      return 0;
}
```

```
Run:
String: 12
Data: 15.67
```

We can create a function template to convert any fundamental data type to a string.

We can also create a function template to return the fundamental data type embedded in a string.

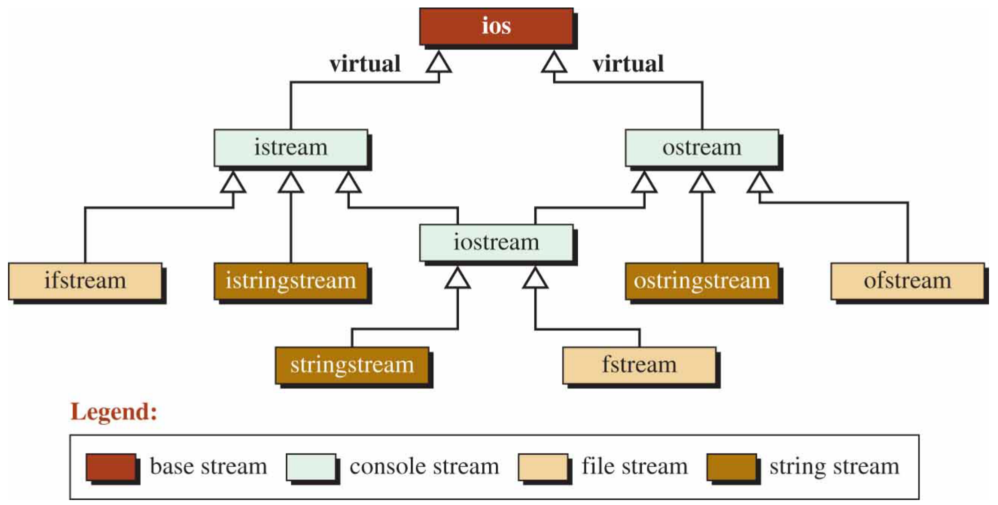
Data Formatting

FORMATTING DATA

Manipulators use the formatting data members (formatting flags, formatting fields, and formatting variables) defined in the *ios* class.

Since all stream classes inherit from the *ios* class, they all inherit these data members and the corresponding member functions to set and unset fields and store values in variables.

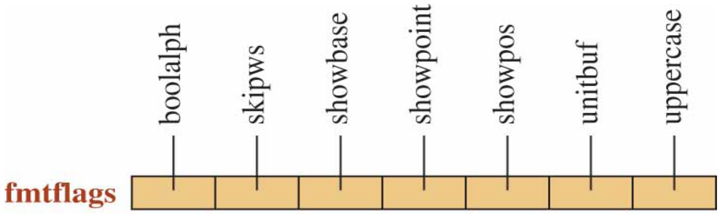
To be able to create customized manipulators, we need to understand how we can directly use these flags, fields, and variables.



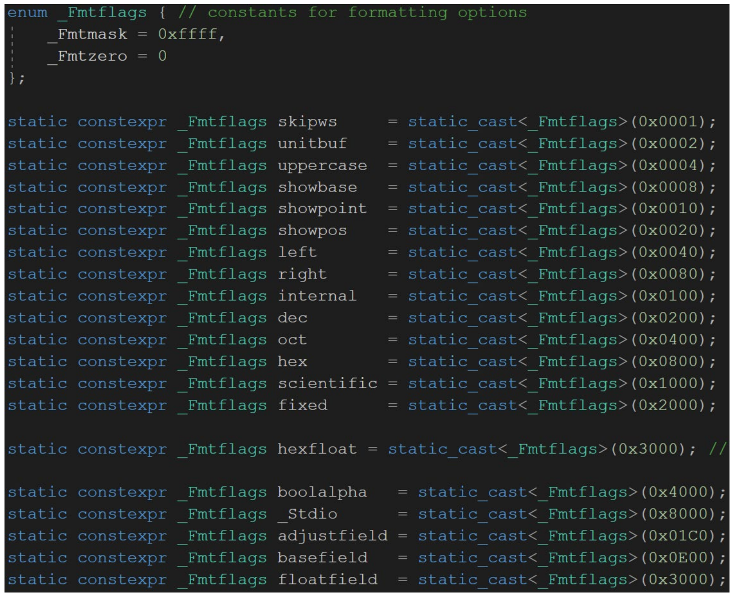
Direct Use of Flags, Fields, and Variables Part 1

Formatting Flags

The *ios* class defines a type named *fmtflags* which can take a combination of any of the seven values. Each bit can be set (value1) or unset (value 0).



Flag	Description
ios :: boolalpha	Show representation of boolean data (*true* or *false*).
ios :: skipws	Skip white space characters (default for input).
ios :: showbase	Show numeric base when printing integral values.
ios :: showpoint	Show the decimal point for floating-point values.
ios :: showpos	Show positive sign (plus sign) for positive values.
ios :: unitbuf	Flush the stream after each action.
ios :: uppercase	Show A-F for hexadecimal and E for scientific (in uppercase).



```
enum _Fmtflags { // constants for formatting options
    _Fmtmask = 0xffff,
    _Fmtzero = 0
};

static constexpr _Fmtflags skipws     = static_cast<_Fmtflags>(0x0001);
static constexpr _Fmtflags unitbuf    = static_cast<_Fmtflags>(0x0002);
static constexpr _Fmtflags uppercase  = static_cast<_Fmtflags>(0x0004);
static constexpr _Fmtflags showbase   = static_cast<_Fmtflags>(0x0008);
static constexpr _Fmtflags showpoint  = static_cast<_Fmtflags>(0x0010);
static constexpr _Fmtflags showpos    = static_cast<_Fmtflags>(0x0020);
static constexpr _Fmtflags left       = static_cast<_Fmtflags>(0x0040);
static constexpr _Fmtflags right      = static_cast<_Fmtflags>(0x0080);
static constexpr _Fmtflags internal   = static_cast<_Fmtflags>(0x0100);
static constexpr _Fmtflags dec        = static_cast<_Fmtflags>(0x0200);
static constexpr _Fmtflags oct        = static_cast<_Fmtflags>(0x0400);
static constexpr _Fmtflags hex        = static_cast<_Fmtflags>(0x0800);
static constexpr _Fmtflags scientific = static_cast<_Fmtflags>(0x1000);
static constexpr _Fmtflags fixed      = static_cast<_Fmtflags>(0x2000);

static constexpr _Fmtflags hexfloat = static_cast<_Fmtflags>(0x3000); //

static constexpr _Fmtflags boolalpha   = static_cast<_Fmtflags>(0x4000);
static constexpr _Fmtflags _Stdio      = static_cast<_Fmtflags>(0x8000);
static constexpr _Fmtflags adjustfield = static_cast<_Fmtflags>(0x01C0);
static constexpr _Fmtflags basefield   = static_cast<_Fmtflags>(0x0E00);
static constexpr _Fmtflags floatfield  = static_cast<_Fmtflags>(0x3000);
```

We can set or unset them using the *setf*() or *unsetf*() function

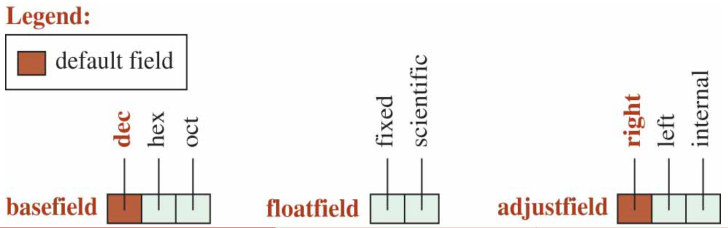
```
fmtflags ios :: setf(flag)              // It sets the corresponding flag
fmtflags ios :: unsetf(flag)   // It unset the corresponding flag
```

Direct Use of Flags, Fields, and Variables Part 2

Formatting Fields

The *ios* class also defines a group of three fields for formatting data in which each field has two or three fields that can be exclusively set or unset.

Only one bit at a time can be set.



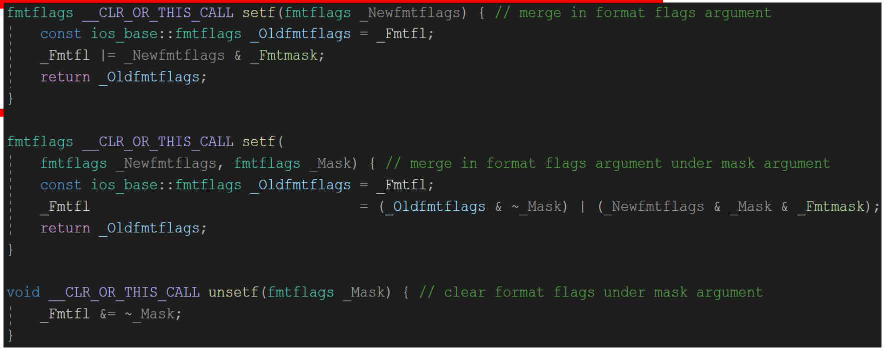
The default setting is shown in color, but note that *floatfield* has no default value (the best selected by the system if none is explicitly selected).

Flags	Values	Description
ios :: basefield	ios :: dec	Set integral values in decimal.
	ios :: hex	Set integral values in hexadecimal.
	ios :: oct	Set integral values in octal.
ios :: floatfield	ios :: fixed	Show the floating-value in fixed format.
	ios :: scientific	Show the floating-value in scientific format.
ios :: adjustfield	ios :: right	Right justify the data in the field.
	ios :: left	Left justify the data in the field.
	ios :: internal	Add fill character after the sign.

To set a field we need to use two parameters.

The second parameter (field) unsets all of the previous fields in the field; the first parameter then sets the desired field.

fmtField ios :: setf (addingField, field)
fmtField ios :: unsetf (field)

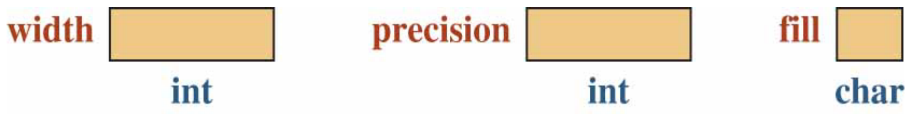


```
fmtflags __CLR_OR_THIS_CALL setf(fmtflags _Newfmtflags) { // merge in format flags argument
    const ios_base::fmtflags _Oldfmtflags = _Fmtfl;
    _Fmtfl |= _Newfmtflags & _Fmtmask;
    return _Oldfmtflags;
}

fmtflags __CLR_OR_THIS_CALL setf(
    fmtflags _Newfmtflags, fmtflags _Mask) { // merge in format flags argument under mask argument
    const ios_base::fmtflags _Oldfmtflags = _Fmtfl;
    _Fmtfl                             = (_Oldfmtflags & ~_Mask) | (_Newfmtflags & _Mask & _Fmtmask);
    return _Oldfmtflags;
}

void __CLR_OR_THIS_CALL unsetf(fmtflags _Mask) { // clear format flags under mask argument
    _Fmtfl &= ~_Mask;
}
```

Direct Use of Flags, Fields, and Variables Part 3

Formatting Variables

The formatting variables are named *width* (of type *int*), *precision* (of type *int*) and *fill* (of type *char*).



Member function	Description
int ios::width(int n)	Set the number of position to be used by a value.
int ios::width()	Reset the width field to 0
int ios::precision(int n)	Set the number of position after the decimal point.
int ios::precision()	Unset the *precision* field.
int ios::fill(char c)	Set the type of character to fill the empty position.
int ios::fill()	Unset the *fill* field.

These fields are used to define how many positions are set aside for a value, how many positions are set aside after a decimal point, and what character should be used to fill the unused positions.

Note that if the fill field is not set, the fill character defaults to a space.

Program to Print Three Data Items

```cpp
#include <iostream>
using namespace std;

int main ()
{
        // Declaration and initialization of three variables
        bool b = true;
        int i = 12000;
        double d = 12467.372;
        // Printing values
        cout << "Printing without using formatting" << endl;
        cout << "Value of b: " << b << endl;
        cout << "Value of i: " << i << endl;
        cout << "Value of d: " << d << endl << endl;
        // Formatting the Boolean data and print it again
        cout << "Formatting the Boolean data" << endl;
        cout.setf (ios :: boolalpha);
        cout << b << endl << endl;
        // Formatting the integer data and print it again
        cout << "Formatting the integer data type" << endl;
        cout.setf (ios :: showbase);
        cout.setf (ios :: uppercase);
        cout.setf (ios :: hex, ios :: basefield);
        cout.setf (ios :: right, ios :: adjustfield);
        cout.width (16);
        cout.fill ('*');
        cout << i << endl << endl;
        // Formatting the floating-point data and print it again
        cout << "Formatting the floating-point data type" << endl;
        cout.setf (ios :: showpoint);
        cout.setf (ios :: right, ios :: adjustfield);
        cout.setf (ios :: fixed, ios :: floatfield);
        cout.width (16);
        cout.precision (2);
        cout.fill ('*');
        cout << d << endl;
        return 0;
}
```

The Program shows how we can directly format three data items of type *boolean*, *integer*, and *floating-point*.

Although it is possible to format data using the formatting flags, fields, and variables, it shows lengthy.

```
Run:
Printing without using formatting
Value of b: 1
Value of i: 12000
Value of d: 12467.4

Formatting the Boolean data
true

Formatting the integer data type
**********0X2EE0

Formatting the floating-point data
type
********12467.37
```

Predefined Manipulators Part 1

Manipulators Related to Formatting Flags

We have exactly fourteen formatting-flag manipulators (one each for setting and unsetting) of those seven flags. The ones in color are the default.

Flag	Manipulator	Effect	In	Out
boolalpha	noboolalpha boolalpha	Show Boolean values *as* 0/1. Show Boolean values *false*/*true*.	√ √	√ √
skipws	noskipws skipws	Do not skip whitespace in input. Skip whitespace in input.	√ √	
showbase	noshowbase showbase	Do not show the base of decimal. Show the base of decimal show.		√ √
showpoint	noshowpoint showpoint	Do not show the decimal point. She the decimal point.		√ √
showpos	noshowpos showpos	Do not show the positive sign (+). Show the positive sign (+).		√ √
unitbuf	nounitbuf unitbuf	Do not flush the output. Flush the output.		√ √
uppercase	nouppercase uppercase	Do not show char in uppercase Show char in uppercase		√ √

Predefined Manipulators Part 2

Manipulators Related to Formatting Fields

We have exactly eight manipulators related to the fields. The ones in color are the default.

Field	Manipulator	Effect	In	Out
basefield	dec Hex oct	Show integers in decimal. Show integers in hexadecimal. Show integer in hexadecimal.	√ √ √	√ √ √
floatfield	fixed scientific	Show type in fixed format. Show type in scientific format	√ √	
Adjustfield	right left internal	Right justify the data in the field. Left justify data in the field. Justify data internally in the field.		√ √ √

Manipulators Related to Formatting Variables

We have three manipulators that look like a function with parameters in this group.

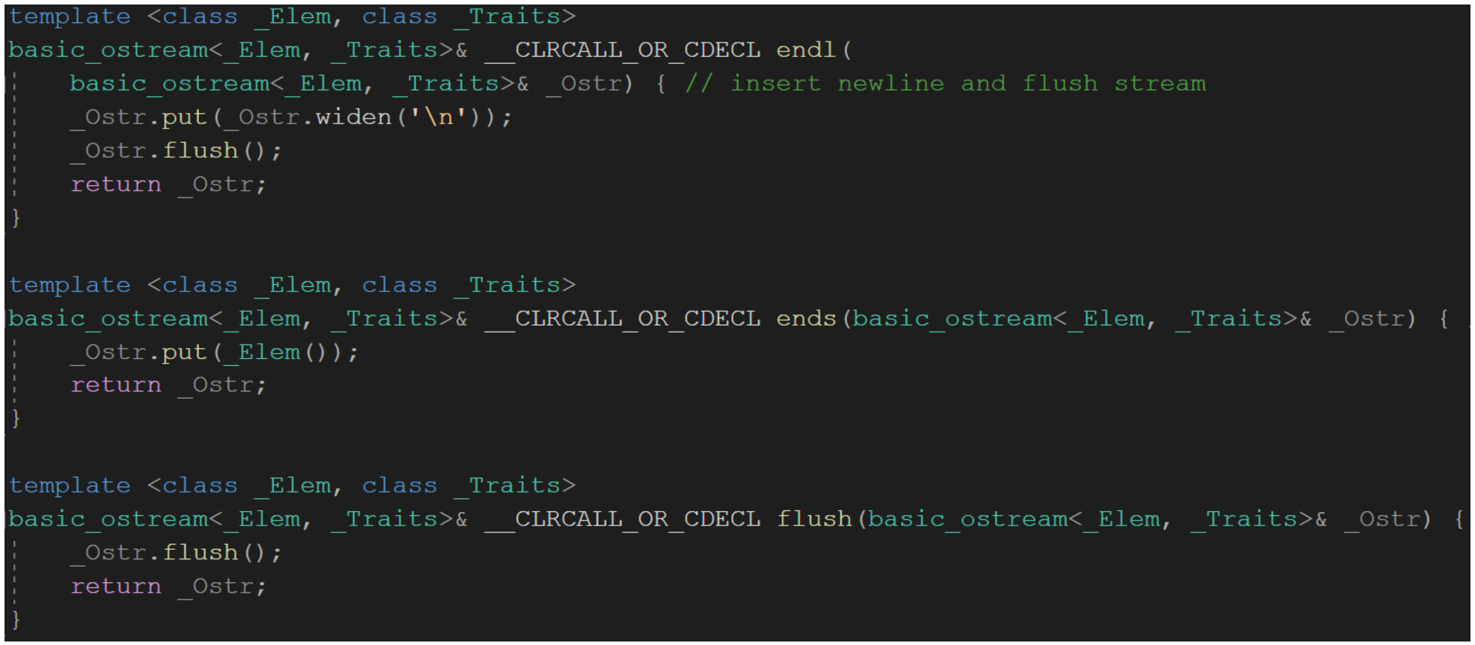
Variable	Manipulator	Effect	In	Out
width	setw(n)	Show integers in decimal.	√	√
precision	setkprecision(n)	Show type in fixed format.		√
field	setfill(c)	Right justify the data in the field.	√	√

Predefined Manipulators Part 3

Manipulators Not Related to Flags or Fields

There are four manipulators that are not related to any flag or field. They perform specific actions on the stream.

Manipulator	Effect	In	Out
ws	Extract white spaces.	√	
endl	Insert new line and flush the buffer.		√
ends	Insert end of string character.		√
flush	Flush stream buffer.		√



```
template <class _Elem, class _Traits>
basic_ostream<_Elem, _Traits>& __CLRCALL_OR_CDECL endl(
    basic_ostream<_Elem, _Traits>& _Ostr) { // insert newline and flush stream
    _Ostr.put(_Ostr.widen('\n'));
    _Ostr.flush();
    return _Ostr;
}

template <class _Elem, class _Traits>
basic_ostream<_Elem, _Traits>& __CLRCALL_OR_CDECL ends(basic_ostream<_Elem, _Traits>& _Ostr) {
    _Ostr.put(_Elem());
    return _Ostr;
}

template <class _Elem, class _Traits>
basic_ostream<_Elem, _Traits>& __CLRCALL_OR_CDECL flush(basic_ostream<_Elem, _Traits>& _Ostr) {
    _Ostr.flush();
    return _Ostr;
}
```

Program Using Manipulators for Formatting

```cpp
#include <iostream>
#include <iomanip>
using namespace std;

int main ()
{
	// Declaration and initialization of three variables
	bool b = true;
	int i = 12000;
	double d = 12467.372;
	// Printing values
	cout << "Printing without using formatting" << endl;
	cout << "Value of b: " << b << endl;
	cout << "Value of i: " << i << endl;
	cout << "Value of d: " << d << endl << endl;
	// Formatting the boolean data and print it again
	cout << "Formatting the Boolean data" << endl;
	cout.setf (ios :: boolalpha);
	cout << boolalpha << b << endl << endl;
	// Formatting the integer data and print it again
	cout << "Formatting the integer data type" << endl;
	cout << showbase << uppercase << hex << right
	<< setw (16) << setfill ('*') << i << endl << endl;
	// Formatting the floating-point data and print it again
	cout << "Formatting the floating-point data type" << endl;
	cout << showpoint << right << fixed << setw (16)
	<< setprecision (2) << setfill ('*') << d << endl << endl;
	return 0;
}
```

We repeat previous program but use manipulators instead of formatting flags, fields, and variables to achieve the same goal.
Note that we need to include the header file <iomanip> for manipulators related to the formatting variables.

```
Run:
Printing without using formatting
Value of b: 1
Value of i: 12000
Value of d: 12467.4


Formatting the Boolean data
true


Formatting the integer data type
**********0X2EE0
Formatting the floating-point data type
********12467.37
```

Manipulators Definition

If we want to create new manipulators, we must first learn how pre-defined manipulators are defined.

Manipulator without Arguments

It is not very difficult to find out how manipulators without arguments are implemented.

The system has two overloaded operators with one argument that take a pointer to function as shown below.

```
istream& istream :: operator>>(istream& (*pf)(istream&));
ostream& ostream :: operator<<(ostream& (*pf)(ostream&));
```

Each manipulator without an argument can be implemented as a function as shown below.

The name of the function is passed as a pointer to function.

```
istream& name(itream& is)
{
      action;
      return is;
}
```

```
ostream& name(ostream& os)
{
      action;
      return os;
}
```

A manipulator with no argument can be created using a function that takes a stream parameter and returns a stream.

Program with Customized Manipulators

```
#include <iostream>
using namespace std;

// Defining a function named alpha
ostream& alpha (ostream& os)
{
      os.setf (ios :: boolalpha) ;
      return os;
}
// Defining a function named noalpha
ostream& noalpha (ostream& os)
{
      os.unsetf (ios :: boolalpha) ;
      return os;
}

int main ()
{
      // Declaration and initialization of two boolean
      variables
      bool b1 = false;
      bool b2 = true;
      // Printing values of variables with alpha and noalpha
      manipulators
      cout << alpha << b1 << " " << b2 << endl;
      cout << noalpha << b1 << " " << b2 << endl;
      return 0;
}
```

```
Run:
false true
0 1
```

```
#include <iostream>
#include <iomanip>
using namespace std;

ostream& currency (ostream& stream)
{
      cout << '$';
      stream.precision (2);
      stream.fill ('*');
      stream.setf(ios:: fixed, ios:: floatfield);
      return stream;
}

int main ( )
{
      cout << currency << setw (12) << 12325.45 << endl;
      cout << currency << setw (12) << 0.36 << endl;
      return 0;
}
```

```
Run:
$****12325.45
$********0.36
```

Manipulator with Arguments

Manipulator with Arguments

Simulation of manipulators with arguments is a little more involved.

A pointer to function cannot have an argument. In other words, we cannot have a pointer to a function with a parameter, such as *setw*(4); the pointer can only be named *setw*.

This means we need to change our strategy.

Instead of a pointer to function, we can consider the phrase *setw*(4) as a call to a constructor of a class with only one data member of type integer.

In other words, we interpret the following statement.

```
cout << setw(4);
```

As an operator with two operands.

The first operand is an instance of type *ostream*; the second is an instance of type *setw*.

Manipulator with Arguments Part 2

Now we can create a manipulator with an argument using the following steps:

1. **We need to have a class whose name is the same as the name of the manipulator and with one data member of the same type as the type of manipulator argument.**

2. **We need to overload the insertion or extration operator for the class in which the first parameter is of type stream (*ostream* or *istream*) and the second parameter is the same as the class in step 1.**

3. **In the body of the overloaded operator, we code the action to achieve the purpose of the manipulator.**

A manipulator with an argument can be created by defining a class with one data member in which the insertion or extraction operator is overloaded.

Interface File for a Class Named Length

```cpp
#ifndef LENGTH_H
#define LENGTH_H
#include <iostream>
using namespace std;

class length
{
      private:
            int n;
      public:
            length (int n);
            friend ostream& operator << (ostream& stream, const length& len );
};
#endif
```

```cpp
#include "length.h"

int main ()
{
        cout << length (10) << 123 << endl;
        cout << length (20) << 234 << endl;
        return 0;
}
```

Run:

```
       123
                 234
```

```cpp
#include "length.h"

// Definition of the length member function
length :: length (int n1)
: n (n1)
{
}
// Overloaded operator <<
ostream& operator << (ostream& stream, const length& len )
{
      stream.width (len.n);
      return stream;
}
```

Let us simulate the manipulator *setw*(n), but we call it *length*(*n*) to be treated as a new manipulator.

The Program shows the interface for the class *length* (we use a lowercase letter for the name of the class to be consistent with the manipulator).

Summary

Highlights

- ❑ Console Streams
- ❑ File Streams
- ❑ String Streams
- ❑ Data Formatting

End of Class

Thank you

E-mail: youngcha@konkuk.ac.kr